Pitfalls of agent system development on the basis of a Travel Support System

Maciej Gawinecki[1], Mateusz Kruszyk[2,3],
Marcin Paprzycki[1], and Maria Ganzha[1]

[1] Polish Academy of Sciences, Systems Research Institute,
Warsaw, Poland
{Maciej.Gawinecki,Marcin.Paprzycki,Maria.Ganzha}@ibspan.waw.pl,
[2] Adam Mickiewicz University, Department of Mathematics and Computer Sciences,
Poznań, Poland
[3] *Content Forces*, Content Management Services provider,
Amsterdam, Netherlands and Poznań, Poland
www.contentforces.pl

Abstract. Belief that a particular software engineering paradigm is universal for all domains is an illusion and agent-oriented engineering is not an exception. This we have learned during the development of an agent-based Travel Support System. The system was developed as a distributed environment to provide user with personalized content helping in travel planning. In this article we focus on these issues of our systems, where agents fit and give practical alternatives, where they do not. We believe that lessons learned in our project generalize to other project involving utilization of agent technology.

multi-agent system, development methodology, content management, personalization

1 Motivation

Nowadays a software architect, challenged to develop an application solving certain problems does not have to start building it from the scratch. Being a supporter of *re-use-what-available* philosophy, she can rather select relevant software development paradigm and existing off-the-shelf technologies. Obviously, each existing paradigm provides different abstraction for conceptualizing a given problem. The role of an architect is to know limitations and possibilities of different abstractions and choose the most intuitive and efficient one(s).[4] Therefore, believing the a particular paradigm is universal for all domains is an illusion and agent-oriented engineering is not an exception [1]. The same way as in all other

[4] In business practice the choice of right approaches is of course much more complicated and depends not only on software requirements, but also on various costs of using specific technology, skills of available programmers, predicted long-term support for existing technologies, etc.

software engineering tasks, a number of factors must be considered when selecting an agent-based approach [2, 3]. Let us list some of the more prominent ones.

An environment that is distributed, highly dynamic, uncertain and complex. Such an environment requires distribution of data, control or expertise and these objectives can be naturally supported by agents. For example consider production system in a factory; where points of control behave in both autonomous and cooperative way, and can adapt to local environment changes in order to realize a global goal [4]. Moreover, in a distributed environment access to remote resources can be improved by providing a light agent with mobility: the agent representing certain point of control can move to the target location where data necessary for computation is stored—instead of transferring large quantities of data over the network (as is the case in a traditional remote procedure call).

Agents as a natural metaphor. Organizations and societies consisting of cooperative or competitive entities can be naturally modeled by agent teams. Agent-oriented engineering allows classical methods of building complex systems (*decomposition*, *abstraction* and *organization*, as in object-oriented paradigm [5]) to be applied in distributed dynamical environments [6].

Dealing with legacy systems. Genesereth and Ketchpel suggests using agents as wrappers for legacy software, which in such a way can be reused by other components in heterogeneous system [7].

At the same time it is important to acknowledge that agent paradigm is relatively novel and may fail in cases in which traditional approaches (client-server architecture, object-oriented paradigm etc.) and technologies (Web Services, Java RMI, Content Managements Systems etc.) have taken their deserved place, confirmed by business practice. This is also the lesson that we have learned, building our Travel Support System and thus we would like to share our experiences in this article. This knowledge may also be helpful, in the case when someone may naively may claim that agents are a "silver bullet" for software development, while these arguments is still largely untested in practice [8].

In the next section we briefly summarize the main design characteristics of our agent-based Travel Support System. We follow with a description of major problems that we have run into. We complete the paper by description of proposed solutions to these problems.

