

# Towards Agent Oriented Application Frameworks

Davide Brugali  
Politecnico di Torino  
Carnegie Mellon University  
and  
Katia Sycara  
Carnegie Mellon University

---

The goal of this paper is to present how techniques of the Agent Technology have been exploited to enhance software reusability and maintainability. The principal contributions are the definition of the concept of Agent Oriented Application Framework as a reuse technique and the comparison with Object Oriented Application Frameworks and Component Development.

Additional Key Words and Phrases: Software Agents, Component Customization

---

## 1. THE NEED OF A FLEXIBLE COMPUTATIONAL MODEL

Building a reusable application framework requires a deep understanding of its application domain in terms of the entities and the relationships that can be captured by reusable components and by the reusable patterns of interactions between the components.

When the application domain (e.g. large-scale open distributed applications, integrating heterogeneous systems) has rapidly evolving requirements, flexibility in the pattern of interactions between the components of a framework is mandatory if the framework itself is to be reusable.

Object Oriented (OO) Application Frameworks [Fayad and Schmidt 1997] have

---

This research has been sponsored in part by the project "Tecnologie di elaborazione distribuita nei servizi multimediali per l'azienda virtuale e il commercio elettronico" in collaboration with CSELT, Turin, Italy, by ONR Grant by ARPA Grant F33615-93-1-1330, and by NSF grant IRI-9612131.

Name: Davide Brugali

Address: C.so Duca degli Abruzzi, 10129 Torino, Italy

Affiliation: Dip. Automatica e Informatica, Politecnico di Torino, brugali@polito.it

Name: Katia Sycara

Address: 5000 Forbes Ave., Pittsburgh, PA 15213-3891

Affiliation: Robotics Institute, Carnegie Mellon University, katia@cs.cmu.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

proven to be an extremely powerful reuse technique, but in some cases they are excessively brittle. This is due to the fact that in OO programming the interactions between two components are specified and implemented as operations on one of the two: the more likely the interactions are to be changed in future applications, the less reusable is the component that expresses the joint behavior [Mili et al. 1995].

It is necessary to have a computational model which allows a higher adaptability of the reusable components of a framework in terms of the interactions with other components.

Several attempts to extend the traditional object model have been proposed (e.g. *Actors* [Agha et al. 1992], *Active Objects* [Minoura et al. 1993], *Distributed Object Model* [Vinoski 1997]) the last one being software agents (for a survey see [Riecken 1994]). Other attempts to enhance software flexibility are, for example, Connector Models [Allen and Garlan 1997], Interaction Protocols [Bokowski 1996], and Contracts [Holland 1992]. .

## 2. SOFTWARE AGENTS

Software Agents are integrated systems that incorporate major capabilities drawn from several research areas: Artificial Intelligence, Databases, Programming Languages, and Theory of Computing. Usually, agent-based systems are conceived as "one-off" systems built to investigate a single application. A new trend in Distributed Artificial Intelligence (DAI) (see for example [Sycara et al. 1996]) considers software agents as software units of design, that may be customized and composed with other similar units to build complex systems. The corresponding agent model is an abstraction that corresponds to functional aspects of real entities more directly than to blocks of executable code and extends the traditional object model in several ways:

- Agents, like objects, have an internal state which reflects their knowledge. However, this knowledge may be based on default assumptions or partially specified and refined during an agent's life time.
- Agents have reasoning capabilities which determine their internal behavior; the agent's behavior is usually specified in a declarative way (e.g. by rules, constraints, goals) and may change dynamically at run-time.
- Agents show an external behavior consisting of communicative acts to other agents (human or software) or control actions on software or hardware devices. Agents communicate with one another through standard agent-independent communication languages [Genesereth and Ketchpel 1994].
- Agents have an identity which distinguishes one agent from the other agents in the same system and it is expressed in terms of a name, an address, and a description of the services it provides [Sycara et al. 1996]. An agent's identity can be communicated to other agents and agents can freely appear and disappear in the system or change their location.

