IC-AI'99

502SA

Patterns as a Means for Intelligent Software Engineering

Patterns as a Means for Intelligent Software Engineering

D. Deugo, F. Oppacher, J. Kuester[*], I. Von Otte[*]
School of Computer Science,
Carleton University
1125 Colonel By Drive,
Ottawa, Ontario, Canada, K1S 5B6

Abstract - *In this paper, we make a case for the development of intelligent software engineering patterns. Patterns have proven extremely useful to the object-oriented programming community. However, of the large amount of pattern research, little effort has been devoted to developing intelligent software engineering patterns. We wish to correct this situation. We believe, for example, that the ongoing success of agent systems depends on the development of sound software engineering principles for them. Patterns are a recognized means to this end, and one that we wish to promote.*

Keywords: Communication, Agent, Pattern

1 Introduction

The objective of software engineering is to produce software products [11] which are systems delivered to customers with documentation that describes how to install and use them. Therefore, the objective of intelligent software engineering must be to produce complex and intelligent software products. We can use Artificial Intelligence (AI) techniques to help build these systems, but intelligent software engineering is mainly concerned with how to incorporate these techniques into the resulting systems.

As research progresses in an area, certain research elements (e.g., principles, facts, fundamental concepts, techniques, algorithms and architectures) become well understood. As an indicator of this, take for example the call for papers for the workshop on Mobile Agents in the Context of Competition and Cooperation [10]. We find comments such as, "… gaining more widespread acceptance and recognition as a useful abstraction and technology" and "we are uninterested in papers that describe yet another mobile agent system." Since many mobile agent systems now exist, we believe the program committee felt that there was no need to see others describing similar approaches. The question to answer at this point is, what should the research community do next with the knowledge it has gained?

We propose the use of patterns as the principal tools for intelligent software engineering. This is not a matter of only documenting the solution and problem surrounding each research element; this material already exists in many papers. One must go further and deeper with their explanations, identifying the forces and the contexts of the problems that give rise to the proposed solutions. These are the undocumented and often misunderstood features of the research elements, and we believe they must be exposed before the corresponding software engineering principles can be incorporated into business applications. For example, we have lost track of how many times we have read a paper that indicated that it used KQML [6] for the communication between agents and not been able to understand why it was used? There may have been an obvious advantage to using it, or maybe it did not matter. What we wanted to understand was which forces and context lead to this decision, because if we need to make a similar decision in the future we need this information.

Our objective in this paper is to introduce the notion of intelligent software engineering patterns. Since not many are familiar with

[*] Originally from the University of Paderborn, funded by DAAD, the German Academic Exchange Service, while at Carleton.

software patterns, and those that are often think of them as only problem and solution pairs, we begin with an introduction to patterns and pattern languages. Next, we strengthen our argument for the importance of patterns for intelligent software engineering. Finally, as an example of intelligent software engineering patterns, we provide several of them dealing with mobile agent communication. Although we describe only abbreviated patterns due to space limitations, our goal is to provide the reader with enough information to apply them immediately and to convince the reader to build their own intelligent software engineering patterns in the future.

2 Patterns and Pattern Languages

Software patterns have their roots in Christopher Alexander's work in the field of building architecture. After reading his work, it was clear to software engineers that, like building designs, there are many recurring problems and solutions used in the design of software systems. Unfortunately, they noted that many of these combinations were hard to find except for in the minds of the most experienced developers, for if they were, projects would have been built on time, within budget and without bugs! Moreover, knowing the problem and solution were not enough. Software engineers needed to know when the solutions were appropriate for the given problems.

In Alexander's book, "The Timeless Way of Building" [1], we find the following definition for a pattern:

- A pattern is a three-part rule that expresses a relation between a certain context, a problem, and a solution.
- Each pattern is a relationship between a certain context, a certain system of forces that occurs repeatedly in that context, and a certain spatial configuration that allows these forces to resolve themselves.
- A pattern is an instruction, which shows how the spatial configuration can be used, repeatedly, to resolve the given system of forces, wherever the context makes it relevant.
- The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we should create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate the thing

Although there are many formats for patterns, the minimal format contains the following, or essentially similar, headings.

