MULTIAGENT SYSTEM DESIGN BASED ON OBJECT ORIENTED PATTERNS

Elizabeth A. Kendall, Margaret T. Malkoun, C. Harvey Jiang
Computer Systems Engineering, Royal Melbourne Institute of Technology
City Campus, GPO Box 2476V, Melbourne, VIC 3001 AUSTRALIA
email address: kendall@rmit.edu.au

ABSTRACT

The relationships between agents and objects, and the role of object oriented analysis in multiagent system development have been discussed in earlier papers; these findings are employed here as a foundation for agent design. Patterns are flexible, reusable, and effective object oriented designs. The role of patterns for multiagent systems is discussed, and individual patterns are proposed to address agent concurrency, collaboration, and reasoning. The paper illustrates how to employ several patterns in concert to yield an architecture for multiagent systems, and clarifies how sets of objects can be employed to design agents, important software abstractions for pro- active behavior and distributed problem solving.

1.0 INTRODUCTION

The term agent has various meanings in the literature. In this paper, we adopt the definitions of [25]. A weak agent is i) autonomous ii) social iii) reactive and iv) pro- active, and a strong agent, in addition to the above, has one or more of the following: v) mentalistic notions (beliefs, goals, plans, and intentions) vi) rationality vii) veracity and viii) adaptability. Within autonomous behavior are the concepts of agent migration and transportability [23], and a social multiagent system must be open [1]. Agents are being used in a wide range of applications, including user interfaces [13], enterprise integration [17], manufacturing [11], network management [21], and workflow support [7]. While agents are widely used, a robust and maintainable architecture based on software engineering principles, such as object oriented design patterns, has not yet been formulated. Software reusability, in particular, is still an important problem in agent based systems [7].

Relationships and distinctions between agents and objects are discussed in [8,9]. An object is a software abstraction with behavior, inheritance, encapsulation, compound properties, state, and identity. A strong agent is an object that is able to reason and to collaborate via structured messages. Agents are also active objects, in that they have their own thread of control, and they are pro-active, which encompasses migration. An agent system aims to be more flexible and adaptive than an object system through autonomous and pro-active behavior. Part of the agent system's flexibility is in its openness, where agents can enter and/ or exit a society, and in its ability to carry out opportunistic problem solving through various techniques, including constraint based reasoning, plans, negotiation, and reasoning.

Several object oriented design patterns relevant to multiagent systems are detailed below. In particular, the design patterns for active objects, brokers, mediators, sensors, adapters, and interpreters address the design problems of agent concurrency, migration, negotiation, object- agent coexistence, and automated reasoning. These patterns and their applicability to multiagent systems are discussed in the following sections. The net result is a robust and maintainable software architecture for multiagent systems based upon proven object oriented designs.

2.0 OBJECT ORIENTED DESIGN PATTERNS FOR AGENTS

2.1 Aspects of Agency

Strong agents reason to select a capability or plan that could achieve their stated goal(s). Passive objects or object oriented databases should be used to represent agent knowledge to facilitate knowledge base maintenance [8]. Strong agent capabilities are stored in a plan library. A plan is instantiated when a triggering event occurs and an instantiated plan is an intention. An intention executes in its own thread, and an agent may have several intentions executing concurrently. Agents cooperate and negotiate with each other in conversations, and languages (such as KQML, COOL) [2], and AgenTalk [10] have been developed to support this. Migrating agents must be able to gain access to and share resources; they must also be able to enter and leave open systems.

These aspects of agency are summarized in Figure 1, and they are analogous to those presented elsewhere [4,7]. An agent is indicated, along with all the aspects of agency discussed above. The agent's reasoning is within its interpreter, beliefs, and plans. The agent's declarative knowledge (beliefs) may

be impacted by a sensor interacting with external objects. Three intentions are shown for illustration. One intention involves an effector object that then interacts with external objects. The other two intentions feature collaboration with other agents, either within the original society or external to it. Each of the agent's intentions has its own thread of control. Additionally, agent collaboration, and interaction with effectors and sensors, must consider issues of synchronous communication.

