Introducing Pattern Reuse in the Design of Multi-Agent Systems

Massimo Cossentino[1], Piermarco Burrafato[2], Saverio Lombardo[2], Luca Sabatucci[2]

[1] ICAR/CNR – Istituto di Calcolo e Reti ad Alte Prestazioni / Consiglio Nazionale delle Ricerche
c/o CUC, Viale delle Scienze, 90128 Palermo, Italy
cossentino@cere.pa.cnr.it

[2] DINFO - Dipartimento di Ingegneria Informatica, Università degli Studi di Palermo
Viale delle Scienze, 90128 Palermo, Italy

Abstract. In the last years, multi-agent systems (MAS) have proved more and more successful. The need of a quality software engineering approach to their design arises together with the need of new methodological ways to address important issues such as ontology representation, security concerns and production costs. The introduction of an extensive pattern reuse practice can be determinant in cutting down the time and cost of developing these systems. Patterns can be extremely successful with MAS (even more than with object-oriented systems) because the great encapsulation of agents allows an easier identification and disposition of reusable parts. In this paper we discuss our approach to the pattern reuse that is a phase of a more comprehensive approach to agent-oriented software design.

1 Introduction

In the last years, multi-agent systems (MAS) have proved successful in more and more complex duties; 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 multi-agent systems considering that this activity implies not only modelling an agent in place of an object but also capturing the ontology of its domain, representing its interaction with other agents, and providing it with the ability of performing intelligent behaviours. 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 [12]) others from Software Engineering (MaSE [11], Tropos [24]) but there are also methodologies coming directly from Robotics (Cassiopeia [13]). 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.

We can consider that the process of designing a MAS is not very different from other software design processes, if we look at the process inputs and outputs. In order to increase the results, we think that an important role can be played by an analysis of the inputs and of the activities to be performed as well as 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 method for developing multi-agent software that integrates design models and philosophies from both object-oriented software engineering and MAS using UML notation. It has evolved from a long period of theory construction [16][1][2] and experiments in the development of embedded robotics applications [3][4][15][17] and now is the design process used in a more comprehensive approach to robotics that encompasses a flexible vision architecture and an extensive modelling of environmental knowledge and ontology [6]. Moreover, it also proved successful in designing information systems [19].

The design process is composed of five models (see Fig. 1): the System Requirements Model is an anthropomorphic model of the system requirements in terms of agency and purpose; the Agent Society Model is a model of the structure of the agents involved in the solution, of their social interactions and dependencies; the Agent Implementation Model is a model of the solution architecture in terms of classes and methods; the Code Model is a model of the solution at the code level and the Deployment Model is a model of the distribution of the parts of the system (agents) across hardware processing units, and their movements across the different available platforms.

In PASSI great importance has the reuse of existing patterns. We define a pattern as a representation and implementation of some kind of (a part of) the system behaviour. Therefore each pattern in our approach is composed of a model of (dynamic) behaviour, a model of the structure of the involved elements, and the implementation code.

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 agent in their world.

While they face the problem domain they need to determine the functionalities required for the system, identify the agents (assigning the previously identified functionalities to them) and their roles, represent the ontology of the domain, and describe the agents' communication. We have not introduced an explicit model of the goals of the system because several contributes can already be found in literature ([7][8][9][10]), and can be used in order to perform this activity.

In the solution domain the designer essentially produces some representations of the structure of the agents and of their dynamic behaviour. From this specification, the designer (or more likely the programmer), after having chosen the implementation architecture, produces the code and deploys it as described in the deployment model.

It is in this progress of activities, mainly looking at the work performed in the solution domain, that we identified the most useful structure for our patterns: one structural representation of the agent (a class diagram), one dynamical representation of the behaviour expressed by the agent(s) (an activity diagram) and the corresponding code.

We propose a classification of our patterns in four different categories: the action pattern (a functionality expressed by an agent – for example a specific task – usually it represents only a portion of the agent), the behaviour pattern (again a portion of an agent but it addresses a more complex functionality, often performed by the agent using more than one of its tasks), a component pattern (a complete agent capable of performing some kind of behaviours), a service pattern (composed by at least two component patterns where the involved agents interact in order to actuate some kind of cooperative behaviour).