2 Background

Travel Support System (TSS) is an academic project aiming, among others, at convincing agent-idea skeptics that building an agent-based system for planning a travel is nowadays both reasonable and possible with use of on-the-shelf technologies [9]. Our work was inspired by the following scenario. *Hungry foreign tourist arrives to an unknown city and seeks a nice restaurant serving cuisine that she likes. Internet, contacted for advice about restaurants in the neighborhood, recommends mainly establishments serving steaks, not knowing that the tourist is a fanatic vegetarian.* This scenario determines the following functionalities of the system:

- *Content delivery.* Content should be delivered to the user in browser-processable form, i.e. HTML, WML etc. and match the user query.
- *Content personalization.* Delivered content should be personalized according to the user-model to avoid situations like the one presented in the scenario.
- *Adaptation of personalization.* Habits of the user can change, therefore her model should be adapted on the basis of her activities recorded by the system.

In fact, these functionalities are realized only by a part of the Travel Support System, called *Content Delivery Subsystem*. In what follows we focus our attention only on this particular subsystem (hereafter called system). The remaining parts of the TSS, responsible for data management and collection have been depicted on figure 1 and described in detail in [9]. This latter reference (and references collected there to our earlier work) should be consulted for all remaining details concerning the TSS. As far as the technologies utilized in the TSS, the RDF

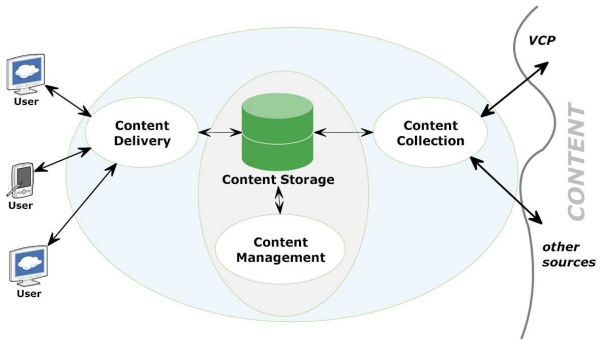

Fig. 1. Travel Support System general architecture.

language has been applied to demarcate data (to allow machines process semantically rich data and meet requirements of Semantic Web applications [10, 11]). Jena framework has been used to manage RDF graphs [12] (RDF graphs are persisted as Jena models in traditional relational databases). When conceptualizing the system, the Model-View-Controller design pattern [13] has been applied for clear separation between pure data (model) and its visual representation (view).

Let us now list the most important agents that have been designed and implemented in our system

- *Proxy Agent (PrA)* integrates non-agent user environment with the agent-based system (precisely described in [14]). It is able to receive HTTP requests from a user browser (since it wraps a simple "home-made" HTTP server), and forward them to the system and return an answer from the system in the form of an HTTP response.
- *Session Handling Agent (SHA)* is responsible for realizing user requests. It plays the role of controller in the MVC pattern. Specifically, it (1) receives

user request from the *PrA*, (2) creates model responding the request or delegates the *PA* to do it, (3) requests the *VTA* to transform the model into the browser-readable view, and (4) passes the response to the *PrA*. Additional responsibility of the *SHA* is to track user feedback and log it in the *History* database.

- *Profile Managing Agent (PMA)* is responsible for initializing and learning user profile on the basis of user feedback (see [15], for more details about learning algorithm used in the system). It provides a user profile to the *PA*.
- *View Transforming Agent (VTA)* is a response to the need of providing content to various user devices, which can render documents described in markup language (e.g. HTML) as well as simple TXT messaging. The *VTA* generates a view in terms of a HTML/WML/TXT document matching a given model. It wraps and utilizes Python-based Raccoon server, which applies pointed XSL stylesheet to a given XML document [16].
- *Personal Agent (PA)* acts on behalf of its user, personalizing recommendations restaurants with respect to the user profile (see [15], for more details about filtering algorithm used in the system). It is created only for a duration of the session, for a user who is logged in. Notice that user can log-in and log-out during a session and while user is logged out the *PA* can orchestrate work that is preparing a response set. When the response set is delivered and user logs-out, the *PA* is "killed."
- *Restaurant Service Agent (RSA)* Wraps Jena model with data of Polish restaurants.