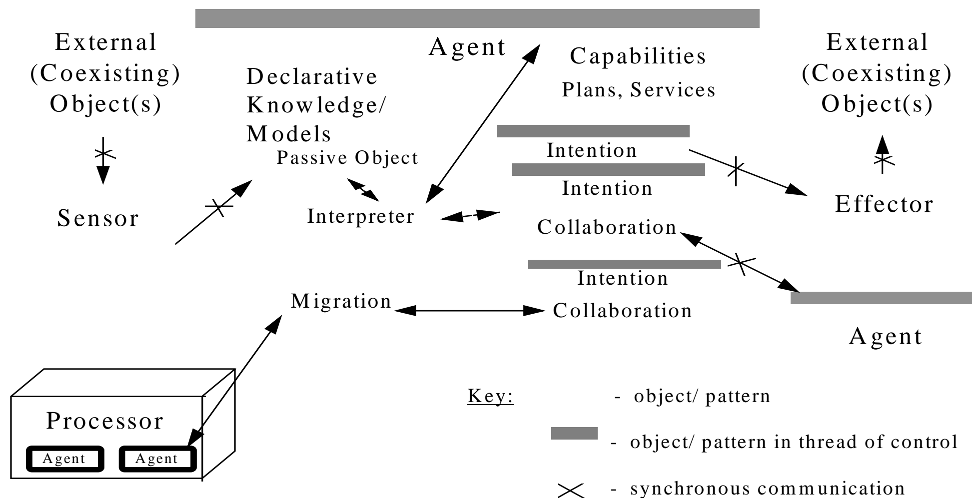

Figure 1: Aspects of Agency

2.2 Required Patterns

Modularity is achieved in object oriented patterns through the design of a team of well defined, cohesive objects that work together to solve a problem. Each object on the team has a clearly defined responsibility. In the pattern, interfaces between the objects minimize coupling. Team members that are of the same class category, or expected to play the same role in the team, must subscribe to the same interface so that they are interchangeable. Lastly, a pattern addresses object implementation, such as static relationships and any data type information. If C++ is the target language, some of the data types can be parameterized via templates.

A design pattern [5] has four essential aspects: i) *a name*, ii) *a problem and its context*, iii) *the solution* - the objects, their responsibilities, interfaces, and implementation, and iv) *results and tradeoffs*. The context for multiagent systems involves distributed problem solving and a need for pro-active, autonomous behavior where a user's direct intervention should not be required. Once this context has been stated, patterns for agent based systems can be identified by starting at item ii); what are the design problems involved in the aspects of agency described in section 2.1 ?

- *Interaction with Coexisting Objects:* A sensor must recognize the interface of external objects; it also interacts with agents via structured messaging. An effector must isolate an agent from platform or domain factors.
- *Collaboration with Other Agents:* Agents collaboration is done via structured messages, such as KQML, COOL, or AgenTalk. The collaboration may follow coordination protocols.
- *Synchronization and Concurrency:* Agents, intentions, and coexisting objects act in different threads of control and access shared resources. Agents may be shared resources.
- *Migration:* Agents must access each other and other resources in a way that is transparent to location.
- *Automated Reasoning*: An agent selects a service or plan on the basis of their knowledge.
- *Autonomic Behavior*: An agent may need to exhibit continuous response.

3.0 RELEVANT PATTERNS

3.1 Patterns for Synchronization and Concurrency

Figure 2 is a Booch [3] object diagram based upon the active object pattern [12] and the ACE concurrency encapsulation pattern [18]. These patterns address synchronization and concurrency. Four objects appear in the active object pattern: the Client Interface, the Scheduler, the Resource, and the Activation Queue. The Client Interface provides the interface to a client application. The invocation of a method triggers the construction of a Method Object which is then enqueued on the Activation Queue by the Scheduler. When the Resource is available, the Scheduler removes an activity from the queue and dispatches it to the Resource.

The additional two objects in Figure 2 are the Thread and the Guard. The Thread is a thread of control, and this object encapsulates any operating system specific behavior for thread creation, termination, and management. Thread is a parameterized class, parameterized by Resource Type, and the Resource's do() member function is what is executed in the Thread. The Resource may be shared; the Guard is responsible for managing shared or concurrent access to the resource by the (potentially) multiple Schedulers. The Guard is responsible for managing exclusive access to the resource, when state must be preserved throughout an interaction sequence. The Client Interface is responsible for other synchronous interactions.