Now we are working on the Agent Factory Project funded within the Agentcities initiative [22], and our goal is to implement a service for the network community, that is composed of a pattern-based agent production process and a repository that will contain the patterns that we will identify and many others that will be introduced by other members of the community.

In order to support the localization of our patterns in both of the two different most diffused FIPA [18] platforms (FIPA-OS [14] and JADE [23]) we are planning to represent the models and the code of each pattern using XML. In the case of the models we will use the diffused XMI representation of the UML diagrams while for the code we will introduce a meta-representation of the agent using XML. Then applying to it an XSLT transformation we will instantiate the code localized for the selected platforms. Obviously this approach is possible because FIPA-OS and JADE are based on the Java language and share a similar structure, while the solution could become more difficult without these favourable conditions.

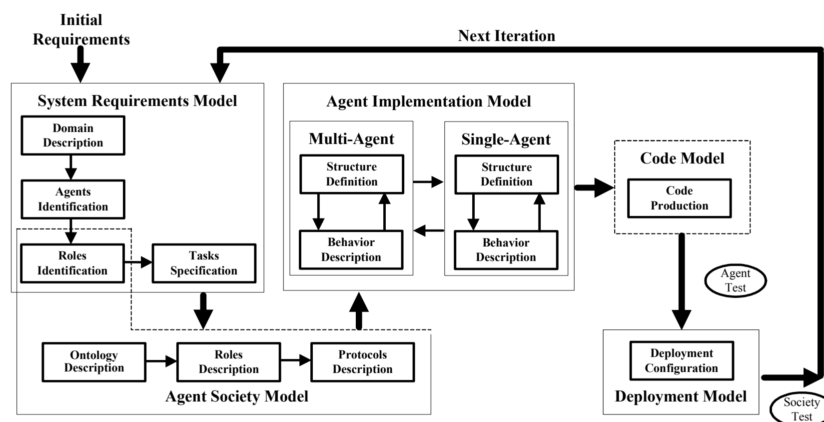

Fig. 1. The models and phases of the PASSI methodology

The remaining part of the paper is organized as follows: section 2 gives a quick overview of the PASSI methodology; section 3 presents the Agent Factory Project; section 4 provides a discussion on patterns; meta-language representation of agents is discussed in section 5, and some conclusions are presented in section 6.

2 The PASSI Methodology

PASSI [5] is composed of five models that address different design concerns and twelve steps in the process of building a model.

In PASSI we use UML as the modelling language because it is widely accepted both in the academic and industrial worlds. Its extension mechanisms (constraints, tagged values and stereotypes) facilitate the customized representation of agent-oriented designs without requiring a completely new language.

The models and phases of PASSI are (see Fig. 1):

1. **System Requirements Model**. An anthropomorphic model of the system requirements in terms of agency and purpose. Developing this model involves four steps: *Domain Description (D.D.)*: A functional description of the system using conventional use-case diagrams. *Agent Identification (A.Id.)*: Separation of responsibility concerns into agents, represented as stereotyped UML packages. *Role Identification (R.Id.)*: Use of sequence diagrams to explore each agent's responsibilities through role-specific scenarios. *Task Specification (T.Sp.)*: Specification through activity diagrams of the capabilities of each agent.

2. **Agent Society Model**. A model of the social interactions and dependencies among the agents involved in the solution. Developing this model involves three steps in addition to part of the previous model: *Role Identification (R.Id.)*. See the System Requirements Model. *Ontology Description (O.D.):* Use of class diagrams and OCL constraints to describe the knowledge ascribed to individual agents and the pragmatics of their interactions. *Role Description (R.D.)*: Use of class diagrams to show distinct roles played by agents, the tasks involved that the roles involve, communication capabilities and inter-agent dependencies. *Protocol Description (P.D.)*: Use of sequence diagrams to specify the grammar of each pragmatic communication protocol in terms of speech-act performatives like in the AUML approach [25].

3. **Agent Implementation Model**. A model of the solution architecture in terms of classes and methods, the development of which involves the following steps: *Agent Structure Definition (A.S.D.)*: Use of conventional class diagrams to describe the structure of solution agent classes. *Agent Behaviour Description (A.B.D.)*: Use of activity diagrams or state charts to describe the behaviour of individual agents.