- **Name:** As the saying goes in the object-oriented community, a good variable name is worth a thousand words and a good pattern name, although just a short phrase, should contain more information than just the number of words would suggest. Would the word 'agent' be a good pattern name? The answer is no. Although it means a lot more than just the single word would suggest, it has too many meanings! One should strive for a short phrase that still says it all.
- **Problem:** A precise statement of the problem to be solved. Think of the perspective of a software engineer asking himself, how do I ….? A good problem for a pattern is one that software engineers will ask themselves often.
- **Context:** A description of a situation when the pattern might apply. However, in itself the context is not the only factor determining the situations in which the pattern should be applied. Every pattern will have a number of forces that must be balanced before applying it. The context helps one determine the impact of the forces.
- **Forces:** A description of an item that influences or constrains the decision as to when to apply the pattern in a context. Forces reveal the intricacies of a problem and can be thought of as items that push or pull the problem towards different solutions or indicate trade-offs that might be made [5]. A good pattern description should fully encapsulate all the forces that have an impact upon it.
- **Solution:** A description of the solution in the context that balances the forces.

Other sections such as rationale, resulting context, implementation, sample code, known uses and related patterns are included to help with the description.

A good pattern provides more than just the details of these sections; it should also be generative. Patterns are not solutions; Patterns generate solutions. You take your 'design problem' and look for a pattern to apply in order to create the solution. The greater the potential for application of a pattern, the more generative it is. Although specific patterns are useful, a great pattern has many different applications. For this to happen, pattern writers spend considerable time and effort attempting to understand all aspects of their patterns and the relationships between those aspects. This generative quality is so difficult to describe that Alexander calls it "the quality without a name", but you will know a pattern that has it once you read it. It is often a matter of simplicity in the face of complexity.

Although useful at solving specific design problems, you can enhance the generative quality of patterns by assembling related ones to form a pattern language, enabling you to build software frameworks or families of related systems. A pattern language's collection of patterns forms a vocabulary for understanding and communicating a "whole", revealing the structures and relationships of its parts that fulfill a shared objective.

For example, individual patterns might help you design a specific aspect of a mobile agent, such as how it models beliefs, but a pattern language might be able to help you build all types of agents.

Intelligent software engineering pattern languages are very important for the patterns to be successful. Forcing each pattern to identify its position within the space of existing patterns is not only good practice, it is also good research. In other words, all intelligent software engineering patterns should be part of an intelligent software engineering pattern language. It is not only helpful to you, but to all other software engineers who will use the patterns to develop their systems in the future.

3 Patterns are Important for Intelligent Software Engineering

For any software system to be successful and run safely, it must be constructed using sound software engineering principles, and not constructed in an ad-hoc fashion. Unfortunately, much of intelligent software development to date, especially involving agents, has been done ad hoc [4], creating many problems – the first three noted by [8]:

1. Lack of agreed definitions
2. Duplicated effort
3. Inability to satisfy industrial strength requirements
4. Difficulty identifying and specifying common abstractions above the level of single agents
5. Lack of common vocabulary
6. Complexity
7. Only goals and solutions presented

These problems limit the extent to which “industrial applications” can be built using AI technology, as the building blocks have yet to be exposed or defined for the software engineers. Objects and their associated patterns have provided an important shift in the way we develop applications today, since the level of abstraction is greater than procedural or data abstraction. We find it essential to begin an effort to document AI’s abstractions so that others can share in the vision. Patterns provide a good means of documenting these building blocks in a format already accepted by the software engineering community. Patterns also have the added benefit that no unusual skills, language features, or other tricks are needed to benefit from them [9].

4 Communication Patterns for Mobile Agents

A mobile agent framework consists of places and agents. A place is a location on node in a network and provides an environment for agents to execute. Several agents can be located at the same place.

You can classify agents as static, service agents or as mobile agents. Service agents live in the same environment as mobile agents, but do not change places. Mobile agents move to different places to accomplish their tasks. The mobility of agents can vary from few to many changes in location. Agents can pursue their own goals or can work together and collaborate with other agents.