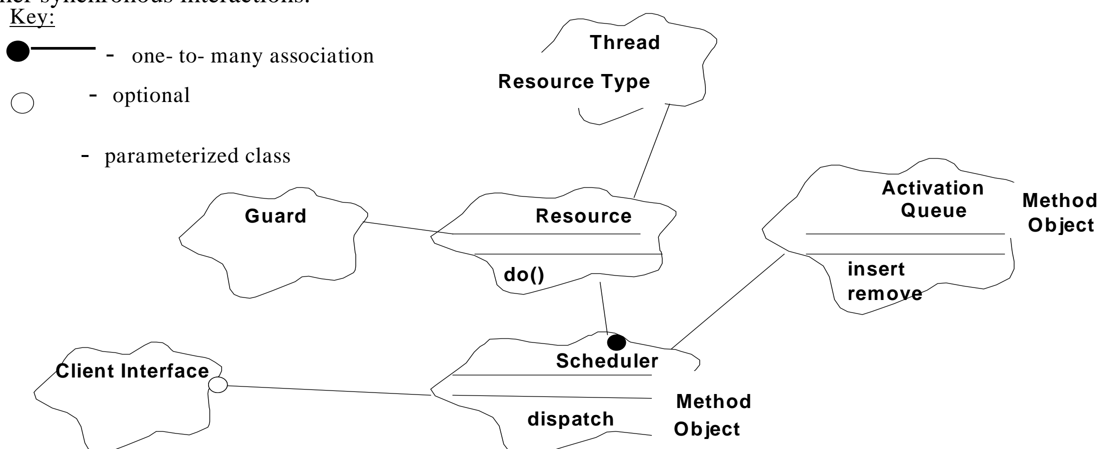

Figure 2: The Active Object Pattern with Thread and Guard Extensions

In the active object pattern, the Client Interface and the Guard decouple the interface *and the synchronization behavior* from the active object (Resource) itself. This is very significant in that it can be used to remove the inheritance anomaly [14, 18]. If an agent must support truly autonomic, concurrent behavior that is continuous, the resource may in fact have more than one thread of control. One or more private threads [6] may be added to the pattern in Figure 2 to support this.

3.2 Patterns for Interaction and Collaboration

3.2.1 The Mediator, Sensor and Adaptor Patterns

Agent collaboration and interaction with objects give rise to similar design problems. Whereas delegation can enhance reusability, proliferating interconnections increase complexity, complicate maintenance, and reduce reusability. The Mediator pattern [5] and its derivatives, Sensor and Adapter[15] address these issues and are proposed here for aspects of agent interaction and collaboration. The pattern is made up of a Colleague class category and two Mediator classes: an abstract Mediator class that defines an interface for communicating with colleagues, and a concrete Mediator class that implements the cooperative behavior by coordinating the colleagues. The Mediator pattern localizes collaborative behavior to one class. This is beneficial in that it replaces many- to- many interactions and removes the need for every colleague to have knowledge of every other colleague.

A Mediator- Colleague structure with only two Colleagues is an Adapter pattern [15]. The Adapter transforms messages being sent from one Colleague into a form that is comprehensible by the other Colleague. The Adapter pattern is applicable to the Effectors in Figure 1 that reformat the agent's actions into something that is recognizable by coexisting objects. The Sensor pattern [15] is relevant to the Sensors and the Effectors in Figure 1 in that the agents follow structured messaging but the objects typically do not. The roles of the Mediator, Adapter, and Sensor patterns in agent based systems are summarized in Figure 3.

There are other patterns that address collaboration and interaction with the same net result. The Adapter is a pattern in its own right (as opposed to the one discussed above which is a specialization of the Mediator)[5], with the purpose of translating incoming requests in one form into a form that is able to be understood by a domain object. With this Adapter, the agent becomes the client who sends messages to an Adapter. The Adapter actually translates the messages and passes them on to the Adaptee (coexisting object). Alternatively, the Effectors can also be viewed as examples of Wrappers[16] that encapsulate domain legacy objects and their details.

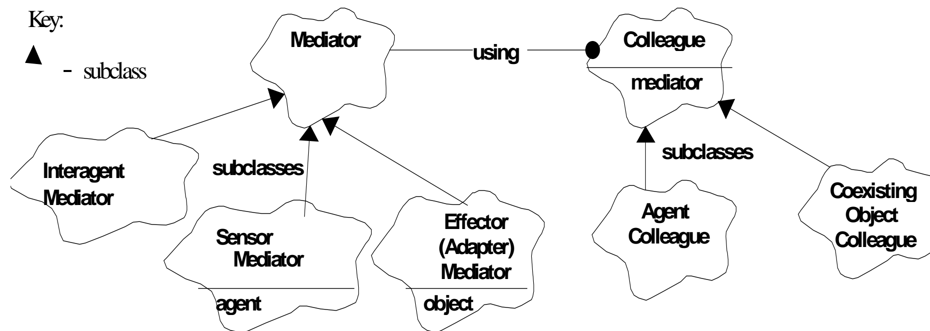

Figure 3: The Mediator, Sensor, and Adapter Patterns in Agent Based Systems

3.2.2 A Decentralized Collaboration Pattern

If communication bottlenecks become a problem, decentralized collaboration can be utilized. For this, each agent would require an instance of a Proxy for every interface that it supports, and it would have to maintain information or knowledge of every agent it wants to interact with. A Proxy [5] controls access to the Real Subject; it can also provide a distinct interface. The KQML Proxy would support the KQML interface, while the COOL Proxy and the AgenTalk Proxy would subscribe to the other interfaces, respectively. In addition to intercepting messages in a given agent dialect, the Proxy provides a layer of indirections and is responsible for determining if the message should be passed onto the agent (the Real Subject). For example, the agent may have already refused the request, or the request may be out of the agent's realm of interest.