4. **Code Model**. A model of the solution at the code level requiring the following steps to produce: *Code Reuse Library (C.R.)*: A library of class and activity diagrams with associated reusable code. *Code Completion Baseline (C.C.)*: Source code of the target system.

5. **Deployment Model**. A model of the distribution of the parts of the system across hardware processing units, and their migration between processing units. It involves one step: *Deployment Configuration (D.C.)*: Use of deployment diagrams to

describe the allocation of agents to the available processing units and any constraints on migration and mobility.

Testing: the testing activity has been subdivided into two different steps: the (single) agent test is devoted to verifying its behaviour with regards to the original requirements of the system solved by the specific agent. During the society test, the validation of the correct interaction of the agents is performed, in order to verify that they concur in solving problems that need cooperation.

2.1 The Support of The CASE Tool in Designing with PASSI

In this section we describe the CASE tool we have developed for designing multi-agent systems following the PASSI methodology.

Our work starts from the consideration that most commercial CASE tools are only object-oriented. Besides, the design of a MAS is often very difficult for unskilled users. We believe that the support of an agent-oriented CASE tool can simplify the MAS designer's work, increase the reuse of code (through a database of agents/tasks patterns), and permit the automatic production of a considerable part of the code. Moreover, our tool helps untrained users to follow a proper software engineering approach.

We have realized our tool by building an add-in for the commercial UML-based CASE tool Rational Rose. It enables the user to follow the PASSI's process of analysis and design, providing a set of functionalities that are specific for each phase of the process by means of sub- and pop-up menus that appear after having selected some UML elements (classes, use cases and so on). The tool also allows the designers to perform check operations, which are based on correctness of single diagrams and consistency between related steps and models. The main functionalities of our tool are as follows:

Automatic compilation of diagrams: Our Rose add-in allows us to save analysis and design time by totally or partially drawing some diagrams in an automatic way. This enables designers to go through the twelve steps of PASSI in a very fast and easy way. For example, the Agent Identification is totally drawn by the tool once the user has chosen what functionality to insert into an agent and a consistency check is positively performed. This simple identification of an agent triggers a series of automatic operations: a) a Task Specification diagram is assigned to the created agent; b) the model of the agent skeleton is depicted in the Multi- and Single-Agent Structure Definition (SASD); c) a Single-Agent Behaviour Description (SABD) diagram is assigned to the new agent; and so forth. Other diagrams, such as Communication Ontology Description, Roles Description and Single-Agent Structure Definition are partially drawn as pieces of new information are gradually inserted into the PASSI's models.

Automatic support to execute recurrent operations: Apart full and partial assembling of diagrams, the tool enable developers to also modify the model at any point and obtain the automatic update of all of the diagrams that depend on the modification.

Project consistency: In general, the tool permits a check of the model. When it is invoked, it verifies the entire model correctness and consistency between the diverse diagrams yielded till that point. Furthermore, a check operation is automatically run whenever the user completes any phase. The check will inform the user about any error or inconsistency.
Automatic compilation of reports and design documents: The add-in can produce a report of the entire model in a Microsoft Word format. Together with the diagrams, the document will contain textual descriptions and some tables summarizing the agent tasks, roles, communication, ontology, etc.
Access to a database of patterns: We intend to provide the tool with a repository of pattern as described in the next sections.
Generation of code and Reverse Engineering: The Rose add-in can generate code from the diagrams of the implementation model. The code that is produced is actually an agent skeleton written in Java, including tasks subclasses. Furthermore, our add-in enables reverse engineering by the creation of a Single-Agent Structure Definition diagram in the Rose model from a source Java file. During this operation, the tool will refresh all the related diagrams. More consistent parts of code will be automatically produced with the introduction of the patterns.

3 The Agent Factory Project

The Agent Factory Project is an Agentcities.NET [22] deployment project. Our commitment is to deploy a service that provides Agentcities' designers with a tool for either building new agents and/or upgrading their existing ones. The agents are meant to be FIPA compliant that means they need to belong to platforms such as Jade or FIPA-OS. The key issue of the project is to produce a repository of agents' patterns (see next section) that will be described as pieces of model and code.

