Patterns for Cooperation

Amund Aarsten, Davide Brugali, Giuseppe Menga

Dept. of Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi, 24
10129 Torino - Italy
tel. +39 11 564 (7012), fax (7099)
E-Mail : {amund, brugali, menga}@polito.it

Abstract

Collaborating autonomous agents are becoming increasingly attractive for systems where problem solving and decision making must be distributed. Unfortunately, they are not yet practical for industrial application.

Selected aspects of agent theory, however, can be very useful to software development. In this paper, we give three patterns which can form a base for this.

The patterns are the result of a process improvement project for the textile industry, which we will use as an example.

INTRODUCTION

In recent years, the field of autonomous collaborating agents have received increased interest. This is due to some fundamental requirements encountered in industrial systems which are difficult to meet using traditional centralized control architectures; the most significant being robustness, scalability, reconfigurability, and reusability. Furthermore, industrial applications are usually complex, and complexity demands decomposition.

The term "agent" is currently used in many different ways[1]. For our purposes, an agent is a computer program with specialized skills, an expert which collaborates with other experts. A system of intelligent agents would resemble a software application based on distributed objects, but with several novel characteristics. In contrast to most traditional systems, which adopt a hierarchical architecture, the underlying structure is flat, and the high level control activity is distributed among all the agents; agents communicate directly with each other through a common medium and have the ability to take advantage of the capability of other agents cooperating with them. Therefore, the behavior of a system of intelligent agents depends on the way each agent decomposes the assigned task and on the relations of collaboration that it establishes whit other agents.

Full-fledged intelligent agents are starting to emerge in some research communities. However, agent technology is still a very new and immature field, and far from being a realistic alternative for industrial

[1]A discussion of agent technology, and examples, can be found in ˝[CACM94].

applications [16]. In order to learn more about agents' role in industry and make their application practical, we think that the introduction of the technology must ˝be gradual.

In this paper we show how we applied, in a recent project, some agent concepts to an industrial process in the field of textile manufacturing, where different companies have to cooperate to produce the final product. We look at the architecture in an agent based system in terms of objects and the visibility problems involved by that architecture, as well as how the ˝cooperation between agents can be realized. The patterns in this article represent an attempt to extract some ˝reusable and domain-independent principles.

The Project

The project under consideration will create a supply and distribution chain in the volatile (seasonal) environment of high fashion men/ladies' wool garnment manufacturing. The process encompasses spinning, weaving, dying/finishing and ready made clothing in a typical multi-sites multi-suppliers international environment. The different local factories are ˝seen as cells: each cell, while operates to achive its best advantages, cooperates with other cells to improve the overall performance of the whole system in terms of quality, costs and time response. This project involves many companies, each one Europe-wide representative for one of the key phases of the whole textile ˝production cycle.

One typical process in such a cycle could be when a weaving company needs raw products. In this case, the company would negotiate with different spinning companies in order to find the available offer that best fits its needs, in terms of cost, delivery time, and other performances. Evalutating an offer often requires a "what-if" scheduling of jobs and resource allocation; at the same time, each spinning companies may have to perform a similar planning for responding to the weaving company's request. This evaluation iterates until the best offer from for each spinning company is ˝found.

Our contribution to the project has provided the architectural infrastructure that allows the different local factories to automate the cooperation process. For this purpose a local factory has been modeled as an agent which cooperates with other similar agents.

PATTERN 1: COMPATIBLE HETEROGENEOUS AGENTS

Context

System behavior is the result of collaborating agents. Collaboration is a dynamic activity which may involve different partners at different times. Intelligent agent are often able to decompose their tasks in subtasks and to initiate collaboration with expert agents which can carry out the different subtasks. This dynamic activity requires agents to have homogeneous aspects in ˝order be able to understand each other.