The differences in mobility have created the need for several patterns to manage the communication between agents.

4.1 Direct Coupling Pattern

Problem: How do mobile agents communicate directly with others in a mobile agent system to accomplish a task?

Context: You are developing an application incorporating mobile agents.

Forces:

- Mobile agents change their locations frequently.
- Agent communication takes place between agents on a peer-to-peer basis.
- A robust and efficient communication mechanism is needed that allows communication between static and mobile agents.
- A simple solution where each agent is assigned an address is obviously not applicable because this address becomes invalid as soon as the agent moves to another place. A more sophisticated approach is needed that allows agents to change their places and maintain communication with their communication partners.

Solution: Since every agent needs to know the positions of its colleagues, establish and maintain a tight coupling between communicating agents using a modified Observer Pattern [7]. Have each agent register with the others it wants to communicate with and when those agents change their positions have them notify the registered agent of the change. Therefore, the registered agent becomes the observer of the others. If the communication is bidirectional, each agent is an observer as well as observed agent. In contrast to the original Observer pattern, the observed agent sends its new position in addition to a general notification event.

Resulting Context: By applying this pattern, there is a tight coupling between agents, enabling them to communicate directly with one other. IBM's Aglets [9] environment, for example, supports this mobility notification by default in every agent. The Aglet environment does not support automatic bookkeeping of agent locations, and therefore provides less functionality than the pattern. For small systems, this technique works well, especially when agents do not have many communication partners or do not change their locations frequently. However, in a system where one agent has many potential communication partners and moves frequently, Direct Coupling results in an increase in network message traffic. This increase creates a potential bottleneck because an agent will need to send notification messages every time it changes its position.

4.2 Proxy Agent Pattern

Problem: If the agent sending a message does not expect the receiving agent to move or the Directly Coupling pattern is not applicable for performance reasons, how do mobile agents communicate with others?

Context: You tried applying the Direct Coupling Pattern, but found that it resulted in a bottleneck due to the increased network message traffic. In addition, there are now situations when agents must always be reachable at the same address. You have also made a design decision to separate the task of handling agent mobility from the service provider agent in the mobile agent's home place.

Forces:

- Mobile agents change their positions frequently but each has a home place.
- Agent communication takes place between many agents on a peer-to-peer basis.
- A robust and efficient communication mechanism is needed that allows communication between static and mobile agents. Static agents are easy to find, because they always 'live' at the same place.
- Communication occurs arbitrarily between many agents.

Solution: Use a modified Proxy Pattern [5]. When an agent moves away from one place, it creates a Proxy Agent at its home location to hide its change of position. When this Proxy Agent receives a message, it can either forward the message to the corresponding agent, or store the message and deliver it later on request. The former can be called an Active Proxy Agent, the latter a Passive Proxy Agent. You can view an Active Proxy Agent as message forwarding whereas the Passive Proxy Agent resembles a mailbox concept.

Resulting Context: If the Proxy Agent is active, it needs to keep track of the location of the original agent. Therefore it has to become the observer of the original agent and the original agent needs to notify the Proxy Agent when it changes location. An advantage of the active solution is that it ensures prompt message delivery. A disadvantage is the need for registering and deregistering the original agent. If messages arrive infrequently and the original agent is highly mobile, the overhead of the registering and deregistering outweighs the advantage of prompt message delivery.

If the Proxy Agent is passive, it does not need to keep track of the location of the original agent, but it must be able to store the messages. Since message delivery takes place only on request of the original agent, this particular solution is not applicable if message delivery has to be synchronous. However, in some cases a Passive Proxy Agent might be better than an active one, especially if messages are sent regularly and the original agent changes its location often.

By applying the Proxy Agent Pattern, you decouple the original agent from its communication partners. This results in less message overhead because only the Proxy Agent has to be notified of a move by the original agent.

One major advantage of the Proxy Agent Pattern is transparency: the communicating agents do not know whether they are communicating with a Proxy Agent or the 'real' agent. This results in code that is more flexible because agents are not dependent on agent types. The main disadvantage is the extra level of message indirection, which can cause additional message traffic.