3 Problems encountered during the development of the system

In this section we present four problems, we met during agentifing our system. We tried to present them as general issues with aids, so other developers could utilize our experience in their work.

3.1 Agents as wrappers for legacy software

Utilization of agent as wrappers for legacy software, was proposed as aid for systems with heterogeneous software [7]. This is representation of a general move toward message-oriented communication, which increases the interoperability, portability, and flexibility of a distributed applications [17]. However, we found such utilization of agents justified only in one of the following situation: (a) where there is no other middleware solution, which would connect heterogeneous parts of an application, or (b) usage of agents brings additional functionality to wrapped software (as e.g. as Observer or Adaptor design pattern).

Let us illustrate this situation by an example from our system. JADE agents use two semantically rich languages: SL and ACL [18]. For example, in the following message the *RSA* agent informs the *PA* about requested restaurants:

```
(request
  :sender ...
  :receiver ...
  :ontology tss-ontology
  :language fipa-sl
  :content''
        (result
           (action (find-restaurants :query .... ) )
           (''<?xml version=''1.0''?>
       <rdf:RDF>
          <res:Restaurant rdf:ID=''Poland_LD_Lodz_
              _Kuchnia_Polska_Obiady_domowe996614020''>
           <loc:streetAddress> ul. Lutomierska 8
           </loc:streetAddress>
         </res:Restaurant>
         ...
       </rdf:RDF>")
    )
  ''
)
```

It can be seen that the message contains also RDF/XML serialized data describing restaurants. The process of creating an ACL message by the *RSA* and reading it by the *PA* has been depicted in Figure 2. RDF data describing restaurants and persisted in the Jena model must be serialized to RDF/XML. The rest of the message content is constructed with use of Java beans representing certain concepts in communication ontology and then encoded in Lisp-like strings. All resulting message-parts are combined into a single ACL message and communicated to the *PA* with uses the RMI technology (standard technology that JADE uses to transfer messages). The *PA* reads the content of the message in exactly the reverse way. This process is definitely time- and resource-consuming and it is

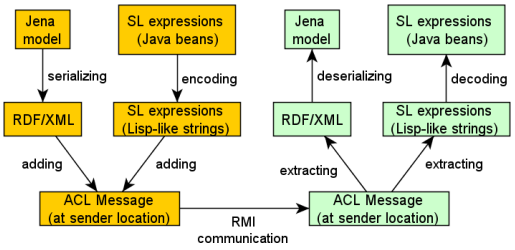

Fig. 2. Communication costs in Travel Support System.

not justified in our system, where agents do not take advantage of SL's features. The SL language was developed to provide agents with ability of communicating their beliefs, desires and uncertain beliefs; this takes place particularly in the case of, so called, BDI agents. However, our agents are not BDI agents and do not

utilize semantically rich communication. Moreover, as it was described above, remote Jena models persisted in the database can be reached with use of simple database connection, without time-consuming serialization of Jena models and putting them inside of ACL messages. In this context, one should also remember about additional effort of a developer, who needs to design communication ontology (in the SL language).

An alternative could be (1) to use of simple database connections, in cases where data sources where interfaced by agent wrappers, (2) introducing traditional technologies, such as Java RMI, for requesting remote services (such as view transformation), or (3) if possible, integrating interacting agents within a single host. Summarizing, interoperability among parts of an application is simply warranted by Java-based interfaces of an application.

3.2 Replacing traditional technologies with agents

You see agents everywhere. Many young developers narrowly follows the vision of Nwana and Ndumu to *agentify* all software functionalities [19]. However, this is very common mistake to design whole system in an agent-oriented architecture, while most of the work can be done with use of traditional approaches and technologies [1]. We have made this mistake for a purpose. The main **objective** of the original design was to utilize agents in all possible functions. Let us now look into some more details as to what we have found.

