Building Agents with Agents and Patterns

L. Sabatucci [(1)], M. Cossentino [(2)], S. Gaglio [(3)]
(1,3) DINFO - Dipartimento di Ingegneria Informatica, Università degli Studi di Palermo - Viale delle Scienze, 90128 Palermo, Italy
sabatucci@csai.unipa.it; gaglio@unipa.it
(2) Istituto di Calcolo delle Reti ad Alte Prestazioni, Consiglio Nazionale delle Ricerche
cossentino@pa.icar.cnr.it

***Abstract*—The use of design patterns proved successful in lowering the development time and number of errors when producing software with the object-oriented paradigm. Now the need for a reuse technique is occurring for the emergent agent paradigm, for which a great effort is currently spending in methodology definitions. In this work we present our experiences in the identification, description, production and use of agents patterns. A repository of patterns was enriched during these years so to request a classification criteria and a documentation template useful to help user during the selection.**

***Index Terms*—Multiagent systems, patterns, reuse models and tools.**

I. Introduction

IN the last years, multi-agent systems (MAS) achieved a remarkable success and diffusion in employment for distributed and complex applications. In our research we focus on the design process of agent societies, activity that involves a set of implications such as capturing the ontology of the domain, representing social aspects, and intelligent behaviours. In the following, we will pursuit a specific goal: lowering the time and costs of developing a MAS application. We think that a fundamental contribution could come by the definition of reuse techniques and tools providing a strong support during the design phase. We identified in design patterns a good solution to this need. Significant motivations to the use of design patterns in a project are:

- **Patterns communicate knowledge**: they allow experts to document, reason and discuss systematically about solutions applied to specific problems. Patterns also help people to learn a new design paradigm or architectural style, and help new developers ignore traps and pitfalls that have been learned only by costly experiences [8].
- **Patterns increment quality of software**: design patterns are signs of quality because their use implies safe and elegant solutions that are validated by the experience rather than from testing [15].
- **Patterns improve the documentation process**: the pattern catalogue constitutes a documentation repository where the designer may explore possible solutions for his/her problem: each pattern provides a comprehensible way of documenting complex software architectures by expressing the structure and the collaboration of participants at a level higher than source code [16].
- **Patterns decrease development time**: design patterns are strategies helping people to find their way through complex situations by applying ready solution to solve difficult problems. Also they help in diagnosing, revising, and improving a group's work [8][10].
- **Patterns improve software maintenance**: a project obtained with patterns reuse is robust and simpler to modify with respect to traditional projects [15].

Our definition of pattern come from traditional object-oriented design patterns, revised for the agent paradigm. In particular we use an ontological approach, strongly influenced by the study of multi-agent system (MAS) meta-models.

In this paper we will present AgentFactory II, a tool for working with patterns for agents, integrating a user interface to select and apply pattern from a repository. AgentFactory II is based on the experience done with a previous version [5] that was useful to explore the possibility of design a multi-agent system using design patterns as building block and successively to generate code from them. The major innovation of the tool is an expert system able to reason about the project and patterns, and a complex system to generate source code and documentation.

In the following we will discuss our tool. In the section II we discuss our pattern definition that is the base of our approach; in the section IV we illustrate the complex architecture adopted to realize the tool; in the section V we illustrate the *DocWeaver*, a specific agent of this architecture, that is responsible to generate the documentation in a specific agent-oriented style. Finally, in the section VI we report some conclusions.

II. The PASSI Design Process

In our work we will refer to the PASSI [3] methodology that represents the starting point and the natural context of our pattern definition and application. PASSI (Process for Agent Societies Specification and Implementation) drives the designer from the requirements analysis to the implementation phase for the construction of a multi-agent system. The design work is carried out through the construction of five models obtained by performing twelve sequential and iterative activities. Briefly, the phases and activities of PASSI are:

- **System Requirements**. It produces a description of the functionalities for the system-to-be, driving an initial decomposition of the problem according to the agent

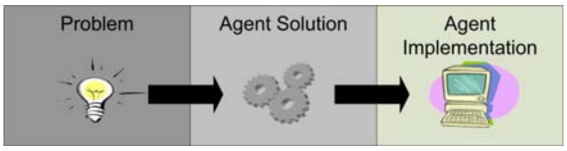


Fig. 1 – The three levels architecture for our pattern definition

paradigm. The four activities are: (i) the Domain Requirements Description, where the system is described in terms of the functionalities; (ii) the Agent Identification where agents are introduced for dealing with identified requirements; (iii) the Role Identification where agents' interactions are described by the introduction of roles; (iv) the Task Specification where the plan of each agent is draft.