4.3 Communication Sessions Pattern

Problem: How do mobile (or service) agents manage complex communication with others in an easy and efficient way?

Context: You are developing a mobile agent system in which mobile agents interact with either mobile or static agents in complex conversations occurring over a period of time. The typical scenario is that two agents agree to communicate, interact by exchanging messages, and finally stop their conversation.

Forces:

- Mobile agents change their position frequently.
- Messages of different conversations arrive interleaved in a random order.
- Complex interactions involve many messages.
- The design should be simple and easy to use
- The design should restrict the agents as little as possible.
- The communication should be as efficient as possible.
- You need to make it possible for agents to have simultaneously conversations and manage these conversations in a simple and efficient way.

Solution: Introduce the concept of a session. A session is an open communication link between two agents that is represented by a session object in each agent [3]. Examples of this type of communication include message passing, RMI or sockets.

The use of sessions allows agents to manage several concerns. First, conversations are separate. Agents receive messages from a specific session object and can assign a handler to each. In this handler, the current state of the conversation can be encapsulated. Moreover, an open session indicates an ongoing interaction with a corresponding agent.

Second, it is possible to separate session management from the mobility problem. We can force an agent not to change its location in a session, or that such a change implicitly closes all open sessions. This allows a direct and simple communication between agents, which results in less overhead and higher performance. If mobility during ongoing conversations is required, a proxy object can be used.

Resulting Context: You can establish sessions in either an active or passive manner. An active setup blocks the calling agent until the communication link is established. This is useful in peer to peer communication. If an agent is offering a service and may accept a large number of communicating agents, a passive setup can be used. This results in a client-server interaction. Every time, an agent performs an active setup, a new session and a new thread, handling the session, is created.

If conversations become very complex, it might be useful to define a formal conversation plan [2]. A conversation plan is a finite automaton. Automata states represent states of the conversation, and transitions correspond to incoming messages. The purpose of a conversation plan is to make the structure of a conversation explicit and therefore easy to maintain and understand. The structure is similar to the State pattern [7].

4.4 Badges Pattern

Problem: How can an agent find a suitable communicating agent without specifying a concrete agent?

Context: A group of agents that satisfy certain properties needs to collaborate and communicate with others in the same place. For example, a network mapping agent needs to exchange data with other mapping agents at its current place and needs to find the addresses of the mapping agents currently located at the same place.

Forces:

- Communicating agents are not known in advance and can change.
- Partner selection is permitted based on a specific agent type.
- The design should be flexible, and as general as possible.
- Communication should be direct after communicating agents have been identified.

Solution: Attach badges to agents. Give every badge a unique id, and have every agent carry a set of badges. Badges can be pinned on and off an agent. A place provides a service to find a local agent carrying a certain badge.
Agents belonging to the same group can carry the same badge. If an agent wants to search for another in its group at the current place, it uses the place's badge service and requests an agent carrying a specified badge. If no agent exists, the request fails, otherwise the id of a random agent from the set of suitable agents is returned. After given a badge, an agent can establish communication with another directly.

Resulting Context: We can generalize this concept, if we allow logical expressions as queries to the place's badge service. It is then possible to express queries like SEARCH_AGENT AND HASVALIDSOLUTION. With this mechanism it is easy to request agents in a specific state, or to search for agents of a specific subgroup. The badge service is easy to implement because a place is notified when an agent enters or leaves the place. The service can request the badges from every incoming agent, and maintain a directory that maps from badges to sets of agents carrying the badge. If an agent leaves, its badge is removed from these sets. The structure of this pattern is an extension of the Broker pattern, specialized to the specific needs of the agent context.

4.5 Event Dispatcher Pattern

Problem: In a mobile agent framework there is often the need for efficient communication between an agent and a group of other agents. Usually communication is unidirectional, like "fire and forget" messages. How do agents achieve this form of communication in a mobile agent framework?

