

# On Agentware: Ruminations on Why We Should Use Agents

Federico Bergenti

AOT Lab  
Parco Area delle Scienze 181/A, 43100 Parma, Italy  
bergenti@ce.unipr.it  
<http://aot.ce.unipr.it/>  
FRAMETech  
Via San Leonardo 1, 43100 Parma, Italy  
bergenti@frame-tech.it  
<http://www.frame-tech.it>

**Abstract.** Agent-oriented software engineering has not yet solved the basic problem of *why* we should use agents to build our software system. Why is it convenient to use agents instead of more mature technologies like, for example, software components? This paper addresses this issue and compares a BDI-like agent model with well-known component models like Enterprise JavaBeans, CORBABeans and .NET components. The two main results of such a comparison are: (i) agents are more reusable and more composable than components, and (ii) agents allow to describe systems at a higher level of abstractions than components. This work is not meant to be conclusive; rather it intends to start a debate on these and related topics.

## 1 Introduction

The ultimate objective of agent-oriented software engineering (AOSE) is to provide all necessary tools to engineer agent-based systems. Such tools include, but are not restricted to, methodologies and development frameworks. Nowadays, AOSE has partially accomplished its task: we have some good methodologies and some good development frameworks also. Yet, all this work may not be sufficiently rewarded if the AOSE community would continue avoiding the very basic question of *why* we should use agents to build software systems. Why shall we employ agent-based technologies instead of choosing any other technology? Aren't available technologies sufficient? These, and many other similar questions that may come to your mind, are a pain for us interested in agents. Put simply, we have no *real* answer to them. The debate on the differences between agents and objects exacerbated the situation: the supposed advantages over the object-oriented approach seem poor and the results of some comparisons are simply wrong. Just to cite a common mistake: people often forget that the metaobject protocol [9] was introduced in mid-80's and they keep saying that one big difference between agents and objects is that agents send messages while objects invoke methods.

This paper is meant to be a first step in the direction of answering the very basic question of why agents are a good paradigm for software development. I proceed toward this end in the following way: first, I compare agents with state-of-the-art technology for software development, i.e., software components like Enterprise JavaBeans (EJBs) [15], CORBABeans [14] and .NET components [6]. Then, I elaborate on the first main result of such a comparison: agents are more reusable and more composable than components. Finally, I discuss the second main result of the comparison: agents raise the level of abstraction with respect of component-based development. I formalize this new level of abstraction as a new system level and I briefly summarize its properties.

Before approaching the main topic of the paper, I need to briefly discuss the agent model that I use. I have to address this issue because components are a concrete object model and I need a concrete agent model to draw a reasonable comparison. I chose the agent model the ParADE [2] introduces. Such a model is a BDI-like incarnation and it was designed to focus on the main characteristics that agents share with components, i.e., reusability, composability and interoperability. It is common to emphasize this design focus using the word *agentware* for software build through the composition of agents of this sort. For the sake of brevity, I do not describe the model in much detail and I restrict the details to what I need to support the comparison with components. It is worth noting that the results of this work also apply to other BDI-like agent models.

## 2 A Model of Agentware

Agentware is software made through the composition of a set of interacting agents. In the ideal world, agents are taken from a repository of commercial off-the-shelf (COTS) agents and the multiagent system is composed and reconfigured on the fly to cope with varying conditions. Agents are known to the multiagent system by means of a unique identifier and their state is characterized through beliefs and intentions. You may want to use the word goals for such intentions because there is no difference between the two abstractions here. In the attempt to support different agent models, I do not go formal and I refrain from defining the properties of beliefs and intentions.

In addition to a unique identifier and a mental state, an agent has capabilities. These are described in terms of what the agent can do, i.e., the possible outcomes of its actions, and how the agent can interact with other agents.