For collaboration via coordination protocols the agent must be able to determine its behavior based upon the state of the coordination protocol [10] that it is engaged in. One agent may be engaged in several conversations simultaneously, requiring that it be able to accomplish context switching. The Memento pattern [5] externalizes an object's internal state so that the object can be restored to this state later. Agent proxies that support languages which involve conversations must store and recover their state. With the Memento pattern, they can delegate this to a companion Memento object. The Memento class is parameterized by Protocol, which gives the possible states and transitions. The resulting pattern for decentralized agent collaboration is shown in Figure 4.

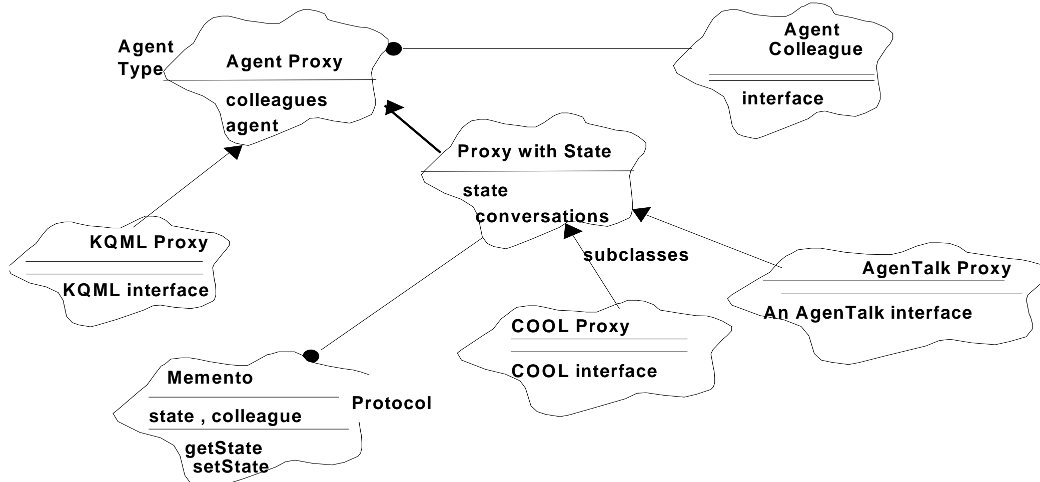

Figure 4: A Pattern for Decentralized Agent Collaboration

For the coordination protocols that follow a conversation or a speech act state machine, companion Memento objects are shown. The Proxy is the originator of the Memento, and a Memento is instantiated whenever a conversation is undertaken with another agent in the given language. The class State with Proxy therefore stores all conversations (instances of Memento) in a data member. When a collaboration is suspended, the Memento instance's state is set for storage so that when the conversation is subsequently resumed it can be recovered.

If Mediators were added to the pattern in Figure 4 for an architecure with both centralized and decentralized communication, they would be responsible for determining which colleagues need to cooperate. The proxies would be responsible for intercepting messages for the agent in a given language, while the mementos would store the conversations that the agent is presently engaged in so

that the proxy could recover them. As before, the benefit of employing this design or pattern is that agent behavior for structured message passing and conversing in finite state machine protocols is abstracted and delegated to component objects. Other behavior can be inherited, specialized, or delegated separately.

3.3 The Broker Pattern

Decentralizing access to an open society, or making every agent responsible for the security and interactions within its open society, leads to the same possibly N- to- N connections described in section 3.2.1. The Broker pattern[19], which is an abstraction of CORBA[22], provides for communication and location transparency for objects and/ or applications that wish to be clients and servers of one another. With this pattern, an agent can act as a client or a server to any registered agent or object. The agent (or its proxy) can become a virtual member of any open societies managed by the brokers. Bridges between societies are also supported.

In the Broker Pattern, agents who wish to be clients and servers for one another must use or employ a broker who is responsible for locating a server once a client has requested its services. Both the client and the server must have registered themselves with the broker. Both the server and the client employ proxies who respond to the interface known by the broker; the proxies may also be located in a different (the broker's) address space. As in section 3.3, these proxies support an interface that may not be known by the agent itself and provide a level of indirection to shield an agent from a request that should not be answered.