We intend to provide the Agentcities Network with a web-based application that enables the MAS designers to easily build their own agents and upgrade them. As for the *building operation*, the users will be able to select either the agent platform for deployment – mainly FIPA-OS and JADE – and the functionalities they want to introduce from the repository. As for the *upgrading operation*, the users will be able to input their UML models of an agent and add new capabilities from the repository so as to get back the upgraded models. For both operations, the application will eventually provide some validation mechanisms for designers' inputs.

Agent Factory is thought to accelerate analysis and design phases of multi-agent systems by easy reuse of patterns to be identified and plugged into the project's models and/or the code. Its two main features – creation of new agents and upgrading of existing ones – will allow multi-agent systems designers to speed up prototyping.

As most Agentcities' members are concerned with the world of agents, and most of their teams are involved in the development of multi-agent systems and platforms, we believe that the service we are going to develop will be a valid support for their work, allowing them to obtain relevant benefits in terms of productivity.

Among the other things, our work will also focus on the possibility of giving a contribution to standardization activities such as those of FIPA. Building up a large

repository of patterns may render us important information about their possible generalization. This could make us address feasible ways of standardization for patterns of agents, and explore the opportunity of proposing a related specification for FIPA.

With regard to the solution strategy, we imagine a pattern as a couple of diagrams: a structural and a behavioural diagram (see next section). Users will be able to input their original agents' models in the XMI format, which will then be transformed into XML representations of the agents (see section 5). This will provide an easy way to instantiate the target code from such a structured representation of data.

4 Patterns

Regarding the patterns of agents, many works have been proposed (among the others, [20][21]); as already discussed, our concept of pattern addresses an entity composed of portions of design and the corresponding implementation code. We look at an agent as a composition of a base agent class and a set of task classes – this is the *structure*. The behaviour expressed by the agents using their structural elements can be detailed in a dynamic diagram such as an activity/state chart diagram – this is the *behaviour*.

From the structural point of view, we consider four classes of patterns. They are described as follows:

- *Action pattern.* A functionality of the system; it may be a method of either an agent class or a task class.
- *Behaviour pattern.* A specific behaviour of an agent; we can look at it as a collection of actions; it represents a task.
- *Component pattern.* An agent pattern; it encompasses the entire structure of an agent together with its tasks.
- *Service pattern.* A collaboration between two or more agents; it is an aggregation of components.

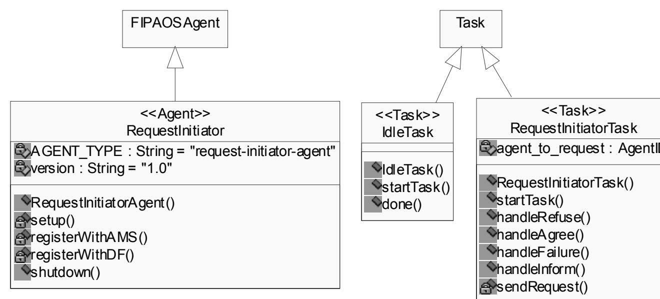

Fig. 2. Static structure of an agent and its tasks

As we know, some elementary pieces of behaviour are largely used. For example, if two agents communicate using one of the FIPA standard protocols, the parts of code devoted to dealing with communication can be reused. On the one hand, we can consider tasks as encapsulating behaviours that can be put in patterns (patterns of behaviour). On the other hand, if we consider an agent as an entity capable of pursuing specific goals, carrying out some operations (e.g. communication, moving across platforms, getting some hardware resources), we see that we can also identify patterns of agents. Furthermore, we may put together two or more patterns of agents to obtain a pattern of service. Thus, we can access and use single patterns or a composition of them.

It is now important to highlight that we can identify some specific patterns for some specific fields of application. For example, as for robotics, patterns of tasks may be useful to reuse common behaviours like planning and obstacle avoidance. Hence, it turns out that some application domain classification needs to be done. Looking at the functionality of the patterns, we can consider four categories:

- *Mobility*. These patterns describe the possibility for an agent to move from a platform to another, maintaining its knowledge.
- *Communication*. They represent the solution to the problem of making two agents communicate by a communication protocol.
- *Elaboration.* They are used to deal with the agent's functionality devoted to perform some kind of elaboration on relevant amounts of data.
- *Access to hardware resources.* They deal with information retrieval and manipulation of source data streams coming from hardware devices, such as cameras, sensors, etc.