- **Agent Society**. It is the phase where the agent paradigm is fully exploited. It is composed of four activities: i) in the Domain Ontology Description the system domain is represented in terms of concepts, predicates and actions; ii) the Communication Ontology Description focuses on the agents' communications, described in terms of referred ontological elements, content language and protocol; iii) in the Role Description the distinct roles played by agents are detailed within their dependencies.
- **Agent Implementation**. It is a model of the solution architecture in terms of required classes with their attributes and methods. It is composed of two main streams of activities (structure definition and behaviour description) both performed at the single-agent and multi-agent levels of abstraction.
- **Code**. It is a model of the solution at the code level. It is largely supported by patterns reuse and automatic code generation.
- **Deployment**. It is a model of the distribution of the parts of the system across hardware processing unit; it describes the allocation of agents in the units and any constraint on migration and mobility.
- **Testing.** It has been divided into two different activities: the Agent and the Society test. In the first one the behavior of each agent is verified with regards to the original requirements whereas during the Society Test, integration verification is carried out together with the validation of the overall results of the iteration.

III. Pattern for Agent Definition

In order to work with design pattern we require a formal definition. We agree with the traditional object-oriented definition for design patterns, but we introduced some changes in order to adapt it for the agent paradigm.

We define a pattern as “a problem which occurs over and over again in our environment, and then describes the core solution to that problem” [1]; the common use of design patterns is to describe best practices, good designs, and capture experience in such a way that it is possible for others to reuse them [8].

Our design patterns approach was conceived during the development of the PASSI process [3] with the goal of introducing a viable reuse technique for the development of MASs: our reuse technique uses some PASSI diagrams for describing the proposed solution. In this way the “solution” introduced is expressed in agent oriented terms, for instance agent, role, communication, goal and so on.

Table 1 – Description for the GenericAgent pattern

Name: GenericAgent **Classification**: internal architecture/single-agent **Intent**: this pattern may be used as the root before applying all single-agent patterns because it gives to an agent the ability of registering/deregistering to the platform services (AMS and DF). **Motivation**: this pattern is useful for agents who want to discover if the system offers a specific service and what agents can provide it. The GenericAgent pattern adds the ability of registration to the platform (white/yellow pages) so that the agent is accessible for conversations. **Preconditions**: none. **Postconditions**: the agent is able of registering and de-registering to AMS e DF. **Solution** (Structure, Participants and Collaboration): the target agent is enriched with an attribute for listing the description of all its services offered to the community. A *registerDF()* and *registerAMS()* methods with their correspondent *deregisterDF()* and *deregisterAMS()* are introduced to agent class. **Related Patterns**: this pattern may be the predecessor for all single-agent patterns. The LogAgent is a variant of this pattern which may be used specifically for debugging/testing aims.

Jackson in an analysis of the software design [8] distinguishes between the problem and the solution contexts: the problem and its solution are separated entities located in two different conceptual positions. The solution stays *in the computer and in its software* (machine domain) whereas the problem is *in the world outside from it* (application domain). Our approach to the definition of agent patterns spreads across both of the application and machine domains. However we need to specialize the Jackson’s domains to cope with the agent concept. When using agents as a design paradigm the solution is generally quite abstract respect when it is expressed in object oriented terms. We split the machine domain in two sub-domains, introducing the “agency domain” between the problem and implementation domains (see Fig. 1). Our pattern architecture is based on these three levels:

Pattern problem. A fundamental part of a pattern is the textual description of the problem for which it may be useful. It is composed by: (i) *motivation*, an explanation of how (and why) the pattern works, and why it is good, putting into evidence steps and rules required to resolve the problem; (ii) the application context describes the conditions under which the problem and the solution seem to recur, and for which the solution is desirable; (iii) *related patterns* element describes

Table 2 - Rules for the GenericAgent pattern

```
(deffunction generic_agent (?name)
  (if (generic_agent_precond ?name) then
      (add_new agent ?name)
      (add_new agent_action “register_DF” ?name)
      (add_new agent_action “unregister_DF” ?name)
      (add_new agent_action “register_AMS” ?name)
      (add_new agent_action “unregister_AMS” ?name)
  ))

  (deffunction generic_agent_precond (?name)
      (if (exist (agent ?name)) then
          (return FALSE)
       else
          (return TRUE)
  ))
)
```

other patterns that could solve a similar problem. As an instance of pattern we report the *GenericAgent* described in details in Table 1.