The main scenario of the system is content delivery, which is realized in a client-server architecture, where the system plays a passive role of server. This client-server architecture has been naturally transformed into the *FIPA Request Interaction Protocol* [18], in which the *Initiator* plays the client role, and the *Responder* plays the server role. Specifically, the *SHA* (Initiator) requests that the *VTA* (Responder) generates a view from the model. Separate functionality has been created as the *PrA*, which wraps the HTTP server. All these agents plays vital roles in our adaptation of the Model-View-Controller pattern in agent-like environment.

Summarizing, in general the MVC pattern utilizing the HTTP protocol can be characterized as:

- *stateless*—each user request is independent to others, so the results of response to a user request have no influence on results of another one, by an analogy to the HTTP;[5]
- *reactive*—MVC components stay inactive between user requests, so they react only to external requests, simply like *active objects* [20];
- *synchronous*— as process of realizing a single user request is a sequence of steps, where each next step cannot be realized until the previous one has been finished: receiving HTTP request, preparing model, preparing view and returning HTTP response;

[5] With an exception to a term session, which—however—has been successfully handled by traditional CMS frameworks

– *parallel*, but not *concurrent*—parallelism is utilized to decrease interleaving in I/O operations.

Therefore, in this case, the well known properties of agents defined as *proactiveness*,[6] *asynchronous* communication, *statefulness* and *concurrency*[7] cannot be utilized.

Previously, the MVC pattern has been successfully incorporated and tested in business practice by use of traditional technologies, such as the Spring Framework [21]. In our case utilization of agents for this pattern resulted in the following disadvantages of the system:

– difficulty of integration of the proposed solution with traditional Content Management Systems, due to use of niche technologies (Raccoon, agents); this seems more reasonable in the situation where content presented to a user is composed also from fragments not delivered by agents.
– forcing a potential developer to learn designing web content from scratch,
– introduction of the solution that can be less stable and efficient that thoroughly tested solutions, for example (1) not properly managed concurrency in the HTTP server can be a bottleneck in the system and (2) above mentioned communication overhead can slow down data flow in the system.

Overall, upon reflection we can say that we have modeled a part of the system with higher abstraction than naturally necessary, which resulted in difficulties of verifying and reasoning about such solution (i.e. the simpler the model the easier it is to think about it, to verify its correctness and to remove errors).

3.3 Solving conflicts in functionalities acquaintance

Creating too many agents, each one realizing separate functionality, is yet another pitfall in agent-oriented development [1]. The main problem is a potential communication overload caused by exchange of messages between separated functionalities [22]. Guidelines for solving such problems can be found in the Prometheus methodology, which proposes to analyze, in the given order, the following factors, while specifying particular agents: (1) data and knowledge acquaintance, (2) relationships among agents and (3) interactions frequency. Functionalities which use the same data or interacts with each other often are suggested to be integrated into a single agent [23]. However this approach often conflicts with a situation, where related resources, data etc. must be located on separate hosts (e.g. due to performance issue or because they belong to different owners). However, in this case communication can be limited by introducing mobile agents. Let us see this approach on the example.

At first, we applied the Prometheus methodology to the Travel Support System; the results can be seen in Figure 3. Red lines demarcate access of agents to

[6] Agents can react also changes of their internal state (being usually results of reasoning), what can be perceived as proactive behaviour

[7] programming technique, in which two or more, often cooperative, threads are making progress, often used to model real world entities

particular databases and ontologies (directions of arrows point read and/or write access). Black lines describe dependency of a pointed agent on a given functionality. For instance, *PMA* reads data from the *Stereotypes* database, and both writes to and reads data from *Profiles* and *Statistics* databases. Moreover, the learning process requires access to (1) *History* database (and thus communicating with the *SHA* to obtain the learning data) and *Restaurants* database (wrapped by the *RSA*). Let us consider this example further. *Content-based learning* adapted