3.4 Patterns for Automated Reasoning

Two patterns that are relevant to agent automated reasoning are the Interpreter and the Strategy patterns [5]. The Interpreter pattern defines objects for the grammar of a language. Composite objects are used to represent the syntax and the statements to be interpreted, and an Iterator is used to traverse the structures. Interpreters relevant to agent oriented reasoning, such as one that interprets beliefs, goals, plans, context conditions and invocation conditions, could be designed via this pattern. Plans would be instantiated into intentions when the beliefs, context, and invocation conditions are satisfied. This would provide the autonomous, reactive, and pro-active behavior that is central to the definition of an agent with a knowledge based representation.

The Strategy pattern encapsulates or objectifies an algorithm or a strategy for solving a problem. With this approach, each plan in the plan library can be an object, and each plan has its own invocation and context conditions. The key to the pattern is that each stategy has the same interface; each plan and each intention can also have the same interface. The Interpreter and Strategy patterns represent object oriented designs for agent automated reasoning. Performance issues are crucial to agent automated reasoning, so the designs must be refined to address this.

3.5 Patterns for Database Access

Agents must work with legacy systems, such as databases. A persistent object stored in an object oriented database would be viewed by the agent as an external object. Relational database systems are far more complex. An agent must be able to transform records from the relational database into implementation objects and vice versa. The Objects from Records [24] is a pattern which facilitate the exchange of information between relational databases and object oriented agent clients.

In the Objects from Records pattern, there is an application object that works together with a Record object and a RecordMap object. The Record is responsible for getting, setting, and committing field objects to and from the database. A RecordMap defines the layout for all Records of the same kind; this layout is in turn detailed by FieldMaps. In general terms, the Objects from Records pattern is a more complicated form of an adapter or a wrapper. Although object- to- relational database conversion (and vice versa) is a complicated procedure, a wide variety of products are now available that provide object-oriented access to relational databases.

4.0 AN AGENT PATTERN

4.1 Pattern Layers

The patterns discussed in Section 3 must be integrated to formulate an agent pattern. As illustrated by Figure 1 and the various patterns, the internal structure of an agent can be quite complex. Each pattern dealt with a different aspect of the agent, and these different aspects of an agent make it sensible to use a layered pattern [20] for integration. Each layer will deal with one specific aspect and use the services of the layer directly below or above it. This layered approach is similar to the approach given for multiagent systems elsewhere [4,7], but more detail is provided here.

The stated context for the Layer pattern is *a large system needing decomposition*. The problem is further clarified as a system *with a mix of low and high level issues where high level issues rely on low level ones*. In this paper, the large agent system has been decomposed in section 3; low and high level issues have been detailed. Therefore, here the Layer pattern is used to bring order and structure to the decomposition. This is depicted in Figures 5 and 6 where an agent is shown to have four layers: Network, Collaboration, Reasoning, and Application Interface.

The Application Interface layer, Layer 1, encapsulates the agent's ability to collaborate and interact with external objects, applications, and databases within its own address space through the use of the Mediator, Sensor and Adapter patterns in Figure 3. (The Database Adapter shown in Figure 5 may in fact be an Objects from Records pattern if the database is relational.) The Reasoning layer, Layer 2, holds the Interpreter and Strategy patterns for automated reasoning and the instantiation of intentions in threads. Each thread becomes an active object (Figure 2). Layer 3, the Collaborative layer, models the agent's ability to communicate and coordinate with other agents through coordination protocols. Figure 4 links the communication and coordination functions through the use of the Mediator, Proxy and Memento patterns. The Network layer, Layer 4, provides the agent with the ability to physically transmit messages across threads and processes to other agents. It incorporates the Interagent Mediator and Broker patterns; if external objects are in other threads the relevant sensors and effectors are also in the network layer.