However, in the industry, existing information systems are intrinsically heterogeneous. These systems can not be thrown away when a new technology is introduced; they must be integrated with it. Agents should therfore offer an abstract view of the underlying systems which provide the specific capabilities and functionalities that make the agents expert. The balance between heterogeneity of the information systems and homogenity provided by the agents has a strong impact on the kind of information that each agent should maintain of the others, and therefore on the reusability and scalability of the entire system. Furthermore homogeneity avoids the need to impose too strong coupling among agents, which reduces the possibility of introducing new resources and capabilities ˝without any changes in the pre-existing system.

Problem

We must find the right balance between using a standard representation for agents and being able to use different types of agents. To which level of detail should be agents understand each other's activity? How can we integrate the existing systems used in the industry into ˝an agent system? How can we reuse existing agents when developing new ones? We need techniques for agent system development that are flexible enough to allow all of the above.

Forces

Minimizing the standard representation required of each agent, i.e. the set of capabilities, puts less restrictions on the types of agents which can be integrated into the system. It also makes agents less complex.

On the other hand, a limited set of capabilities can mean fewer possibilities for information exchange, and therefore less profitable collaboration.

Solution

Represent each capability as an object interface. Limit the standard capabilities common to every agent to include support for the following:

- Interface negotiation: an agent can be queried for its capabilities. The interface negotiation is based on a simple mechanism for querying an agent whether it supports a given interface (or a compatible specialization of it, as will be explained below.) This is an ˝explicit interface query, as in the Component Object Model [11]. Another possible solution could be to have a dynamic invocation mechanism as in CORBA [10]. This is necessary since agents can be introduced in different and dynamic environments, where the possibility to adapt to different types of ˝collaborators is essential.
- Collaboration: the ability to exchange information and share resources with other agents, in order to carry out a complex or decentralized activitity. We will cope with the collaboration problems in pattern 3, *Leader/Collaborator/Collaboration*.

Organize the interfaces representing agent capabilities in a generalization/specialization graph, where specialized interfaces are compatible with their generalizations. In a mature application domain, there will probably be a standardized graph of agent interfaces. New capabilities can often be introduced as specializations of existing ones, therefore being compatible with the existing system and thus offering seamless integration.

This way each agent can take advantage of the cooperation of other agents by decomposing its tasks following the capability classification and by negotiating with the collaborators the adequate service interfaces.

A typical interface negotiation scenario could be as follows: an agent searching for a suitable collaborator would ask all candidates if they support a general interface, perhaps not optimal but adequate for the job at hand. The agent would then query those who supported it for more specific interfaces, repeating the process until an optimal collaborator was found. This is ˝comparable to the service subscription negotiation process in TINA-C [17].

Existing systems can usually be integrated in the agent system through the application of the ADAPTER pattern from [2].

Implementation

Define an abstract class *Agent* which offers a common interface for every kind of intelligent system belonging to the agency, and supports the basic mechanisms for interface negotiation and collaboration between agents.

We implement the interface negotiotion support by giving each capability interface a name, and including in each agent a collection holding the interface names it supports. The method *queryInterface()* takes the name of an interface as parameter and simply looks it up in this collection. Note that it might seem natural to map each capability interface to a C++ abstract class, and using *dynamic_cast* or a similar mechanism to find out whether an agent supported a given interface or not. However, while certainly practical for a non-distributed prototype, it relies on multiple inheritance and creates complex inheritance dependencies which are better avoided in a ˝distributed context.

An agent has to be able to perform several services concurrently. Therefore, define a class *Server* (see for instance the Pattern *Client/Server/Service* in [6]), which manages independent threads of control and their concurrent access to the resources encapsulated by the ˝agent.

Derive by inheritance (see fig. 1) a class *AgentImp* from *Agent* and from *Server*. Build every new agent as specialization of the class *AgentImp*. This solution is inspired by the use of separate interface objects [Coplien], but using inheritance to couple the interface to its implementation as proposed in the pattern *Interface to Control Modules* in [6]; it also corresponds to the *class adapter* version of the ADAPTER pattern from [2]. When an existing system has to be integrated in agent system, the *AgentImp* acts as the *object adapter* version of that pattern.