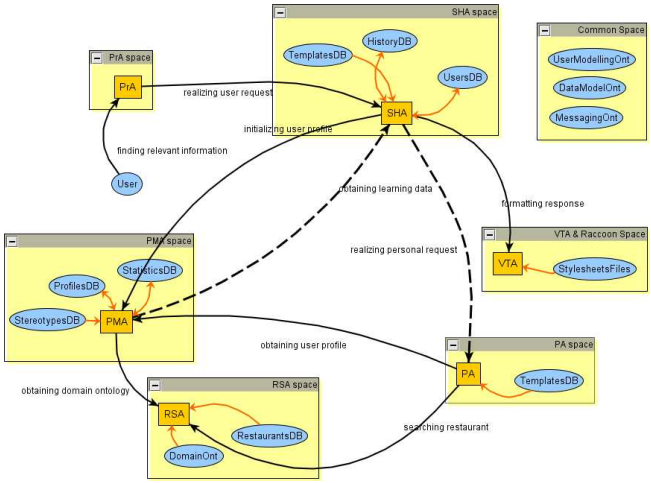

Fig. 3. Dependencies in the Travel Support System.

in the system requires great amount of information: both description of recommended objects (*Restaurants*) and history of user feedback about those objects (*History*). According to the integration rule we should have integrated all related functionalities of the *PMA*, the *SHA* and the *RSA* agents together with access to the necessary data sources in a single agent. But we have made an exception to this rule for the following reasons: (1) restaurant data could be provided by the company not belonging to the system and thus external to it, (2) the *PMA* and the *SHA* should be hosted on separate machines, because learning process requires large amount of CPU resources, while the *SHA* is obligated to respond to many users' requests in a timely fashion. An alternative would be to designate an agent that learns single user profile and moves to the host where appropriate data for a particular learning phase is stored. Mobility of a such an agent has two advantages: (1) *performance boost*—it decreases communication overload while accessing remote data, and (2) *design metaphor*—it provides the developer with possibility to realize certain computations from a single point of control which can move itself, while releasing her from necessity of passing control along various remote hosts (as in standard *remote procedure calling*). Example of such a solution can be seen in Figure 4, where Mateusz's *PA* travels across remote hosts.

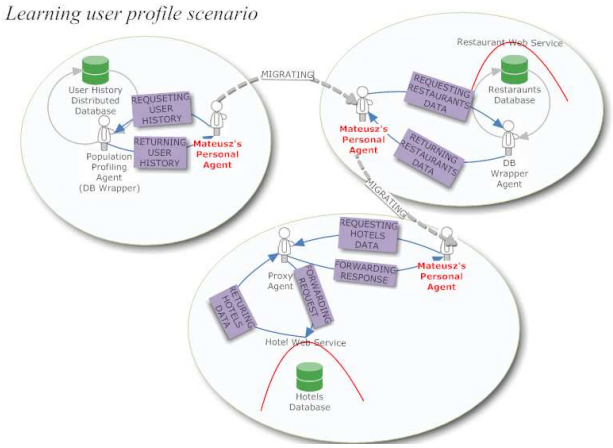

Fig. 4. Personal Agent migrating during learning process.

3.4 Placing Personal Agent in real environment

Generally speaking, automated personal assistants are one of the ways that agents are viewed [24]. This perspective says, that each user is represented in a system by a personal agent. We have learned that this approach can be justified mostly in situations, where such an agent resides on a user machine. Let us see reasons for this conclusion.

In our system we decided to utilize personal agent that is responsible for filtering and personalizing data delivered to the user [25]. In its original design, such an agent (also called *intelligent interface agent*) was supposed to exist on a user machine or mobile phone. The main reasons for this proposed design were:

- *security*—user profile is not explicitly accessible to the system,
- *mobility*— an agent can accompany the user in her travel, moving together with her mobile device,
- *resource separation*— an agent utilizes resources belonging to its user, not to the system.