Agents communicate through message passing and communication allows them to exchange representations of theirs and others beliefs, intentions and capabilities. Normally, beliefs, intentions and capabilities are represented through logic formulae and the ultimate purpose of communication is to exchange logic formulae that incorporate modalities for beliefs, intentions and capabilities. This may seem a rather extreme approach to agent-based communication, but it simply generalizes the available work on agent communication languages (ACLs). Let's take the FIPA ACL [7] as an example: it defines performatives together with feasibility preconditions and rational effects. When an agent receives a message, it can assert that the feasibility precondition holds for the sender and that the sender is trying to achieve the corresponding rational

effect. This is basically a rather knotty way to let the receiver know what is going on in the sender's mental state. The advantage of using a structured ACL instead of a more natural exchange of representations of beliefs, intentions and capabilities simplifies the development of reactive agents capable of complex interactions.

The semantics that we normally use with most popular ACLs rely on heavy assumptions on the capabilities of agents. For example, the FIPA ACL requires an agent to reason on the receiver's mental state before sending any message. This seems too strong a requirement if we want to use agents in large applications where many, and possibly unforeseen, interactions occur. This is the reason why ParADE introduces a FIPA-like ACL with minimalist semantics. Such a language provides an operational means for agents to exchange representations of beliefs, intentions and capabilities. Table 1 shows some performatives of the ParADE ACL together with their semantics. The semantics is modeled as the effect that the sender wishes to achieve when sending the message.

**Table 1.** Semantics of some performatives of the ParADE ACL

Message	Semantics
inform(s, r, p)	$I_s B_r p$
achieve(s, r, p)	$I_s I_r p$
queryRef(s, r, p(x))	$I_s B_s [B_r q \wedge \text{unifies}(p(x), q)]$
request(s, r, a)	$\  \text{achieve}(s, r, \text{done}(a)) \ $
agree(s, r, a)	$\  \text{inform}(s, r, I_s \text{done}(a)) \ $
refuse(s, r, a)	$\  \text{inform}(s, r, \neg I_s \text{done}(a)) \ $

The ParADE ACL provides a means for exchanging complex logic formulae through simple messages as the receiver can assert what the sender is intending. This is sufficient to ensure semantic interoperability [2] without revealing too much details of the mental state of the sender, i.e., without breaking encapsulation.

Strictly speaking, the particular sort of ACL is not part of the agent model that I am presenting. Anyway, I outlined the ParADE ACL to ground some considerations that I am going to make in the following section.

Isolated messages are not sufficient to allow agents to communicate fruitfully. The classic example is the case of an agent requesting another agent to perform an action. The first problem is that messages are asynchronous: there is no guarantee that the receiver would act in response to a message, i.e., there is no guarantee that the receiver would actually perform the action. The second problem is that the semantics of a single message might not be sufficient to express application-specific constraints. The semantics of *request* does not impose the receiver to communicate to the sender that the requested action has been actually performed. The sender might hang indefinitely while waiting for the receiver to tell it that the action has been performed.

In order to support fruitful communication, the agent model that I am sketching here provides *interaction laws*. These are rules that an agent *decides* to adopt to govern its interactions with other agents. The interaction laws that an agent decides to follow are part of its capabilities and they are published. The following interaction law is sufficient to solve the classic problems of *request(s, r, a)*:

$$\begin{aligned}
B_s I_r \text{done}(a) &\leftarrow I_r \text{done}(a) \\
B_s \neg I_r \text{done}(a) &\leftarrow \neg I_r \text{done}(a) \\
B_s \text{done}(a) &\leftarrow B_r \text{done}(a)
\end{aligned} \tag{1}$$

The first rule states that if agent  $r$  intends to perform  $a$ , then  $s$  must know it. The second rule covers the opposite case: if  $r$  decides not to perform  $a$ , then  $s$  must know it. The third rule says that as soon as  $r$  comes to know that  $a$  has been done, then also  $s$  must know it.

An interaction law has a precondition and a termination condition. In the example above, the precondition for  $r$  is  $I_s \text{done}(a)$ , i.e.,  $r$  received the request. The termination condition is  $B_r \text{done}(a) \vee \neg I_r \text{done}(a)$ . An agent starts applying an interaction law when the precondition is verified and stops as soon as the termination condition holds.