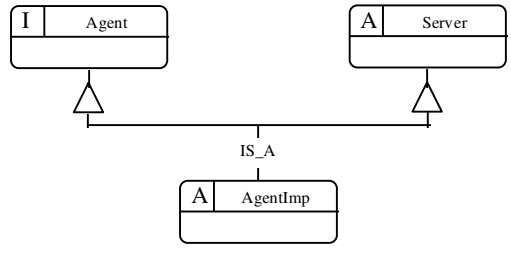

Figure 1 The Class AgentImp

Example

As mentioned in the introduction, we consider the automation of the order negotiation process, generating contracts for approval by the parties involved. For this purpose, we model each company as an agent, which is able to receive production orders, to plan its activity in order to satisfy the requests, to negotiate with other agents (customer and suppliers) production parameters (delivery time, quantity and quality of raw and finished materials) in order to optimize the use of its resources, to exchange information with the end-user.

For this purpose, we defined the interface *TextileAgent* derived from *AgentImp*, which provides, in addition to the interface negotiation and collaboration functionalities, some capabilities which are common to every company. As an example we consider the following:

- SupplyTextileProducts(): this method allows a customer to ˝specify an order to a supplyer;
- BuyTextileProducts(): this method allows a supplyer to confirm to a customer the requested product or to provide a counter-proposal;

- ScheduleTextileProduction(): this method allows an end-user to ˝monitor the status of its order.

We represented each type of company with different interfaces derived from *`TextileAgent`*; in particular (see fig. 2) we defined the interface names *`SpinningAgent`*, *`WeavingAgent`*, *`DyingAgent`*, and *`FinishingAgent`*, each implemented by a class of the same name. For most companies, the class implementation adapted the existing ˝information system.

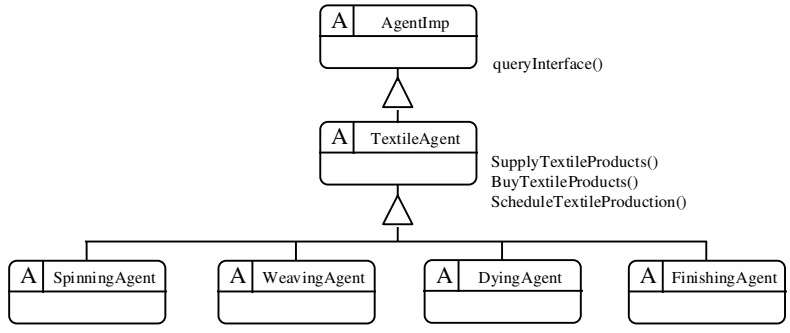

Figure 2 The classes implementing the textile agents

PATTERN 2: VISIBILITY AND COMMUNICATION BETWEEN AGENTS

Context

A complex architecture is decomposed in several autonomous agents that communicate with each other in order to cooperate. The need to communicate induces visibility requirements, with consequences on the reusability of the design. Different communication mechanisms may have different implications on visibility and reusability.

In addition, agents that adapt existing systems need to ˝interact with them.

Problem

How can we identify the necessary visibility requirements in a design, i.e. which other agents should an agent see, and how does it get visibility of those agents? Which is the appropriate communication mechanism in each case? How can we minimize the impact these ˝requirements have on the reusability?

How does an adapting agent communicate with the underlying ˝existing system?

Forces

A higher number of visibility relationships makes communication easier. However, each visibility relationship is a dependency which reduces the reusability of ˝the system.

The volatile parts (i.e. the often changing components) of the system need flexibile and dynamically reconfigurable visibility relationships. This can imply more complex communication mechanisms, which can lead to more work for the programmers and possibly slower performance. Therefore, for the other parts of the system where visibility relationships are more static, the same communication mechanisms may not be appropriate.

The existing system which underlies an adapting agent has no knowledge of the agent, and hence no visibility. But the agent still needs to use and monitor the ˝system's services.