Pattern solution. It represents the solution (introduced when adopting the pattern) in terms of agent-oriented elements. The solution description illustrates the static structure and the dynamic behaviour introduced by the pattern in terms of structure, participants and collaborations. The formal description is a set of rules expressed using a logical language based on Jess. These rules are classified in three groups: i) the preconditions have to be verified before to introduce the pattern, ii) the postconditions are rules to verify after the pattern application (they may condition future patterns application), and iii) the solution rules that are a logical description of the elements constituting the solution and their behaviour/interactions. Our patterns for agents are explicitly defined to be used in conjunction with the PASSI methodology [3]; as a consequence the solution is described using some diagrams from the PASSI phases depicting agents' internal structure and social behaviour. Roles, tasks, communications, and interaction protocols are examples of the involved elements. An instance of rules for the pattern solution is shown in Table 2 for the previously introduced *GenericAgent*; in the subsection IV.B we will describe how these rules influence the design when the pattern is introduced in the project.

Pattern implementation. This represents the lower level of the solution containing the effective implementation in object oriented terms. It uses diagrams of PASSI depicting the static structure of the involved agents in terms of classes, attributes and methods using conventional UML class diagrams and dynamic behaviour of one or more agents involved in interactions using activity or state-chart diagrams.

The main feature of our tool is to automatically generate the solution at this implementation level. This feature will be discussed in the subsection IV.C.

IV. The Agent Factory Tool

The AgentFactory II was designed and developed after the experience obtained developing and using the previous version of the tool. The strategic choice distinguishing this new version of the tool from the previous one is that this was developed as a multi-agent system.

The system is basically composed by four agent organizations [8] (or groups of agents responsible of a functional area, see Fig. 2): i) the pattern architect, ii) the agent model, iii) the aspect weavers and iv) the object model. Each organization will be discussed in detail in the following. An agent, external to all the organizations, is the *UserAgent* responsible to interact with the designer, using a GUI (a screenshot is reported in Fig. 4); this agent has the goal to adapt its GUI to the agents present in the system (that are not known a priori); this is realized using communications, ontology abstraction and reflection technique.

A. The Agent Model Organization

This organization is conceived to realize the agent solution level of our architecture (reported in Fig. 2). This organization is designed to front an hard problem: to maintain the meta-model of our patterns independent from the specific methodology employed to design a system. This is a complex goal because all the agent oriented methodologies use own specific meta-model, involving different concepts or assigning them different meanings.

We structured the “Agent Model” as an holonic organization (shown in Fig. 3) based on three basic roles (that are played by the agents of the organization): i) the MMDF, the head of the hierarchy, ii) the Fragment Agents stay at the intermediate level, whereas iii) the Model Agents are the bodies of this holonic structure.

The most important role on the organization is played by the MMDF (MetaModel Directory Facilitator) agent that is inspired to the FIPA [6] Directory Facilitator (DF); in the abstract architecture defined by FIPA, the DF is the agent responsible to maintain the yellow pages for all the services in the system; communicating with the DF all the agents may register its own services or discovery services offered by other agents. The MMDF agent has a similar function but restricted to the building of the meta-model to use in the design: at the