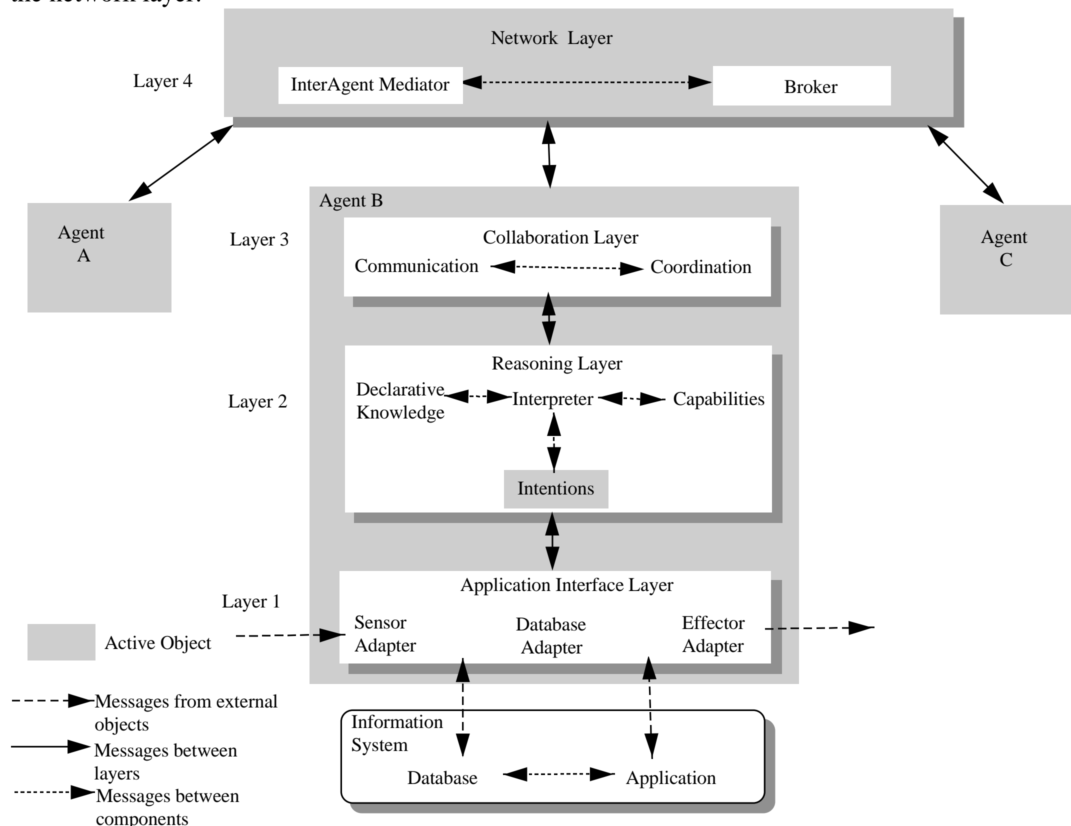

Figure 5: Layered Architecture for Agents

Figure 5 illustrates a top-down and bottom-top approach to communication. Each layer deals with a specific aspect and uses the services of the neighboring layers. A top-down request may be issued from Agent A to Agent B's broker. The broker checks whether Agent A is registered with Agent B. If it is, the broker passes the request to the server proxy which in turn passes it to the collaboration layer. The collaboration layer initiates conversation with Agent A. Depending on the request, Agent B may forward the request onto Agent C or seek the services of the reasoning layer to satisfy the request. The collaboration layer may send several requests to the reasoning layer. The reasoning layer may be able to satisfy the request or may seek the services of the API layer. From this example we can see that each layer has the ability to send requests down the line. Each layer may send one or many requests to a layer below.

A request for services may flow from lower to higher levels, a bottom-up approach. An agent's intention may be triggered by input through the sensor (the application and database work in a similar manner). The sensor may report a change in the external environment to the reasoning layer. The

reasoning layer may instantiate a plan to advise other agents, requiring the services of the collaboration layer. The reasoning layer may instantiate many intentions to satisfy a particular request.

As stated previously, each intention is within it's own thread of control. Each intention may require the services of other layers, for example for communication with an agent in its own or in another address space, or for affecting a coexisting object. Because each intention operates in its own thread, it requires the Resource and Thread aspects of the Active Object pattern (Figure 2). An intention does not get new tasks to perform; it also waits for any synchronous communication, suspending other activities. Therefore it does not require the Client Synchronization, Interface, Scheduler, or Guard components of the pattern.

The agent's own Active Object pattern is orthogonal to the layers in Figure 5, providing the mechanism or resource for all of the activities shown (except the intentions). That is, the network, collaboration, reasoning, and application interface layers all share the agent's thread of control, unless other, private threads are required for continuous activities. The agent's Resource object (Figure 2) has a do() member function that cycles through the needed activities: checking sensors and beliefs, receiving input from other agents, performing automated reasoning, instantiating intentions, and enacting effectors. It does this by cycling in turn through the layers; each layer is responsible for passing control on to the next layer.

4.2 The Pattern's Facades

A facade [5] provides a unified interface to a set of interfaces in a subsystem, and it will be used here for encapsulation and to provide a common interface for each layer. The double headed arrow in Figure 5 represents the flow of services and requests between layers using the facade object, and the approach is outlined in Figure 6. The facade for the Network layer is specialized by a Broker; this leads to associated Interagent Mediators who, once the Broker has made contact between client and server agents, handles direct communication. The facade for the Collaboration layer is specialized by the Client Synchronization Interface object from the Active Object pattern in Figure 2. It is responsible for managing all behavior that must synchronized with objects in other threads. The Synchronization Interface has many associated Proxies; it is also associated to the agent's Scheduler in the way indicated in Figure 2. This relationship, and the agent's Resource, Thread, and Guard objects are not repeated here for simplicity.