However, in our system we assumed, that user devices are light (allowing only for visualizing documents demarcated in a markup language) and that user is unable or prevented from installing additional applications (e.g. for security reasons in a corporate environment). This decision has automatically swept away the expected advantages. Currently it is the system that stores user profiles. Moreover, a *PA* can be accessed by its user only through the *PrA* and this in an obvious ways limits the sense of *PA*'s mobility. And the last, but not least consequence is the necessity of providing additional resources to the *PA*, which now utilizes system hardware. To avoid wasting resources we decided to instantiate a *Personal Agent* only for duration of its user session. Therefore, we could have—for example—1000 users registered in the system, but only 100 of them actively

interacting with the system, and thus only 100 *PA*s would exists. However, this leads to an interesting open question related to collaborative filtering. It is typically assumed that in this case all trusted *PA*s should be accessible every time another *PA* is about to ask them for their opinion about (restaurant) recommendations (to provide maximally relevant response to its user). Since not all *PA*s are "alive" all the time, they cannot provide a response to such a query. Therefore, usage of collaborative filtering, or even usage of *PA*s must be re-considered.

4 How the TSS should be re-designed

Having analyzed problems that have arisen during the design and implementation of the original system, we have found that utilization of traditional technologies for some of its parts would be beneficial for the overall architecture. Therefore let us briefly discuss which functionalities of the system should be realized as agents and which—by utilizing traditional technologies, and how they can cooperate with each other.

4.1 Utilization of agents

Agents can be placed in the following system scenarios:

- *profile learning*—because of their ability to move across different locations; this approach (1) gives also ability to organize agents in teams of agents, each team realizing different algorithm of learning, and (2) introduces possibility of selecting agents to realize a particular task by use of negotiations, depending on agent's current location and current access to the resource;
- *content filtering*—creation of a single *Personal Agent* for each user should replaced by utilization of agents representing groups of users and using different filtering algorithms;

Traditional technologies should be utilized in the following cases:

- MVC-based framework, for example Spring, can be used for content delivery, replacing functionality of the *SHA* and the *VTA* (together with the need of utilization of the Raccoon server),
- *PrA* wrapping home-made HTTP server can be replaced by a traditional servlet container, such as an Apache Tomcat, on which the Spring will be hosted,
- access to remote data sources can be "unwrapped" and provided directly via a simple database connection; concurrent access issue must also be considered here (e.g. by use of *multiply-read-single-write* policy).

4.2 Integration of agents and traditional technologies

It has been shown that in the TSS there exists a demand for co-existence of both *traditional* and *agent* technologies. Therefore appropriate *middleware* must be

created allowing heterogeneous parts of the system to communicate with each other.

Servlets executed by the CMS need to issue commands to JADE agents, for example through the so called *gateway*, which can be featured by the `Gateway Agent` class from `jade.wrapper.gateway` package.

Agent requiring functionality of a certain Web Service (speaking the WSDL language) can send its request to the *Gateway Agent* (from Web Services Integration Gateway add-on [26]), which translates ACL messages into SOAP messages in both directions and forwards them between the Web Service and the requesting agent. See figure 4 for example of such a situation, where the *PA* requires data about hotels provided by an appropriate Web Service.

5 Conclusion

In this paper we have discussed lessons learned from the design and initial implementation of the agent-based Travel Support System. We have shown, that the initial assumption—everything should be an agent has only educational value. However, in a realistic system both agent and traditional technologies have to coexist and be utilized in a judicious way. Following the critical analysis of pitfalls of our design, we have outlined a solution that would make our system more realistic, flexible and efficient. Finally, specific middleware solution was proposed for integration of non-JADE parts with JADE-based parts of the system.

6 Acknowledgements

The authors would like to thank for fruitful discussions: Minor Gordon from Computer Laboratory of University of Cambridge in United Kingdom, Pawel Kobzdej from the Systems Research Institute of the Polish Academy of Sciences and Pawel Kaczmarek from Hewlett-Packard Poland. Many thanks also to Juan A. Bota Blaya from Information and Telecommunicatinos Engineering Department of the Murcia University in Spain for suggestion about the semantic overload issue.