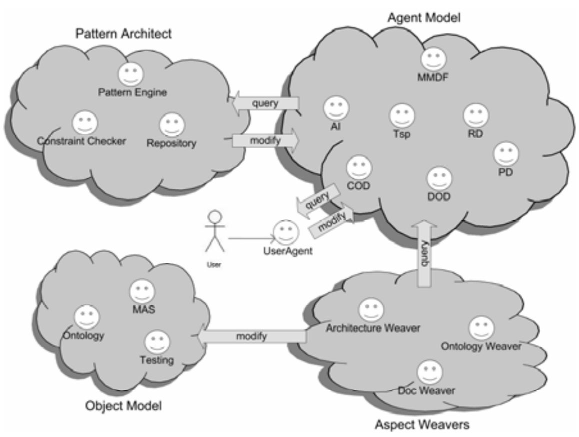


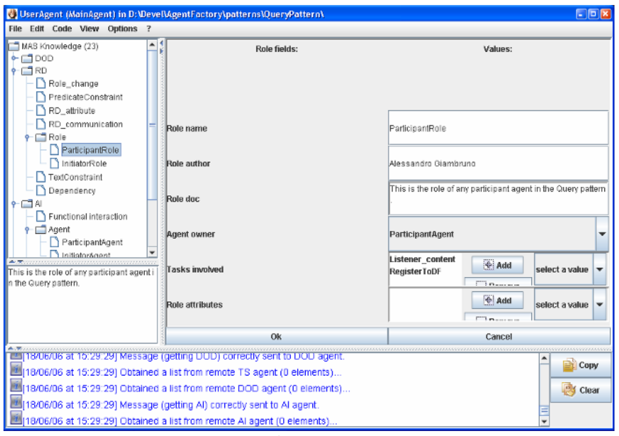
Fig. 2 – Organizations and agents involved in the AgentFactory II tool

beginning the meta-model is empty; when the model agents are executed they register one or more meta-model elements: therefore the MMDF is populated at run-time (according to a specific methodology).

The fragment agents represent "pieces of a methodology" and are responsible to group model agents coming from the same methodology in a unit; this is a double function: i) fragment agents coordinate the work among their model agents (internal collaboration); ii) fragment agents allow the collaboration of elements coming from different methodologies (external collaborations). For illustrating this concept, in Fig. 3 we have shown a possible configuration for the "Agent Model" organization. We have two fragments coming from two agent oriented methodologies: PASSI [3] and Tropos [3]. Each of these fragments is responsible of different elements of the meta-model (*requirement*, *role* and *agent* for PASSI, *goal*, *resource* and *agent* for Tropos); intersections among model agents may be treated in two different ways: a concept may be shared among different fragments (as the *agent* in Fig. 3) or may be exclusive of a methodology.

B. *The Pattern Architect Organization*

This is the organization responsible to manage the pattern repository and to introduce the rules (of selected patterns) into the system. Our pattern implementation is realized using a first order language; we have chosen to extend the Jess language [12] adding the ability to access to the services offered by the Agent Model (for instance to query for a specific element, or to introduce a new element). In Table 2 there is an example of a pattern: the *GenericAgent*; it is a pattern used for giving to an agent the ability of registering/deregistering to the platform services (white/yellow pages). This pattern is useful for agents who want to discover if the system offers a specific service and what agents can provide it. The pattern is done by a rule, *generic_agent*, that is activated using a parameter (the name of the new agent). This simple set of rules verifies (precondition) if an agent with the same name exists in the project, an then (pattern solution) adds the agent with some abilities (*register_DF*, *unregister_DF*, *register_AMS*,



This is the role of any participant agent in the Query pattern.

Role fields:	Values:
Role name	ParticipantRole
Role author	Alessandro Giambruno
Role doc	This is the role of any participant agent in the Query pattern .
Agent owner	ParticipantAgent
Tasks involved	Listener_content RegisterToDF
Role attributes	

```
[18/06/06 at 15:29:29] Message (getting DOD) correctly sent to DOD agent.
[18/06/06 at 15:29:29] Obtained a list from remote TS agent (0 elements)...
[18/06/06 at 15:29:29] Obtained a list from remote DOD agent (0 elements)...
[18/06/06 at 15:29:29] Message (getting AI) correctly sent to AI agent.
[18/06/06 at 15:29:29] Obtained a list from remote AI agent (0 elements)...
```

Fig. 4 - A screenshot of the AgentFactory User agent

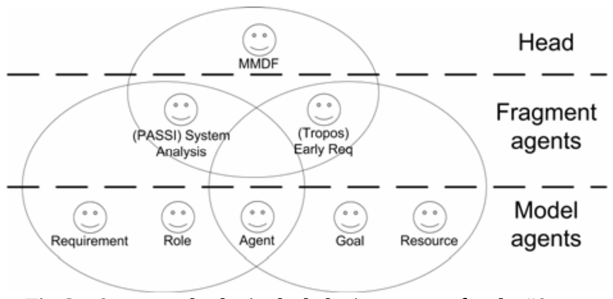


Fig. 3 – Agents and roles in the holonic structure for the "Agent Model" organization

unregister_AMS). In this example there are no postconditions.