With the introduction of precondition and termination condition, interaction laws become an elegant and flexible way to describe interaction protocols. The example above is the interaction law that describes the FIPA request protocol [7]. Similar approaches to overcome well-known problems of the classic descriptions of interaction protocols are also available [13], but they do not exploit the semantics of the underlying ACL.

The interaction laws than an agent may adopt are linked to the possible roles it can play in the multiagent systems, and they may vary over time.

### 3 Comparison with Software Components

The comparison between agents and components starts from table 2. It shows some important component-oriented abstractions and associates them with agent-oriented counterparts.

**Table 2.** Comparison between component-oriented and agent-oriented abstractions

Abstraction	Component-oriented	Agents-oriented
Communication	Task delegation	Task and goal delegation
Message	Requests for actions	ACL messages
Interaction with the environment	Events	Updates of beliefs
State	Properties and relations	Mental attitudes
Interactions	Interfaces,	Capabilities,
between parties	Interface repository	Semantic matchmaker
Runtime	Application server	FIPA platform

Table 2 compares some of the abstractions that form the components' metamodel with the corresponding abstractions of the agents' metamodel. This suggests that a complete comparison should take into account also the abstractions that are missing in table 2. Following this approach, I should take into consideration, e.g., the .NET metamodel [6] and compare it with, e.g., the SMART framework [10]. This approach

seems to overkill the problem because many abstractions in these metamodels are too different and they are hardly comparable.

### 3.1 Comparing Abstractions

In the following, I take the component-oriented abstractions identified in table 2 and compare them with their agent-oriented counterparts. I did not choose any ranking criteria yet and therefore I cannot say which of the two approaches is better.

**Communication Model.** The main difference between agents and components is in the mechanism they use to communicate. Agents use ACLs, like the one that I introduced in the previous section, while components use a metaobject protocol [9]. In the agent-oriented approach, a message is sent to transfer part of the sender’s mental state to the receiver. A special case of this mechanism is when the sender endeavors to delegate a goal to the receiver, e.g., through the *achieve* performative. This special case, known as goal delegation [5], is the basic mechanism that agents use to delegate responsibilities.

The components’ metamodel does not comprise an abstraction of goal and therefore components cannot exploit goal delegation; rather they use task delegation. Components achieve their (implicit) goals asking to other components to perform actions; agents achieve their (explicit) goals delegating them to other agents. This is the reason why I refer to the agent-oriented communication model as *declarative message passing*: agents tell other agents what to do without stating how to do it. On the contrary, I use *imperative message passing* for the component-oriented approach because components cannot say to another component what to do without also saying how to do it.

The possibility of using task delegation only is a strong limitation for components because goal delegation is a more general mechanism. First, task delegation is a special case of goal delegation: the delegated goal has the form *done(a)*, just like in the semantics of the *request* performative. Then, task delegation may inhibit optimizations. Consider, e.g., a component *s* with a goal *g* that needs component *r* to perform *a<sub>1</sub>* and *a<sub>2</sub>* to achieve it; *s* would ask to *r* to perform *a<sub>1</sub>* and then it would ask to perform *a<sub>2</sub>*. As the two requests are not coupled though the underlying idea that *s* is trying to achieve *g*, *r* cannot exploit any possible cross-optimization between *a<sub>1</sub>* and *a<sub>2</sub>*.

If *s* and *r* were two agents instead of two components, *s* would simply delegate goal *g* to *r* and then *r* would decide autonomously the way to go, e.g., it would decide to perform *a<sub>1</sub>* and *a<sub>2</sub>*. This approach couples *a<sub>1</sub>* and *a<sub>2</sub>* through *g* thus enabling *r* to perform cross-optimizations between *a<sub>1</sub>* and *a<sub>2</sub>*.

**Interaction with the Environment.** The environment is a structural part of the agents’ metamodel. Agents execute in an environment that they can use to acquire knowledge. Agents can measure the environment and they can receive events from it. In both cases, agents react to any change in the environment through changes in their mental state. This is radically different from the component-oriented approach where the environment communicates with components through reified events. Components react to reified events after constructing a relation with them.