From the DAI viewpoint, most complex software systems and applications are conceived as organizations of cooperative agents. Agents can be part of a stand-alone application or be distributed over a network. Interactions among agents are established dynamically according to the dependencies among their capabilities.

The same capability may be provided by different (new) agents and the same agent may provide several (new) capabilities. Agents can cooperate since they share the same communication language and a common vocabulary, which contains words appropriate to common application areas and whose meaning is defined in a shared ontology.

Agents enhance system flexibility and adaptability since the software integration they allow is not exclusively at a syntactical level (where system components commonly agree on a set of data structure definitions and on the meaning of the operations on those structures), but rather at a semantic level: system components (agents) communicate in terms of knowledge transfer instead of data transfer.

Developing an application (e.g. a Distributed Portfolio Manager (DPM) [Decker et al. 1996]) as an aggregation of software agents is a process, which is closely related to well-known object-oriented praxis. Essentially, it is necessary to find the key abstractions in the problem space (e.g. the sources of information, the investment experts, the user of a DPM), assigning a role and a responsibility to each abstraction (e.g. an investment expert has to evaluate stock prices), and modeling and implementing each abstraction as an agent (e.g. the Investment Agent).

Researchers have applied multi-agent technology in several application domains: electronic markets [Merz et al. 1994], telecommunications [Busuioc 1994], and manufacturing [Swaminathan et al. 1998]. So far, few Agent Oriented (AO) Application Frameworks have been conceived for the development of multi-agent systems (see for example [Lejter and Dean 1996; Sycara et al. 1996; Conrad et al. 1997]).

### 3. AGENT ORIENTED APPLICATION FRAMEWORKS

The components of a reusable framework are classified as elemental, basic design and domain dependent [Brugali et al. 1997].

The *elemental components* of an AO Application Framework are the building blocks for the implementation of individual agents. As an example, the RETSINA framework [Sycara et al. 1996] provides the following elemental components:

- The *Communication and Coordination* component accepts and interprets messages from other agents, and sends replies.
- The *Planning* component produces a plan that satisfies the agent's goals and tasks.
- The *Scheduling* component schedules plan actions.
- The *Execution Monitoring* component initiates and monitors action execution.

The *basic design components* are specific agent architectures which integrate the elemental components of the framework. They differ from one another and the difference will depend on their internal structure (layered, blackboard, subsumption) and the distribution control protocols they adopt (client/server, peer-to-peer, pipeline). As an example, the RETSINA framework provides a Belief-Desire-Intention (BDI) agent architecture which integrates the elemental components using the following data structures:

- The *Task Schema Library*, that contains both domain independent and domain dependent plan fragments indexed by goals. These plan fragments are retrieved and incrementally instantiated according to the current input parameters.

- The *Belief Database* that contains facts, constraints and other knowledge reflecting the agent’s current model of the environment and the state of the execution.
- Schedule* that depicts the sequence of the actions that have been scheduled for execution.

The *domain dependent components* consist of the customizable implementation of agents with domain-specific functionality and resources (e.g. the Investment Agent is a reusable component for the financial domain). They can be classified in three broad categories: Information Agents, Task Agents and Interface Agents. Interface Agents manage graphical user interfaces, which transform user commands into agent’s objectives and display results. Information Agents wrap sources of information like a database or a Web-Server. Task Agents have specific reasoning capabilities or wrap legacy systems, which can be treated as procedural code invoked by the execution monitor. A multi-agent system usually consists of a set of collaborating agents belonging to all these categories.

### 3.1 Component Customization

Concrete agents are generated by customizing the variable aspects of the framework components. There are basically three kinds of agent customization:

- White-box customization: the framework provides a generic implementation of the basic agent architecture, which the user specializes encapsulating specific capabilities and resources (e.g. a graphical user interface).
- Black-box customization: the framework provides specific implementations of each elemental component (e.g. the *Planner*) that are to be plugged-in into the basic agent architecture.
- Gray-box customization: for each specific agent implemented when the framework is used (e.g. a *Task Agent*), the developer has to specify the agent Knowledgebase and Task Schema library.

All of the three kinds of customization frequently occur in the development of a concrete agent.