The Repository of Patterns will be structure as a database that grants easy update and retrieval of patterns' models. As stated above, we are going to represent the latter as XML files. Each pattern record in the database will have a field containing a link to the related XML file. The repository will provide some mechanisms to ensure that coherence rules are met.

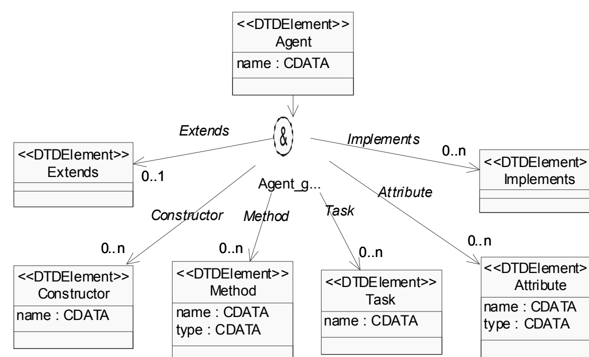

Fig. 3. The DTD related to the agent structure

5 Meta-language representation of agents

Although patterns represent a good technique for generalization, we however need to notice that they may depend on the particular target programming language. This is also true if we talk of multi-agent systems, as different agent platforms and frameworks exist.

In our work we have adopted the FIPA-OS and the Jade frameworks. As both of them present agent structures as classes containing attributes, methods and inner-classes, we have thought to adopt a hierarchical meta-language to describe the agent and his properties. The hierarchy's root level entity is the agent, which contains inherent properties such as attributes, methods and tasks (the inner classes). We have chosen XML as our meta-language as it is oriented to tree data structure representations. It proved very useful for managing agents and tasks. As a matter of fact, this allowed us to easily manipulate their structure and add, edit or delete agents' properties very easily. This is the key point in the application of a pattern to an existing agent in terms of skills and behaviors. We believe that the use of a meta-language can give us a straight way to create and maintain the Repository of Patterns mentioned above. Moreover, agents' source code can be automatically generated from meta-language representation without any manual support.

The choice of XML is also valid in the context of the Agent Factory Project. As a matter of fact, because of the XML easy portability, agents' patterns will be shared in a web server, so that designers of the Agentcities community will be able to access and update them. In what follows we describe the main issues of our meta-language:

Language definition: In order to build an XML agent representation, we retrieve information coming from diagrams such as the SASD and MABD of the PASSI design methodology. In the Single-Agent Structure Definition diagram (see Fig. 2) an agent is represented by a class, which inherits from a super class called either FIPAOSAgent or Agent depending on the agent platform selected. Attributes and methods of this class correspond respectively to data structure and functions that the agent owns. In the XML file an agent is described inside an Agent tag. The DTD fraction that describes the Agent tag is shown in Fig. 3. Agent properties such as attributes or tasks are represented as inner elements of the structure.