The component-oriented approach *seems* to better respect encapsulation than the agent-oriented approach: the state of the component is changed only when the component itself decides to change it in reaction to an event. If we go more in detail, we see that also the agent-oriented approach respects encapsulation. Agents have reasoning capabilities that are the actual responsible of any change in the mental state. Any direct push of knowledge from the sensors to the mental state is ruled through reasoning and the mental state remains encapsulated.

**State Representation.** Both agents and components are abstractions that comprise a state, but they have very different approaches to describe and to expose it. The state of a component is represented through a set of properties that other components can manipulate; such properties are the attributes of the component itself. In addition to properties, the state of a component includes its relations with other components. Such relations represent what the component knows about other components and how it relates to them.

In the model I introduced in the previous section, the state of an agent is represented through a set of beliefs, intentions and capabilities. The main differences with the component-oriented approach are:

1. Agents have an explicit representation of their goals;
2. Agents have explicit knowledge of their environment and not only of other agents;
3. Except for a unique identifier, agents do not have properties, they only have relations with other agents and with entities in the environment;
4. Agents may use deduction to come to know more than what other agents told them and more than what they measured.

It is worth discussing the last point a bit. Components can deduce the value of a property using application-specific mechanisms. The difference is that properties and relations are not easily structured in a logic framework and it is difficult to use general-purpose reasoning techniques with them; any reasoning is hardcoded in the component itself.

**Interaction between Parties.** The different communication mechanisms influence how agents and components open themselves to the outer world. Components use interfaces to enumerate the services they provide and to tell clients how to get in contact with them. Sophisticated component models equip interfaces with preconditions, post-conditions and invariants to support design by contract [11]. Anyway, the important issue of providing a description of the semantics of the services directly in the interface is still open.

The agent-oriented approach eliminates interfaces and provides agents with capabilities that describe what the agent can do, i.e., the possible outcomes of its actions, and how the agent can interact with other agents, i.e., the interaction rules it can adopt. The use of capabilities instead of interfaces has the advantage that we can easily describe the semantics of the services that an agent offers using:

1. High-level, mentalistic abstractions, i.e., beliefs and intentions;
2. The model of the environment, i.e., entities and their relations.

**Runtime Environment.** FIPA started its work in 1996 through the definition of a runtime environment to support interoperable multiagent systems. This work is valuable and today the agent-oriented counterpart of an application server is a FIPA platform, like JADE [1] and LEAP [3]. Actually, application servers are huge pieces of software that offer enterprise features that available FIPA platforms do not provide yet, e.g., transactional persistency and security. FIPA has accepted such a limitation of available implementations and it is going in the direction of creating an abstract architecture of its agent-oriented runtime. Such an architecture can be concretely instantiated using an available FIPA platform or any other runtime support, e.g., an application server. This choice saves both worlds and it seems one of the most reasonable. Anyway, it is worth noting that the very naïve approach of encapsulating an agent into a component so to run it in an application server has some drawbacks. The most remarkable one is that the threading model that the application server imposes to components may not be compatible with agents. Basically, the developer must choose between loosing some enterprise feature, e.g., fault tolerance and transparent scalability, and implementing only reactive agents. Some middleware providing a reasonable compromise are already available, e.g., Caribbean [18] and EJB 2.0 allow asynchronous messaging in EJBs.

### 3.2 Agents Improve Reusability and Composability

The comparison that I sketched above enumerates differences and similarities between components and agent, but it does not state whether an approach is better than the other because I did not adopt any ranking criteria. In this section, I consider two of the most important features of a development technology, i.e., reusability and composability, and I evaluate how and when agents are better than components from these points of view. As noted above, agents:

1. Can use goal delegation instead of task delegation only;
2. Use ACLs that exploit the agent model to give a semantics to interactions;
3. Have a mental state and can use reasoning to generate knowledge from events and messages.

These properties improve the reusability and the composability of agents with respect of components.

Goal delegation partially decouples the agent that delegates a goal from the delegate agent because the two are no longer coupled through a sequence of actions, but only through a shared goal. In concrete terms, this solves the *interface mismatching* problem, i.e., the problem of substituting a component with another component that offers the same services through a different interface. Client components are perfectly fine with the services of this new component, but they cannot actually use them because they do not know how to access the services. Agents solve this problem because the client agent simply delegates the goal, and then the delegate agent autonomously decides what to do to achieve it. Unfortunately, the interface mismatching problem is not completely solved because agents may support different ontologies, but this is an

easier problem especially if we implement open systems that deal with, so called, standard ontologies.