Solution

Where the structure of cooperation relationships is stable, i.e. not subject to frequent changes, the reusability of the design is not greatly compromised by giving the agents involved direct visibility of each other. Having this direct visbility, they can communicate through normal object-oriented method calls, i.e. the Caller/Provider mechanism [6]. Each agent can issue a call to any agent it has direct visibility of, so the call goes in the direction of the visibility relationship.

Where the structure of cooperation relationships is unstable, agents should not have direct visibility of each other. In this case, a different communication mechanism is used, called Broadcaster/Listener [6]. This mechanism is achieved by giving all the agents of the architecture the capability of broadcasting and listening (or subscribing) to events, and can be seen as an implementation of the pattern OBSERVER in [2]. Here, the agents who raises the event does not know who might receive it, but the subscribers receive a reference to the sender along with the event. In this case, the call is therefore in the opposite direction of the visibility relationships.

In the case of adapting an existing system, the adapting agent need visibility of that system and therefore communicates with it using the Caller/Provider mechanism. The underlying system, in contrast, cannot have visibility of the agent; therefore, the only way the agent can monitor and receive information from it is by subscribing to the events it raises; hence, the ˝Broadcaster/Listener mechanism is used.

Implementation

In order to have visibility of its collaborators, every agent includes a collection of references to the agents with which collaboration is needed.

Since the agents have a common interface, this offers several ˝advantage:

- every agent has a common abstract view of its own and ˝others' local resources
- the interface to other agents filters the external access to ˝their local resources.
- the topology of the interconnections among agents is easily reconfigurable: When a new agent is introduced into the system, it broadcasts information about ˝itself for making itself visible.

The references to other agents (*`AgentProxy`* in figure 3) act as interfaces to the collaborators and are derived from the class *`Agent`* discussed in Pattern 1. The functionality of an *`AgentProxy`* object corresponds to that of the CORBA stub [9]. Every agent, while communicating with collaborators, refers to these interfaces instead of the agent itself. The *`AgentProxy`* forwards each request over the communication links to the remote agent it represents or ˝provide access to its local address space.

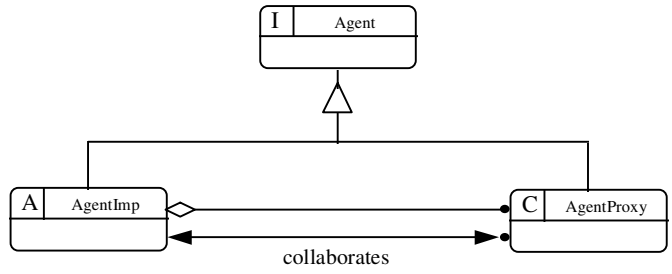

Figure 3: Visibility among agents.

Example

In the project, the *Caller/Provider* mechanism is significant when a commercial relationship has already been established between two companies and either sides of the two parties want information (e.g. in the case when an authorized customer has access to the information pertaining its own ongoing orders.) In this case, the partners in the relationship have visibility of each other. The *Broadcaster/Listener* mechanism is used when a commercial relationship has still to be established ˝(e.g. when a potential customer negotiates a new order, or a supplier performs marketing on the system.) Here, the initiative taker broadcasts an event, and interested parties subscribing to that type of event can ˝then consider initiating a collaboration.

PATTERN 3: LEADER/COLLABORATOR/COLLABORATION

Context

Agents can autonomously plan their activity, but also negotiate joint plans with other agents in order to take mutual advantage of their internal resources and capabilities; this is necessary e.g. for automated contract negotiation.

Problem

An agent must be allowed to collaborate simultaneously with any number of collaborators. Collaboration implies sharing of resources and capabilities.

How can we provide the intra-agent visibility needed to allow cooperation between agents and realize the peer-to-peer communication and concurrency management necessary ˝for collaboration?

Forces

For two agents to collaborate, they need visibility of each other, and also to know about each other's capabilities. As explained in pattern 2, these dependencies can greatly compromise the reusability of the design.