C. *The Aspect Weavers Organization*

A significant characteristic of AgentFactory (already present in the early version of the tool) is the automatic code generation for multi platforms (until now we supported only Jade [2] and FIPA-OS [7], but it was conceived for being extendible with other agent-platforms that are compliant with the abstract FIPA architecture [6]). The previous version of the tool had a code generation engine based on a sequence of transformations according to the MDA architecture [13]. In this new version we are realizing a more complex transformation, that is inspired to the Aspect Oriented Programming (AOP) [13] to reduce the gap between the agent solution (introduced using the patterns from the repository and refined by the designer) and the object-oriented solution (that typically is an object oriented system). We inspired to a development team where different human-roles (that are expert in their own sector) individually work in a specific competence area, giving their personal contribution to the final solution. In our context agents are the experts and each area of competence is an aspect of the agent-oriented solution to take in consideration. Agents have to collaborate in order to converge all their contribution in the same final object-oriented code. In the AOP terminology the engine that realize this convergence is the aspect weaver; this is the motivation for the name of the organization: an aspect weaver is the "expert" of a specific sector of the project; it is responsible to a specific aspect of the project and to generate the output in terms of object-oriented solution. The entire organization is organized to weave all the contributions coming from different aspects and to meet them in an unique solution.

We actually realized only three weaver agents: i) an *ArchitectureWeaver* (responsible of the agent skeleton and communications), ii) an *OntologyWeaver* (responsible to add an ontology reference to the message exchanged by agents) and iii) a *DocWeaver* (that creates the documentation; it will be discussed in details in the section V). The *ArchitectureWeaver* fundamentally carries out the code generation functionality of the previous version of AgentFactory, generating the base architecture of the agents within their abilities/tasks. The *OntologyWeaver* adds the management of the ontology: concept, predicate and actions that are used in the agent knowledge and communications.

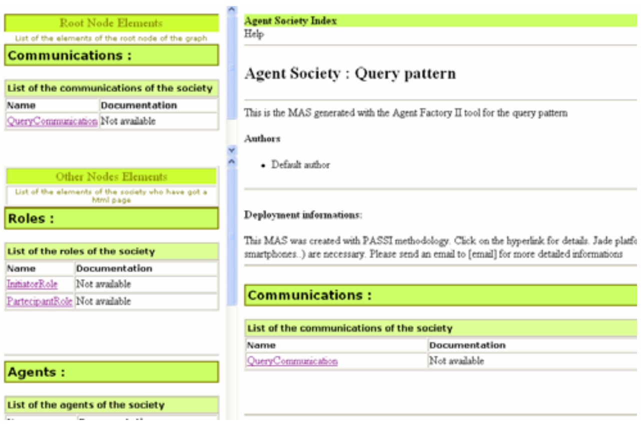
D. The Object Model Organization

This organization is conceived for realizing the agent implementation level of our architecture (see Fig. 2); it is relative to the object oriented solution. Agents of this organization are responsible to treat elements of the object oriented paradigm (such as classes, methods, attributes and so on). The organization is composed by three agents: i) the *MAS* agent, ii) the *Ontology* agent and iii) the *Testing* agent. The *MAS* agent is responsible to handle data for a multi-agent system taking in consideration both the static structure of the agent and the behaviour of the multi-agent system. It is able to export the source code for Jade and FIPA-OS agent platforms. The *Ontology* agent is responsible to generate classes for the system ontology (composed by concepts, predicates and actions). These classes are serializable and are used in the agents' knowledge and communications. The *Test* agent (still under development) is responsible to generate stub and driver agents for simulate the communications and collaborations among system agents (integrations testing).

V. A Weaver Agent: DocWeaver

During the development of multi-agent system we suffered the lack of a specific technique for documenting our source code; we used the Javadoc for generating the API documentation (from comments in source code), but we noted it was difficult to navigate because it implies a shift in the paradigm (from agent-oriented to object-oriented and vice versa); whereas the solution is expressed in agent oriented terms, the documentation is expressed in object oriented terms: the mapping is not direct and easy. Therefore we demanded a way for documenting our solution using directly an agent oriented style.