There are other similar problems concerning reusability and composability that goal delegation solves. I am not aware of any catalog of these problems and the enumeration and relative solution of all of them is out of the scope of this work.

Other improvements in terms of reusability and composability come from the agent-oriented approach to communication. It turns many common tasks, e.g., informing and querying, into application specific only in the sense that they depend on the ontology<sup>1</sup>. Components implement informing and querying through properties and relations and this couples communication with the information model, i.e., with the component-oriented counterpart of the ontology, and with the semantics of get/set messages. Agents remove the dependency on get/set messages and promote reusability and composability, provided that the ontology remains the same.

The logic-based model of mental states allows using deduction and means-end reasoning to infer knowledge. This means that many messages and events can be turned to few common situations. The concrete behavior of the agent is encoded only for a limited set of situations while allowing it to react to a wide range of events and messages. This promotes reusability and composability because the behavior of the agent is partially decoupled from the concrete messages and events that it can handle.

The three characteristics of agents that I have just discussed account for their superiority in terms of reusability and composability. If we try to generalize this result, we can see that their superiority is perfectly reasonable because agents support *semantic composability*. Agents rely on a common executor model, i.e., the agent model, that moves most of the semantics of the composition to the semantics of the ACL. This is exactly the opposite of what components do; the component model shifts most of the semantics of the composition to the semantics of the action that a component executes in reaction to a message or an event. The latter is syntactic composability because the semantics of communication does not exploit a common model of the executor.

If we adopt a shared and accepted ACL, e.g., the FIPA ACL, we can say that agents are semantically interoperable while components are only syntactically interoperable.

## 4 Introducing the Agent Level

People began enumerating the benefits of working at a high level of abstraction in the early days of computer science. When computers could only be programmed in assembly language people felt the urge of higher-level abstractions, e.g., types and procedures. Today, the component-oriented approach provides high-level abstractions, e.g., messages and events, and any further raise of the level of abstraction may seem a poor result. Actually, the level of abstraction that developers use impacts heavily on any figure we may evaluate to measure the quality of a product, and it should not come as a surprise that any increase of the level of abstraction drastically increases the quality of the final product.

---

<sup>1</sup> Messages depend also on the chosen content language, but this is not a real issue.



In order to show that the agent-oriented approach is at a higher level of abstraction than the component-oriented approach, I take Booch's words into account: "*at any given level of abstraction, we find meaningful collections of entities that collaborate to achieve some higher level view* [4]." The meaningful collections of agent-oriented entities comprise beliefs, goals and capabilities, and they are obviously closer to human intuition than component-oriented counterparts, e.g., property, event and message. Following the standard approach, I formalize this idea defining a new system level.

A system level [12] is a set of concepts that provides a means for modeling implementable systems. System levels are layered in a stack and each level is mapped on one lower level. System levels abstract away for implementation details and higher levels provide concepts that are closer to human intuition and far away from implementation. This is why we say that system levels are levels of abstraction<sup>2</sup>. System levels are structured in terms of the following elements:

1. Components: atomic building blocks for building systems at that level;
2. Laws of compositions: laws that rule how components can be assembled into a system;
3. A medium: a set of atomic concepts that the system level processes;
4. Laws of behavior: laws that determine how the behavior of the system depends on the behavior of each component and on the structure of the system.

Newell's knowledge level [12] provides a better perspective for characterizing agents than that of any operational characterization. Besides, Newell's work dates back to 1982 and it did not take into account many of the features that we would like agents to have today. In particular, Newell was very much concentrated on the realization of single agents capable of exhibiting intelligent behavior. Nowadays, we are no longer interested in realizing a system in terms of a single monolithic agent, and we prefer to use agents as building blocks of multiagent systems. The intelligence results from the interaction of agents and it is an emerging property of the whole system. In Wegner's words: "*Interaction enhances dumb algorithms so they become smart agents.*"

Jennings' proposal of the social level [8] goes in this direction trying to raise the level of abstraction of the knowledge level. This approach has the advantage of requiring minimal capabilities to agents, thus accommodating any sort of agent available in the literature. Nevertheless, it may remain too generic and its utility in building complex systems seems poor.