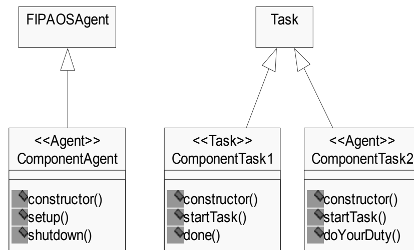

Fig. 4. The structural representation of a very simple pattern

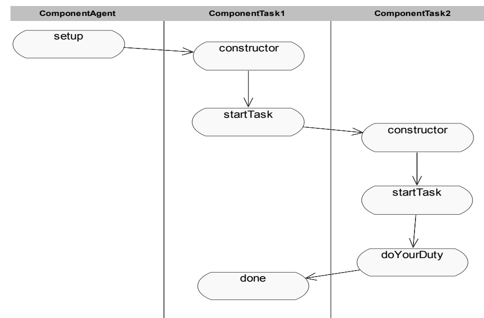

Fig. 5. The behavioural representation of the pattern of Fig. 4

Each of them contains other sub-elements that describe their properties. In the same manner, a Task tag has got sub-elements to specify its characteristics. For example the Parent tag describes parent-child relationships between tasks. This specifies that a task (called parent) instantiates and executes another task (child) from its methods as it has been specified in the Multi-Agent Behaviour Description (MABD) diagram. In FIPA-OS this information could be used to automatically add the related *done* method to the parent task; this method will be called from the task manager when the child task terminates its duty.
In Fig. 4 we can see the structural representation (a SASD diagram) of a toy-pattern of FIPA-OS agent that can be useful to understand our approach. It is composed of the base agent class and two task classes. The behaviour of the agent is described in Fig. 5: the setup method of the agent is invoked from its constructor. It calls the first task that performs some kind of operation and then starts the second task. At the end of the second task the *done* method is invoked in the parent task. The consequent XML description of this agent is shown in Fig. 6.
Patterns and constraints: When a pattern is applied to a project it modifies the context in which it is placed, that is: it introduces new functionality into the system. These additions need to satisfy some constraints. For example 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 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 from 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.

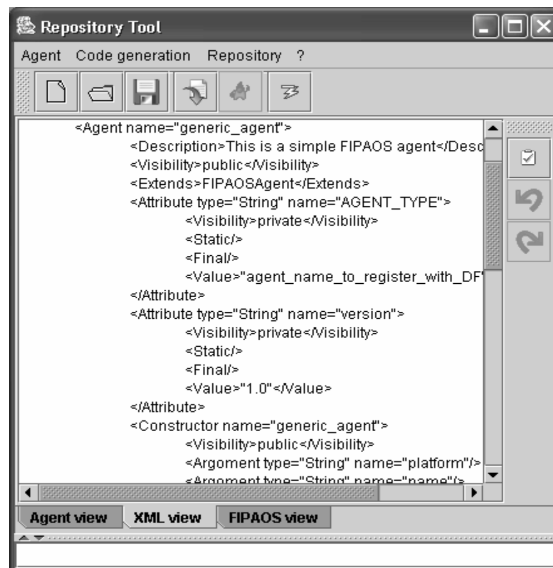

Fig. 6. The XML representation of the pattern described in fig. 4.

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 PASSI's Agent Structure Definition intrinsically represents an implementation viewpoint. As a matter of fact, UML classes correspond to Java classes, and UML attributes and methods correspond to the Java classes' attributes and methods. This important characteristic of the agent structure allows us to look at the source code as one of the possible views of an agent: we could imagine 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. Although using FIPA-OS and Jade implies different design processes, because of different mechanisms (e.g., message handling or task execution control), the same meta-language could be used to represent agents independently from the used platform.

In fig. 7 we can see the JAVA code of the toy-pattern presented in fig. 4 obtained applying the FIPA-OS transformation sheet.

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 previous experience made in fields such as robotics, that we can succeed in creating a very useful service for the Agentcities community.

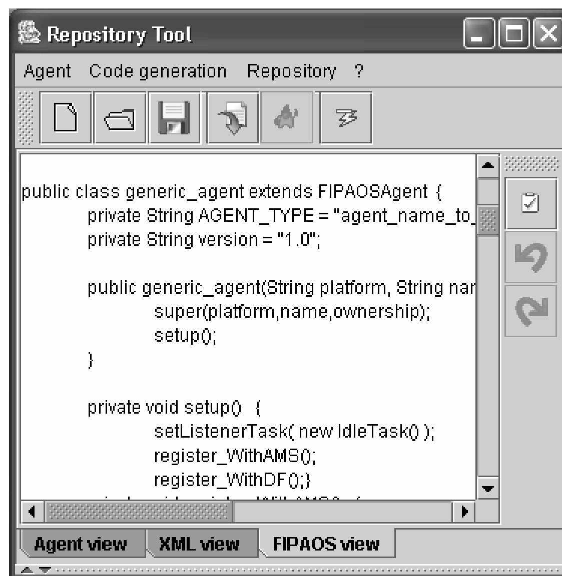

Fig. 7. (Part of) The JAVA code obtained from the XML agent description of fig. 6.

Taking advantage of others projects we are at present working on, we think it is feasible to create a repository that could contain patterns coming from diverse fields of research and application – among the others, image processing and robotics. We are also confident that the contribute of Agentcities members will be precious in order to quickly broaden our database to include more and more useful elements.

Acknowledgements

This research was partially supported by grants from Engineering Ingegneria Informatica S.p.A, Rome (Italy) and the Agentcities.NET initiative [22].