Since an agent can participate in any number of simultaneous collaborations, the concurrent access to the agent's resources becomes a problem. However, it should be possible to design agents without having to deal with the tiresome and error-prone details of resource ˝access synchronization.

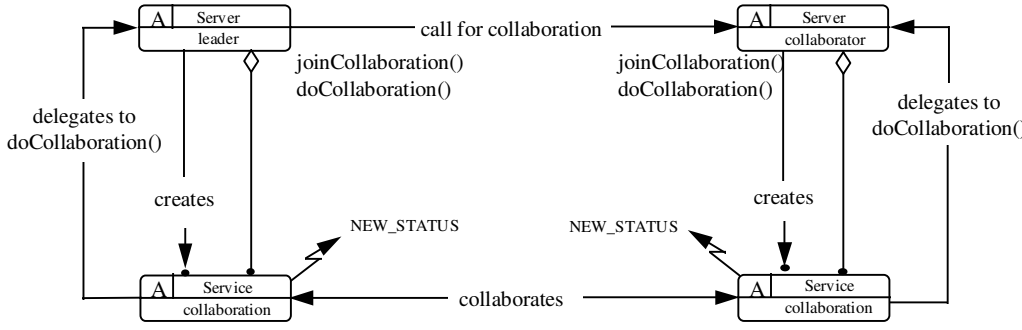

Figure 4: Leader/Collaborator/Collaboration

Solution

Give collaborating agents temporary visibility of each other during the collaboration, and encapsulate the agent's resources in blocking objects [6] which manange ˝concurrent access.

These requirements are satisfied by the Leader/Collaborator/Collaboration model proposed in this pattern. It is based on the pattern *Client/Server/Service* of [6], but new roles are assigned to the different participating classes.

When an agent needs collaboration to fulfill its task, it initiates a sequence of action to establish collaborations. We call this agent the *leader*. It asks one or more *collaborators* for collaboration, indicating the capability it is interested in. If the *collaborators* agree to collaborate, they create an instance of a special object called *collaboration*, which encapsulates a thread of execution and manages the collaboration process of the agent.

A reference to the *collaboration* is sent back to the *leader*, which creates an own *collaboration* object and sets it up for working directly with its counterpart. From this point, there is no longer any distinction between the roles of *leader* and *collaborator*; the communication is peer-to-peer. *Collaboration* objects have direct visibility of each other for the duration of the ˝collaboration.

Each pair of *collaboration* objects works independently and performs a different thread of collaboration between the agents which they belong to. This way, it is even possible that the same pair of agents establish different of *collaboration* instances concurrently.

Implementation

As said in pattern 1, the class `AgentImp` is derive by inheritance from `Agent` and from `Server` (see fig. 1). In order to provide the agents with the collaboration capabilities, we designed the class `Server` following the pattern Client/Server/Service of [6].

The `Server` is an abstract class that has to be redefined in order to build concrete servers. It creates and contains a collection of `Service` objects, each of which executes the server operations in concurrency, while sharing the same set of resources. A server's service can, in turn, use another server and in this case it acts as a client for that server.

Looking at fig. 4, in the Leader/Collaborator/Collaboration pattern, the *leader* and the *collaborator* are implemented as instances of the class `Server` and each *collaboration* is an instance of the class `Service`, an encapsulation of an operating system thread.

When the *leader* needs collaboration from another agent, it creates a local instance of *collaboration* (let us call it `lCollab`) and invokes the method *joinCollaboration()* of the *collaborator* passing a reference to `lCollab` and the name of the required capability as parameters. The *collaborator* responds by creating its own instance of *collaboration*, `cCollab`, passing it the reference to `lCollab`. It then returns a reference to `cCollab` to the *leader*.

Once the collaboration is enstablished, the collaboration process is managed by `lCollab` and `cCollab`, and the roles of client and server between *leader* and *collaborator* may be interchangable. The two *collaborations* communicate by exchanging events following the Broadcaster/Listener mechanism described in pattern 2.