We propose the AgentDoc, an agent oriented style for documenting a multi-agent system; terms included in the documentation are not fixed, but are depending from the specific methodology used and therefore from the specific MAS meta model. AgentFactory II has simplified this because is naturally inclined to use different meta-models, so we have create a *DocWeaver* agent responsible to generate the AgentDoc for the designed MAS. In order to generate this documentation the AgentDoc uses the meta-model in the MMDF. This is not enough because the agent requires information about how an element of the MAS meta-model influences the documentation content. In order to solve this problem the *DocWeaver* uses a (manually built) configuration graph that specifies what elements (graph nodes) have to be included in the documentation (for each instance of the included element an HTML page is generated); whereas the relationships among the elements (graph arcs) generate links among pages. The result is a navigable hyper textual documentation; in the Fig. 5 an example of this documentation is shown generated using a meta-model of the PASSI methodology.


Root Node Elements

List of the elements of the root node of the graph

Communications :

List of the communications of the society	
Name	Documentation
QueryCommunication	Not available

Other Nodes Elements

List of the elements of the society who have got a html page

Roles :

List of the roles of the society	
Name	Documentation
InitiatorRole	Not available
PartecipantRole	Not available

Agents :

List of the agents of the society

Agent Society Index

Help

Agent Society : Query pattern

This is the MAS generated with the Agent Factory II tool for the query pattern

Authors

- Default author

Deployment informations:

This MAS was created with PASSI methodology. Click on the hyperlink for details. Jade platfo smartphones..) are necessary. Please send an email to [email] for more detailed informations

Communications :

List of the communications of the society	
Name	Documentation
QueryCommunication	Not available

Fig. 5 - A screenshot of the documentation generated for the case study

VI. Conclusions And Future Works

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 artificial intelligence and robotics, that it is possible to obtain great results with a correct approach.

In order to support the design of multi-agent system we developed a complex multi-agent system for building agents with a pattern support. This tool is also able to generate the documentation and the source code for the project. Actually the code generated is just a bit richer that code generated in the previous version, however we are working on a more complex organization with a greater number of weaver agents involving other aspects as role, task, plan and so on; in this context we require a more precise coordination mechanism among the weavers. Another improvement under development is the *Tesing* agent, that would be employed for integration testing on multi-agent system.

References

[1] Alexander C. 1979. The Timeless Way of Building. Oxford University Press

[2] Bellifemine F., Poggi A. and Rimassa G. 2001. Developing Multi-agent Systems with JADE. In proceedings of The 7th international Workshop on intelligent Agents. Agent theories Architectures and Languages (July 07 - 09, 2000), LNCS 1986, Springer-Verlag, London, pp. 89-103.

[3] Bresciani P., Giorgini P., Giunchiglia F., Mylopoulos J., and Perini A. 2004. TROPOS: An Agent-Oriented Software Development Methodology, Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers 8(3), pp. 203-236

[4] Cossentino M. 2005. From Requirements to Code with the PASSI Methodology, In Agent-Oriented Methodologies, edited by B. Henderson-Sellers and P. Giorgini, Idea Group Inc., Hershey, PA, USA

[5] Cossentino M., Sabatucci L. and Chella A. 2003. A Possible Approach to the Development of Robotic Multi-Agent Systems. IEEE/WIC Conf. on Intelligent Agent Technology (IAT'03). Halifax (Canada), October, 13-17, pp 539- 44.

[6] FIPA Abstract Architecture – [Available on Internet] http://www.fipa.org/repository/architecturespecs.html

[7] FIPA-OS Website - [Available on Internet], http://fipaos.sourceforge.net

[8] Ferber J., Gutknecht O. 1998. A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. In Third International Conference on Multi Agent Systems (ICMAS'98); p 128.

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

[10] Greenfield J. and Short K. 2003. Software factories: assembling applications with patterns, models, frameworks and tools. In Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM Press, New York, NY, pp. 16-27

[11] Jackson M. 2001. Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley

[12] Jess Rule Engine – available at: http://herzberg.ca.sandia.gov/jess/

[13] Kiczales, G. 1996. Aspect-oriented programming. ACM Comput. Surv. 28, 4es (Dec. 1996)

[14] OMG Model Driven Architecture - [Available on Internet] http://www.omg.org/mda/

[15] Prechelt L., Unger B., Philippsen M. and Tichy W. 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.* 28(6), pp. 595-606.

[16] Schmidt D. and Stephenson P. 1995. Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms, In proceedings of the *9th European Conference on Object-Oriented Programming*, LNCS 952, pp. 399 – 423