References

1. Chella, A., Cossentino, M., Lo Faso, U.: Designing agent-based systems with UML. Proc. of ISRA'2000. Monterrey, Mexico, Nov. 2000
2. Chella, A., Cossentino, M., Infantino, I., Pirrone, R.: An agent based design process for cognitive architectures in robotics. Proc. of Workshop on Objects and Agents, WOA'01. Modena, Italy, Sept. 2001
3. Chella, A., Cossentino, M., Tomasino, G.: An environment description language for multirobot simulations. Proc. of ISR 2001. Seoul, Korea, Apr. 2001
4. Chella, A., Cossentino, M., Pirrone, R., Ruisi, A.: Modeling Ontologies for Robotic Environments. Proc. of the Fourteenth International Conference on Software Engineering and Knowledge Engineering. Ischia, Italy, July 2002
5. Cossentino, M., Potts, C.: A CASE tool supported methodology for the design of multi-agent systems. Proc. of the 2002 International Conference on Software Engineering Research and Practice (SERP'02). Las Vegas, NV, USA, June 2002
6. Infantino, I., Cossentino, M., Chella, A.: An Agent Based Multilevel Architecture for robotics vision systems. Proc. of the 2002 International Conference on Artificial Intelligence (IC-AI'02). Las Vegas, NV, USA, June 2002
7. Antón, A.I., Potts, C.: The Use of Goals to Surface Requirements for Evolving Systems. Proc. of International Conference on Software Engineering (ICSE '98). Kyoto, Japan, April 1998, 157-166
8. van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. Proc. 2nd International Symposium on Requirements Engineering (RE'95).York, UK, March 1995, 194-203
9. Potts, C.: ScenIC: A Strategy for Inquiry-Driven Requirements Determination. Proc. of IEEE Fourth International Symposium on Requirements Engineering (RE'99). Limerick, Ireland, June 1999, 58-65
10. Yu, E., Liu, L.: Modelling Trust in the i* Strategic Actors Framework. Proc. of the 3rd Workshop on Deception, Fraud and Trust in Agent Societies at Agents2000. Barcelona, Catalonia, Spain, June 2000
11. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent Systems Engineering. International Journal on Software Engineering and Knowledge Engineering 11, 3, 231-258
12. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems. 3,3 (2000), 285-312

13. Collinot, A., Drogoul, A.: Using the Cassiopeia Method to Design a Soccer Robot Team. Applied Articial Intelligence (AAI) Journal, 12, 2-3 (1998), 127-147
14. 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
15. Chella, A., Cossentino, M., Pirrone, R.: Multi-Agent Distributed Architecture for a Museum Guide Robot. Proc. of the GLR worshop at the 2001 AI*IA conference. Bari, Italy, Sept. 2001
16. Chella, A., Cossentino, M., Lo Faso, U.: Applying UML use case diagrams to agents representation. Proc. of AI*IA 2000 Conference. Milan, Italy, Sept. 2000
17. Chella, A., Cossentino, M., Infantino, I., and Pirrone, R.: A vision agent in a distributed architecture for mobile robotics in Proc. Of Worskshop “Intelligenza Artificiale, Visione e Pattern Recognition” in the VII Conf. Of AI*IA. Bari, Italy, Sept. 2001
18. O’Brien, P., Nicol, R.: FIPA - Towards a Standard for Software Agents. BT Technology Journal 16,3(1998),51-59
19. Burrafato, P., Cossentino, M.: Designing a multi-agent solution for a bookstore with the PASSI methodology. Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002). May 2002, Toronto, Ontario, Canada at CAiSE'02
20. Kendall, E. A., Krishna, P. V. M., Pathak C. V., Suresh, C. B.: Patterns of intelligent and mobile agents. Proc. of the Second International Conference on Autonomous Agents. Minneapolis, May 1998, 92–99
21. Aridor, Y., and Lange, D. B.: Agent Design Patterns: Elements of Agent Application Design. Proc. of the Second International Conference on Autonomous Agents. Minneapolis, May 1998, 108–115
22. Agentcities.NET: http://www.agentcities.net
23. 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
24. Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. To appear in Information Systems, Elsevier, Amsterdam, The Netherlands, 2002
25. Odell, J., Van Dyke Parunak, H., Bauer, B.: Extending UML for Agents. AOIS Workshop at AAAI 2000. Austin, Texas, July 2000