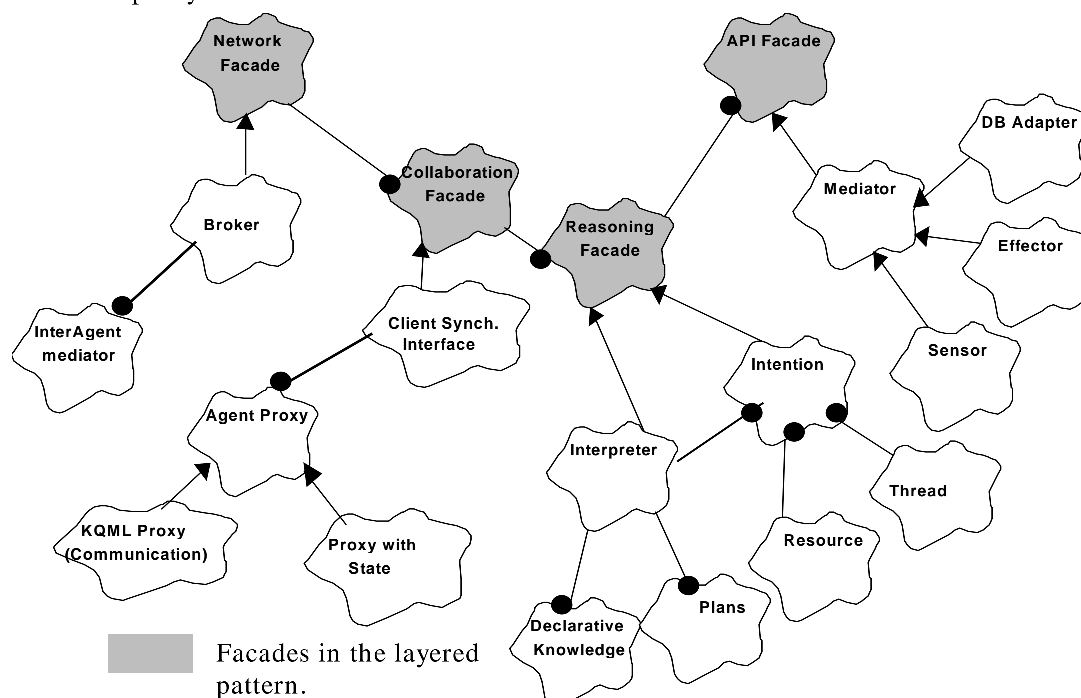

Figure 6: The Agent Pattern

Once the Collaboration layer is finished, it passes control to the Reasoning layer. This layer's facade can be specialized in two ways: to an intention or to the interpreter. The Interpreter may determine new tasks or intentions that need to be carried out, while an Intention, once activated, does not undertake anything new. Each intention may need to deal with the Collaboration layer or with the Application Interface Layer; hence it is appropriate that it is a Reasoning layer facade. The Application

layer facade is specialized to a Mediator, which is in turn specialized to Sensors and Effectors, or to Database Adapters.

The layered pattern provides an overall structure to a complicated system with many modules. Through this framework we are able to formulate the agent pattern in Figure 6. Following the diagram from left to right we can see the association between the layers, a 1:N relationship (shown by the solid circles). Each layer is encapsulated by a facade that is specialized to the objects or patterns described in section 3. The Network facade is responsible for behavior across threads or processes. The collaboration facade is responsible for the collaboration with other agents through an agent communication language using particular coordination protocols. The API facade decouples the support for external objects from the other layers, while the reasoning facade encapsulates both the interpreter and the intentions.

5.0 SUMMARY

In summary, an agent pattern or architecture has been proposed. It is based upon the active object, mediator, proxy, and broker patterns, with additional features for automated reasoning and thread encapsulation. The pattern has been described in terms of component objects with associations, messaging, class parameterization, and some inheritance. The layer and facade patterns have been used to structure the resulting decomposition. Some interfaces have been identified, but more work is needed. This should lead to further refinement of the pattern, especially in the areas of automated reasoning.

This agent architecture is in use at the Royal Melbourne Institute of Technology and the CRC for Intelligent Manufacturing Systems and Technologies in Australia to develop an agent based system for an application in discrete parts manufacturing. C++ implementations have been carried out, and the merits of the pattern have been demonstrated, particularly in the removal of the inheritance anomaly, the delegation of coordination and interaction behavior, and the centralized, CORBA compliant management of an open agent system.

6.0 ACKOWLEDGEMENTS

This project was partly funded by the Cooperative Research Centers for Intelligent Decision Systems and for Intelligent Manufacturing Systems and Technologies under the Australian Government's CRC Program.