References

1. Wooldridge, M., Jennings, N.R.: Pitfalls of agent-oriented development. In Sycara, K.P., Wooldridge, M., eds.: Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98), New York, ACM Press (1998) 385–391
2. Bond, A.H., Gasser, L., eds.: Readings in Distributed Artificial Intelligence. Morgan Kaufmann, San Mateo, CA (1988)
3. Jennings, N.R., Wooldridge, M.: Applications of intelligent agents. In: Agent Technology: Foundations, Applications and Markets. Springer, Berlin (1998)
4. Bussmann, S., Schild, K.: An Agent-based Approach to the Control of Flexible Production Systems. In: Procedding of the 8th IEEE International Conference of Emergent Technologies and Factory Automation (EFTA 2001), Abtibes Juan-les-pins, France (2001) 481–488

5. Booch, G.: Object-oriented analysis and design with applications (2nd ed.). Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA (1994)
6. Jennings, N.R.: Agent-oriented software engineering. In: Proceedings of the 12th international conference on Industrial and engineering applications of artificial intelligence and expert systems : multiple approaches to intelligent systems, Secaucus, NJ, USA, Springer-Verlag New York, Inc. (1999) 4–10
7. Genesereth, M.R., Ketchpel, S.: Software agents. Communications of the ACM **37** (1994) 48–53
8. Wooldridge, M.: An introduction to multiagent systems. John Wiley & Sons (2002)
9. Ganzha, M., Gawinecki, M., Paprzycki, M., Gąsiorowski, R., Pisarek, S., Hyska, W.: Utilizing Semantic Web and Software Agents in a Travel Support System. In: Semantic Web Technologies and eBusiness: Virtual Organization and Business Process Automation. Idea Publishing Group (2006)
10. Resource Description Framework (RDF). `http://www.w3.org/RDF/` (2005)
11. Semantic Web Activity Statement. `http://www.w3.org/2001/sw/Activity` (2001)
12. Jena a semantic web framework for java. `http://jena.sourceforge.net/` (2005)
13. Ramachandran, V.: Design Patterns for Building Flexible and Maintainable J2EE Applications. `http://java.sun.com/developer/technicalArticles/J2EE/despat/` (2002)
14. Gawinecki, M., Gordon, M., Kaczmarek, P., Paprzycki, M.: The Problem of Agent-Client Communication on the Internet. Parallel and Distributed Computing Practices **6** (2003) 111–123
15. Gawinecki, M.: User modelling on a base of interaction with WWW system. Master's thesis, Deparment of Mathematics and Computer Science, Adam Mickiewicz University, Poznan (2005)
16. Raccoon. `http://rx4rdf.liminalzone.org/Racoon` (2005)
17. Rao, B.R.: Making the most of middleware. **12** (1995) 89–06
18. Foundation for intelligent physical agents. `http://www.fipa.org` (2007)
19. Nwana, H.S., Ndumu, D.T.: A perspective on software agents research. The Knowledge Engineering Review **14** (1999) 1–18
20. Guessoum, Z., Briot, J.P.P.: From active objects to autonomous agents. IEEE Concurrency **7** (1999) 68–76 `citeseer.ist.psu.edu/guessoum99from.html`.
21. Spring Application Framework. `http://www.springframework.org` (2006)
22. Tusiewicz, M.: System wieloagentowy: teoria, projekt, implementacja oraz przyklady zastosowań. Master's thesis, Department of Mathematics, Physics and Computer Science, Jaggielonian University, Kraków (2003)
23. Padgham, L., Winikoff, M.: Prometheus: a methodology for developing intelligent agents. In: AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, New York, NY, USA, ACM Press (2002) 37–38
24. Laufmann, S.C.: Agent software for near-term success in distributed applications. (1998) 49–69
25. Nesbitt, S.: Collaborative Filtering on the Web: An agent-based Approach (Literature Review) (1997)
26. JADE Board, Whitestein Technologies AG: JADE Web Services Integration Gateway Guide. `http://jade.tilab.com` (2006)