Taking into account the limitations of both Newell's and Jennings' approaches, I propose a novel system level called, and I will justify later why, *agent level*. The agent level is thought as a reasonable compromise between Newell's position, that avoids dealing structurally with interactions, and Jennings' view of interaction as the only relevant component of a system. At the agent level, single agents exhibit a rational behavior and they are described in terms of mental attitudes; they are also part of a society and they interact through an ACL. The agent level exploits the possibilities of

---

<sup>2</sup> Newell explicitly stated the contrary, but it seems that he used the phrase "level of abstraction" in a different way from how we use it in software engineering today.

rationality and it provides a way for distributing complexity across agents rather than centralizing intelligence into a single monolithic agent.

In order to define the agent level, I first enumerate the components that it comprises:

1. Agents, that interact to implement the functionality of the multiagent system;
2. Beliefs, that an agent has;
3. Intentions, that an agent is trying to achieve;
4. Actions, that an agent can perform on the environment where it executes;
5. Roles, that agents can play in the system;
6. Interaction laws, that rule how agents playing different roles can interact.

Having said that agents and mental attitudes are both components of the agent level should justify its name. This does not apply to the knowledge level, where the agent is the system you implement. It is also inappropriate for the social level, where agents are components with no particular characteristics, and the focus is on the society.

The laws of composition state that a multiagent system is composed of a set of interacting agents. Each agent is associated with a set of beliefs, a set of intentions, a set of actions and a set of roles. The interaction between agents is ruled through a set of interaction laws. Notably, this is not a limitation of autonomy because agents autonomously decide to play a particular role as a way to bring about their goals.

Exploiting the components of the agent level, we can define two familiar concepts that are no longer atomic:

1. Responsibility: an intention that an agent is constantly bringing about and that it does not drop even if it has already been achieved;
2. Capability: the post-conditions of an agent's action, i.e., what the agent is capable to achieve on its own, and the interaction laws it supports, i.e., how it can interact with other agents to achieve its intentions.

The medium of the agent level is the representation that each agent has about its and other agents' beliefs, intentions and capabilities. An agent processes the medium of the agent level, i.e., comes to know other agents' beliefs, intentions and capabilities, through interaction in order to achieve its goals. In my proposal, the principal means of interaction is communication and agents use an ACL to exchange representations of theirs and others beliefs, intentions and capabilities. Only agents produce and consume messages, and the computation of the multiagent system results from the messages that agents manipulate.

The law of behavior is an adaptation of Newell's principle of rationality: if an agent has knowledge that one of the actions in its capabilities will lead to satisfy one of its intentions or to respect one of its responsibilities, then the agent will select that action.

This principle is sufficient to connect the components of the agent level with the selection of an action without implying any mechanism through which this connection is implemented at lower system levels. It may be any technique of means-end reasoning or it may be hardcoded in the program of the agent. Table 2 summarizes the elements of the agent level.

**Table 3.** Elements of the agent level

Element	Element of the Agent Level
System	Multiagent system
Components	Beliefs, intentions, actions, roles and interaction laws
Medium	Representations of beliefs, intentions and capabilities
Behavior	Principle of rationality

Most of the discussions that Newell did about the knowledge level are still applicable to the agent level. In particular, just like the knowledge level, the agent level is radically incomplete and the agent's behavior cannot be designed without going to a lower level.

An agent-level model is not useful if we do not provide a means for concretely implementing the system. If we agree that the agent level can sit on top of the component level, then we need a mapping between elements of the agent level and systems at the component level. Generally speaking, any development tool that supports agent-level abstractions level actually implements this mapping. Just to cite one example, ParADE allows using UML diagrams to model a system at the agent level; then, it generates Java skeletons that are at the same level of abstraction of components.