### 3.2 Re-inversion of control

As a reuse technique, AO Application Frameworks closely resemble both Component Off-The-Shelf (COTS) [Boehm 1995] Development and OO Application Frameworks.

Component-based development consists in building new applications by assembling previously-existing black-box software components. The developer designs the architecture of the application and writes the ”main function” that defines the control flow and information flow between the components. Integrating components requires a considerable effort since their design and implementation is fixed and cannot be customized.

On the contrary, OO Application Frameworks support the so called ”inversion of control”: the framework provides the application architecture and the main control flow, which defines how the user-provided components are executed. The developer builds a new application by customizing the variable aspects of the framework:

usually a framework supports both white-box and black-box customization. White-box customization of a framework requires a deep understanding of the framework implementation.

Both Component Development and OO Application Frameworks require the developer to code a considerable part of each new application. AO Application Frameworks mainly differ from them in that they are intended to promote grey-box customization. Ideally, a new application is built by selecting concrete implementations of agents with a specific functionality and integrating them to form a multi-agent organization.

Different from COTS development, agent integration does not require the developer to write a high-level function that coordinates the agent activities. Instead, the developer creates relationships between the agents and determines the information and control flow between them by customizing the variable aspects of each single agent.

In particular, for each agent the developer specifies (1) the knowledge and capabilities (data, rules, constraints) that the agent needs in order to fulfill the assigned responsibility, (2) the inter-agent communication protocol (that is, how the agent may be used by other agents), (3) the external dependencies between an agent's functionality and resources (Task Schema).

We call this approach to software integration "re-inversion of control" so as to emphasize the differences with the more common approach of OO Application Frameworks. The behavior of the whole application is determined by the behavior of each single agent and their mutual interactions. Interoperability between agents is guaranteed by the common infrastructure and architecture provided by the framework used to implement them.

AO Application Frameworks also differ from "horizontal frameworks" such as CORBA [Vinoski 1997] in the approach to system integration. CORBA provides a set of middleware services for object distribution and interoperability. Specific applications are built on top of the CORBA infrastructure and have to be designed from scratch. On the contrary, AO Application Frameworks are semi-defined applications which solve most of the difficult problems in a specific application domain.

### 3.3 System Adaptability

The declarative representation of capabilities, constraints and execution behavior enable agents to reason about the inter-dependencies of their functionality and to dynamically respond to situations that are not envisioned by designers when the application is constructed (e.g. availability of information sources, changes in information format, changes in performance requirements, runtime exception conditions, etc.)

This means that a multi-agent organization uses specifications of the tasks that need to be performed to select and integrate components (collaborating agents). Collaborating agents are found via requests and matching their capabilities, constraints and input/output information through *middle agents*, i.e. specific information agents provided by the framework that know which agent can provide a needed service [Brugali and Sycara 1998].

Any agent that enters the system and intends to let other agents use its services, must clearly declare this intention to a middle agent by making a commitment

to taking on a well-defined class of future requests. This declaration is called an *advertisement* and contains a specification of the agent's capability with respect to the type of request it can accept. An advertisement can also include meta-level information such as how much the service costs (these costs can indicate resource scarcity), how responsive the service agent usually is, etc.

Middle agents allow a system to operate robustly in the face of agent appearance and disappearance, and intermittent communications, characteristics that are present in uncertain environments such as Internet. If, at runtime, an agent providing some capabilities is unavailable, another agent with the same capabilities can be found (if one exists) and dynamically inserted into the system.

#### 4. CONCLUSION

A number of researchers are reexamining OO's basic promise, that the object paradigm can give the programmer tremendous flexibility [Guerraoui 1996]. Several researchers are exploiting ideas from other disciplines in order to enhance the flexibility of the object model in the development of complex systems. Agent technology is an attempt to accommodate basic OO concepts (e.g. abstraction, modularity) and advanced Artificial Intelligence techniques (e.g. reasoning, learning). The promise is that they will provide the programmer with a basic unit of design (the Agent), which enhances software modularity, maintainability, and reusability. We have pursued this idea further, by raising the level of reuse from single agent components to entire architectures (multi-agent systems) and up to the development of application frameworks.