7.0 REFERENCES

1. Avouris, N. M., and L. Gasser, *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer Academic Publishers, 1991.
2. Barbuceanu, M. and M. S. Fox, "COOL: A Language for Describing Coordination in Multi-Agent Systems," First International Conference on Multi- Agent Systems, pp. 17 - 24, 1995.
3. Booch, G., "Object Oriented Analysis and Design with Applications, Second Edition", The Benjamin/ Cummings Publishing Company, 1994.
4. Farhoodi, F., I. Graham, "A Practical Approach to Designing and Building Intelligent Software Agents", First International Conference on the Practical Application of Intelligent Agents and Multiagent Technology, London, 1996.
5. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison- Wesley, 1994.
6. Gilbert, J. W., "A Software Pattern for the Implementation of Autonomic Object Behavior," Workshop on Patterns for Concurrent, Parallel, and Distributed Object- Oriented Systems at OOPSLA '95, Austin, Texas, 1995.
7. Jennings, N. R., P. Faratin, M. Johnson, P. O'Brien, and M. Wiegand, "Using Intelligent Agents to Manage Business Process", First International Conference on the Practical Application of Intelligent Agents and Multiagent Technology, London, 1996.
8. Kendall, E. A., M. Malkoun, and C.H. Jiang, "A Methodology for Developing Agent Based Systems for Enterprise Integration", EI'95, IFIP TC5 SIG Working Conference on Models and Methodologies for Enterprise Integration, Heron Island, Australia, November, 1995a.
9. Kendall, E. A., M. Malkoun, and C.H. Jiang, "A Methodology for Developing Agent Based Systems", First Australian Workshop on Distributed Artificial Intelligence at the Eighth Australian Joint Conference on Artificial Intelligence (AI'95), Canberra, Australia, November, 1995b.
10. Kuwabara, K., T. Ishida, and N. Osato, "AgenTalk: Describing Multiagent Coordination Protocols with Inheritance," submitted to Tools for Artificial Intelligence Conference, 1995.

11. Kwok, A., and D. Norrie, “Intelligent Agent Systems for Manufacturing Applications”, J. of Intelligent Manufacturing, Vol 4, pp. 285- 293, 1993.
12. Lavender, R. G., and D. C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming”, Pattern lagnuages of Programming Conference, Illinois, 1995.
13. Maes, P., “Agents that Reduce Work and Information Overload,” Comm. of the ACM, Vol 37, No. 7, 1994.
14. Matsuoka, S., and A. Yonezawa, "Analysis on Inheritance Anomaly in Object-Oriented Concurrent Programming Languages", in *Research Directions in Concurrent Object-Oriented Programming* (ed. Gull Agha, P. Wegner, and A. Yonezawa), The MIT Press, Cambridge, MA., 1993.
15. Meunier, R., “The Pipes and Filters Architecture,” in *Pattern Languages of Program Design*, ed by J. Coplien and D. Schmidt, Addison- Wesley, 1995.
16. Mularz, D., “Pattern- Based Integration Architectures,” in *Pattern Languages of Program Design*, ed by J. Coplien and D. Schmidt, Addison- Wesley, 1995.
17. Pan, J. Y. C., and J. M. Tenenbaum, “An Intelligent Agent Framework for Enterprise Integration,” IEEE Trans. on Systems, Man, and Cybernetics, Vol. 21, Nol. 6, November/ December, 1991.
18. Schmidt, D. C., “The ACE Object- Oriented Encapsulation of Lightweight Concurrency Mechanisms”, 1995.
19. Schmidt, D. C., and S. Vinoski, “Object Interconnections: Comparing Alternative Client- Side Distributed Programming Techniques (Columns 3)”, SIGS C++ Report Magazine, May, 1995.
20. Siemens AG, *The Broker Pattern*, ftp site pub/patterns/siemens/broker.A4.ps, November, 1995.
21. Siemens AG, *The Layered Pattern*, ftp site pub/patterns/siemens/layers.A4.ps, May, 1996.
22. Somers, F., “Intelligent Agents for High- Speed Network Management, “ PAAM- 96, First International Conference on the Practical Application of Intelligent Agents and Multi- Agent Technology, London, 1996.
23. Vinoski, S., “Distributed Object Computing with CORBA”, C++ Report, Vol. 5, July/ August 1993.
24. Wayner, P., *Agents Unleashed: A Public Domain Look at Agent Technology*, AP Professional, 1995.
25. Wolf, K., and C. Liu, “New Clients with Old Server: A Pattern Language for Client/ Server Frameworks,” in *Pattern Languages of Program Design*, ed by J. Coplien and D. Schmidt, Addison- Wesley, 1995.
26. Wooldridge, M. J., and N. R. Jennings, “Agent Theories, Architectures, and Languages”, Tutorial, First International Conference on Multi- Agent Systems, 1995.