Example

In our project, we used collaboration to schedule and optimize the overall production cycle, taking into account the dependencies between the involved parties. For instance, the thread produced by a spinning company is needed by a certain date by a weaving company; if this date slips, it can have consequences also for the dying and and finishing companies that use the fabric produced by the weaving company. A single agent represents one of the factories, and is responsible for planning and scheduling the production, with the goal of minimizing inventories, reducing lead time, and respecting release time and/or due date constraints.

If agents are considered separately, the combined planning problem can be split into a series of local replicas, for each of which release times and due dates are defined by completion times and start times of productions in neighboring agents. Optimization of these problems can be performed independently, as long as the release time constraints between modules are ˝guaranteed.

This intuitive topological decomposition results formally as a solution of a mathematical programming problem by relaxing (through an *augmented Lagrangian* approach [10]) the temporal constraints between agents saying that for a given product each operation can only start when the previous one is finished. Therefore, each agent can perform its scheduling locally; these results are then combined by the pairs of agents which are linked in a producer-consumer relationship, who exchange local planning results including temporal constraint violations. The optimization process is iterative; each iteration uses the values (local and combined) from the previous pass.

CONCLUSION

The patterns presented here build on the G++ pattern language for concurrent and distributed systems [6], to which we have added patterns for simple cooperative agent systems. As opposed to most research in agent systems [16], this work starts from an industrial reality and applies some new mechanisms to realize some of the promises of agent technology. We believe that this is the most realistic path towards the distributed intelligent systems of the future.

REFERENCES

[1] G. Blair et. al. *Object-oriented languages, systems, and applications*,. Halstead Press, 1991.

[2] E. Gamma, R. Helm. R. Johnson, J. Villisides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[3] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[4] G. Borghi, D. Brugali, *Autonomous Map Learning for a multi-sensor mobile robot using diktiometric representation and negotiation mechanism*. Proceedings of ICAR'95, Sant Feliu de Guixols (Spain), 20-22 September 1995.

[5] J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991

[6] A. Aarsten, G. Elia and G. Menga. *G++: A Pattern language for computer integrated manufacturing.* In *Pattern Languages of Program Design.* Addison-Wesley, NJ, 1995.

[7] A. Aarsten, D. Brugali and G. Menga. *Designing Concurrent and Distributed Control Systems: an Approach Based on Design Patterns*. Communications of the ACM, November 1996

[8] R. A. Brooks, *A Robust Layered Control System For A Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. RA-2 NO. 1 March 1986.

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, September 1992.

[10] B. B. Welch and J. K. Ousterhout. Prefix tables: *A simple mechanism for locating files in a distributed systrem*. In Proc. 6th Intl. Conf. on Distributed Computing Systems, pp. 184-189, May 1986.

[11] OLE 2 Programmer's Reference, Vol. 1/2. Microsoft Press.

[12] H. E. Bal, J. G. Steiner, A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. ACM Computing Surveys, 21(3):261-322, 1989.

[13] R. S. Chin and S. T. Chanson. Distributed object-based programming system. ACM Computing Survey, 23:91-124, March 1991.

[14] T. Minoura, S. Pargaonkar, K. Rehfuss. *Structural Active Object Systems for Simulation*. Proceedings of OOPSLA'93, Washington D.C., USA, October ˝1993.

[15] G. R. Andrews. *An Overview of the SR Language and Implementation*. ACM Transaction on Programming Languages and Systems, 10(1):51-86, 1988.

[16] Roda, N.R.Jennings & E.H.Mamdani, *The Impact of Heterogeneity on Cooperating Agents*. Proc. AAAI Workshop on Cooperation among Heterogeneous Intelligent ˝Systems. Anaheim, USA, 1991

[17] Telecommunications Information Networking Architecture Consortium. *Service Architecture Draft*. February 1995.

[18] F. Della Croce, G. Menga, R. Tadei, M. Cavalotto and L. Petri, *Cellular control of manufacturing systems*. European Journal of Operational Research 69 (1993) pp. ˝498-509