#### REFERENCES

- AGHA, G., MASON, I., SMITH, S., AND TALCOTT, C. 1992. Towards a theory of actor computation. In *Proc Third Int. Conf. on Concurrency Theory (CONCURR'92)* (1992), pp. 565–579. LNCS 630, Springer-Verlag.
- ALLEN, R. AND GARLAN, D. 1997. A formal basis for architectural connection. *ACM Trans. on Software Engineering and Methodology*.
- BOEHM, C. 1995. Reuse emphasized at next process workshop. *Software Engineering Notes* 20, 5 (November).
- BOKOWSKI, B. 1996. Interaction protocols for composing concurrent objects. In *Proc. ECOOP'96, Workshop on Composability Issues* (1996). dpunkt, Heidelberg.
- BRUGALI, D., MENGA, G., AND AARSTEN, A. 1997. The framework life span. *Communications of the ACM* 40, 10 (October), 65–68.
- BRUGALI, D. AND SYCARA, K. 1998. Agent technology: a new frontier for the development of application frameworks? In M. FAYAD, D. C. SCHMIDT, AND R. E. JOHNSON Eds., *Object Oriented Application Frameworks*. New York: John Wiley and Sons.
- BUSUIOC, M. 1994. Distributed cooperative agents for service management in communications network. In *Proceedings of IEE Eleventh UK Teletraffic Symposium. Performance Engineering in Telecommunication Networks* (1994). BT Labs.
- CONRAD, S., SAAKE, G., AND TUERKER, C. 1997. Towards an agent-oriented framework for specification of information systems. In *Agents'97 Conference Proceedings* (February 1997).
- DECKER, K., SYCARA, K., AND ZENG, D. 1996. Designing a multi-agent portfolio management system. In *Proceedings of the AAAI'96 Workshop on Internet-based Information Systems* (August 1996).
- FAYAD, M. AND SCHMIDT, D. 1997. Object-oriented application frameworks. *Communications of the ACM* 40, 10 (October).

- GENESERETH, M. AND KETCHPEL, S. 1994. Software agents. *Communications of the ACM* 37, 7 (July), 48–53.
- GUERRAOU, R. 1996. Strategic directions in object oriented programming. *ACM Computing Surveys* 28, 4 (December).
- HOLLAND, I. 1992. Specifying reusable components using contracts. In *Proc. ECOOP'92* (1992). Springer Verlag.
- LEJTER, M. AND DEAN, T. 1996. A framework for the development of multi-agent architectures. *IEEE Expert, Special Issue on Intelligent Systems and their Applications* 11, 6 (December), 47–59.
- MERZ, M., MUELLER, K., AND LAMERSDORF, W. 1994. Service trading and mediation in distributed computing systems. In *Proceedings of IEEE Int. Conf. on Distributed Computing Systems* (1994), pp. 450–457. IEEE Computer Society Press.
- MILI, H., MILI, F., AND MILI, A. 1995. Reusing software: Issues and research directions. *IEEE Trans. on Software Engineering* 21, 6 (June), 0–0.
- MINOURA, T., PARGAONKAR, S., AND REHFUSS, K. 1993. Structural active object systems for simulation. In *In Proc. of OOPSLA '93* (October 1993). IEEE Computer Society Press.
- RIECKEN, D. 1994. Intelligent agents. *Communications of the ACM* 37, 7 (July).
- SWAMINATHAN, J., SMITH, S. F., AND SADEH, N. M. 1998. Modeling supply chain dynamics: A multiagent approach. *Decision Sciences*.
- SYCARA, K., PANNU, A., WILLIAMSON, M., AND ZENG, D. 1996. Distributed intelligent agents. *IEEE Expert, Special Issue on Intelligent Systems and their Applications* 11, 6 (December), 36–46.
- VINOSKI, S. 1997. Corba: Integrating diverse applications within distributed heterogeneous environments. *Proceedings of IEEE Communications Magazine* 14, 2 (February).