Context: You are developing an application where agents need to communicate with groups of other agents without knowing the actual members of the groups. Communication is unidirectional and is therefore considered an event or notification. An example for this kind of communication is a dynamically changing group of agents working on the same problem and when one has found a solution, it wants to send a terminate notification to all other agents.

Forces:

- Agents are mobile.
- The members of a group might change over time.
- A simple solution is desired.

Solution: Introduce an intermediate event dispatcher object between the sending and receiving agents. This concept is known and used in the Observer Pattern. Every agent interested in a type of event, registers itself as an observer at the event dispatcher object. An agent originating an event sends the message to the event dispatcher. The dispatcher then sends the message to all registered agents.

Resulting Context: Although the dispatcher has to cope with the mobility of the receiving agents, it can require that agents are static to simplify the implementation. If an agent wants to move while expecting events, they can use either an active or passive proxy agent.

Instead of simple multicasting an event to all receiving agents, it is possible to build more complex coordination objects applying the same principle. One example is an AND-Group. Incoming events are boolean values. The result of the operation is TRUE, if all events (the number has to be fixed) are TRUE, and FALSE otherwise. If one agent sends a FALSE event, the coordination object can send a FALSE message to every receiver immediately. If all agents send a TRUE event, then it can send a TRUE event to all receivers.

5 Conclusion

Although not presented as a pattern language, the next step for us is to reveal the structures and relationships of our patterns for the shared objective of agent communication.

What is in the future of intelligent software engineering pattern research? First, we must develop an initial set of pattern classifications to help focus pattern writers on the targets that are of the greatest importance to those developing ‘real’ intelligent software systems. Second, within each classification, we must identify pattern languages for pattern writers to develop and extend, and force them to position their new patterns within the space of existing patterns. Third, we must constantly remind those involved with artificial intelligence research to not only describe their solutions, but to also think, discuss, and write about the problems their solutions are intended to address and what context and forces led them to a particular solution. Fourth, we must write the patterns.

In short, we believe that following this approach, we will not have to read about “yet another intelligent system framework” anymore. Rather, we will be able to read and understand what problems an intelligent system or framework solves for us and when we should consider using the approach!

6 References

[1] C. Alexander. The Timeless Way of Building. New York: Oxford University Press, 1977.

[2] M. Barbuceanum and M. Fox. Integrating Communicative Action, Conversations and Decision Theory to Coordinate Agents. In K. Rothermel and R. Popescu-Zeletin (eds.), 1st Int. Workshop on Mobile Agents (MA'97) LNCS 1219, Springer-Verlag, 1977.

[3] J. Baumann, F. Hohl, N. Radouniklis, M. Strasser, K. Rothermel. Communication Concepts for Mobile Agent Systems. In K. Rothermel and R. Popescu-Zeletin (eds.), 1st Int. Workshop on Mobile Agents (MA'97) LNCS 1219, Springer-Verlag, 123-135, 1977.

[4] J. Bradshaw, S. Dutfield, P. Benoit, J.D. Woolley. KaoS: Towards and Industrial-Strength Open Distributed Agent Architecture. In J.M. Bradshaw (Ed.), Software Agents, AAAI/MIT Press, 1997.

[5] J.O. Coplien. Software Patterns. SIGS Management Briefings Series, SIGS Books & Multimedia, 1996.

[6] T. Finin, Y. Labrou, J. Mayfield. KQML as an Agent Communication Language. In J.M. Bradshaw (ed.) Software Agents, AAAI Press/MIT Press, 291-316, 1997.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[8] E. Kendall, P.V. Murali Krishna, Chirag V. Pathak, C.B. Suresh. Patterns of Intelligent and Mobile Agents. Autonomous Agents '98 (Agents '98), 1998.

[9] D.B. Lange, and M. Oshima. Programming and Deploying Java Mobile Agents with Aglets. Addison Wesley, 1998.

[10] MAC3. Mobile Agents in the Context of Competition and Cooperation. Autonomous Agents ’99, http://mobility.lboro.ac.uk/MAC3, 1999.

[11] I. Sommerville. Software Engineering. Addison-Wesley, 1996.