## 5 Discussion

The whole length of this paper is devoted to give reasonable argumentations on why developers should seriously take into account the possibility of using agents for their products. The section on the agent level is there to provide a scientific foundation to such argumentations. The two most important outcomes of these discussions are that agents:

1. Provide high level abstractions to developers;
2. Have some advantages over components in terms of reusability and composability.

The first point, i.e., working with high level abstractions, has well-known advantages but it also has a typical drawback: speed. In order to fully exploit the possibilities of agents, we need to implement an agent model similar to the one described in section 2. This model heavily relies on reasoning and agents of this kind are likely to be slow. Nowadays, this does not seem a blocking issue because speed is not always the top-most priority, e.g., time-to-market is often more stringent.

The second point, i.e., reusability and composability, is worth discussing a bit because components are advocated as the most composable and reusable technology. Actually, the improvement that agents obtain comes at a cost: speed, again. The use of goal delegation instead of task delegation requires, by definition, means-end reasoning and we face the reasonable possibility of implementing slow agents.

Fortunately, in both cases, the performances of agents degrade gracefully. We can choose how much reasoning, i.e., how much loss of speed, we want for each and every agent. In particular, we may use reasoning for agents that:

1. Are particularly complex and could benefit from high level abstractions;

2. We want to reuse and compose freely in possibly different projects.

On the contrary, we can fallback on reactive agents, or components, when we have an urge for speed. This decision criterion seems sound because the more an agent is complex and value-added, the more we want to reuse it and compose it with other agents. Moreover, reactive agents are perfectly equivalent to components and we do not loose anything using the agent-oriented approach instead of the component-oriented approach.

## References

1. Bellifemine, F., Poggi, A., Rimassa, G.: Developing Multi-agent Systems with a FIPA-Compliant Agent Framework. *Software Practice and Experience* 31 (2001) 103–128
2. Bergenti, F., Poggi, A.: A Development Toolkit to Realize Autonomous and Inter-operable Agents. *Procs. 5<sup>th</sup> Int'l Conference on Autonomous Agents*, (2001) 632–639
3. Bergenti, F., Poggi, A., Burg, B., Caire, G.: Deploying FIPA-Compliant Systems on Hand-Held Devices. *IEEE Internet Computing* 5(4) (2001) 20–25
4. Booch, G.: *Object-Oriented Analysis and Design with Applications*. Addison-Wesley (1994)
5. Castelfranchi, C.: Modelling Social Action for AI Agents. *Artificial Intelligence* 103(1) (1998)
6. European Computer Manufacturer's Association: Standard ECMA-335, Partition II Metadata Definition and Semantics. Available at <http://www.ecma.ch>
7. Foundation for Intelligent Physical Agents. Specifications. Available at <http://www.fipa.org>
8. Jennings, N. R.: On Agent-Based Software Engineering. *Artificial Intelligence*, 117 (2000) 277–296
9. Kiczales, G., des Rivières, J., Bobrow, D.G.: *The Art of the Metaobject Protocol*. MIT Press (1991)
10. Luck, M., d'Inverno, M.: A Conceptual Framework for Agent Definition and Development. *The Computer J.* 44(1) (2001) 1–20
11. Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall (1997)
12. Newell, A.: The Knowledge Level. *Artificial Intelligence*, 18 (1982) 87–127
13. Singh, M.P., Yolum, P.: Commitment Machines. *Procs. 8<sup>th</sup> Int'l Workshop on Agent Theories, Architectures, and Languages* (2001)
14. Suhail, A.: *CORBA Programming Unleashed*. Sams (1998)
15. Sun Microsystems: *Enterprise JavaBeans Specification: Version 2.0*. Available at <http://java.sun.com>
16. Wegner, P.: Why Interaction is More Powerful than Algorithms. *Communications of the ACM* 40(5) (1997) 80–91
17. Wooldridge, M. J., Jennings, N. R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. *Int'l. J. Autonomous Agents and Multi-Agent Systems* 2 (1) (2000)
18. Yamamoto, G., Tai, H.: *Caribbean Application Developer Guide*. Available at <http://alphaworks.ibm.com>