INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

AGENT POLICY PATTERNS

By
Gordon Ping Gu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Science, Information and Systems Science

School of Computer Science
Carleton University
Ottawa, Ontario
28th June, 2000
© copyright
2000, Gordon Ping Gu

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-57771-6

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

AGENT POLICY PATTERNS

Submitted by
Gordon Ping Gu

In partial fulfillment of the requirements for the degree of
Master of Science, Information and Systems Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

8th September, 2000

AGENT POLICY PATTERNS

ABSTRACT

This thesis documents six agent policy patterns that are extracted from various available agent frameworks. These patterns are Agent Abstraction Pattern, Use Service Pattern, Service-Based Implementation Pattern, Layered Structure Pattern, Agent Policy Delegation Pattern and Hashtable Pattern. These patterns provide a systematical way to structure the policy implementations for multi-agent frameworks and allow agent developers to achieve good encapsulation, hiding the base agent implementation details from other agent developers.

This thesis also introduced an agent policy pattern language that binds the existing agent patterns, i.e., Agent Pattern and Agent Layer Structure Pattern, and the new agent policy patterns. It also illustrates the use of the pattern language in the design of a distributed agent environment for agent-based applications.

CONTENTS

Abstract.....	i
Contents.....	ii
List of Figures.....	vi
List of Table.....	vii
Acknowledgements.....	ix
Chapter 1: Introduction.....	1
1.1. Problem.....	1
1.2. Motivation.....	2
1.3. Approach.....	3
1.4. Objectives.....	4
1.5. Outlines of this Work.....	4
Chapter 2: The agent Paradigm.....	6
2.1. What is an agent.....	6
2.2. Agent Characteristics.....	7
2.3. Supports of Agency.....	10
2.4. The Notion of Agent.....	11
2.4.1. The notion of a weak agent.....	11
2.4.2. The notion of a strong agent.....	12
2.5. Agency and Agent Intelligence.....	13
2.6. Categorizing of Agents.....	14
2.7. Multi-Agent Systems.....	17
2.8. Summary.....	21

Chapter 3: software Pattern.....	22
3.1. What is a pattern?.....	22
3.2. A brief history of patterns.....	23
3.3. Pattern Descriptions.....	24
3.4. Pattern Formats.....	25
3.4.1. GOF Format.....	26
3.4.2. Canonical Format.....	27
3.5. Pattern language.....	30
3.6. Agent Patterns.....	31
3.6.1. Multi-agent System Pattern.....	32
3.6.2. Agent Architectural Pattern.....	34
3.6.3. Agent Activity Pattern.....	37
3.7. Agent Policy Pattern.....	38
3.8. Summary.....	39
Chapter 4: Agent Policy Patterns.....	40
4.1. What is Policy?.....	40
4.2. Agent Policy Pattern language.....	44
4.3. Agent Policy Pattern.....	46
4.3.1. Agent abstraction pattern.....	46
4.3.2. Use Service Pattern.....	57
4.3.3. Service Base Implementation Pattern.....	65
4.3.4. Layered Structure Pattern.....	70

4.3.5. Agent Policy Delegation Pattern.....	76
4.3.6. Hashtable Pattern.....	81
4.4. Summary.....	86
Chapter 5: Agent Policy Pattern Application	87
5.1. Introduction.....	87
5.2. Requirements.....	88
5.3. Scenario.....	89
5.4. High-level Interaction Diagram.....	90
5.5. Class Diagram OF DBOS.....	92
5.5.1. GUI.....	92
5.5.2. Book Ordering Broker.....	93
5.5.3. Agent Database Servers (Bookstores).....	93
5.6. Design Approach Using Policy Pattern Language.....	94
5.7. The Agent Pattern.....	96
5.8. Agent Sub-System--Layer Structure and Agent Abstraction Patterns.....	99
5.8.1. Layer Structure Patterns.....	100
5.8.2. Agent Abstraction Pattern.....	101
5.9. Agent Context Sub-System -- Use Service Pattern.....	101
5.10. Service Sub-System -- Service Based Implementation Patterns.....	102
5.11. Detail Implementation of Agent Policy.....	104
5.11.1. Agent Policy Delegation Pattern.....	104
5.11.2. Hashtable Pattern.....	105
5.12. Summary.....	106

Chapter 6: Conclusion.....	107
6.1. Contribution.....	107
6.2. Future Work.....	108
References.....	109

LIST OF FIGURES

Figure 3.1: GUI pattern language	31
Figure 3.2: Global view of an agent-based application.....	32
Figure 3.3. Generic structure of the Agent pattern – class diagram.....	33
Figure 3.4: Agent Layer Pattern.....	34
Figure 4.1: An overall relation between agent policy patterns	45
Figure 4.2: Agent system contains context and base agent.....	46
Figure 4.3: A direct inheritance from the base agent.....:	48
Figure 4.4: An indirect inheritance from the base agent.....	49
Figure 4.5: An association between a base agent and a proxy agent	50
Figure 4.6: A combination of the abstract and proxy patterns.....	51
Figure 4.7: A scenario of location transparent with the Proxy agent.....	54
Figure 4.8: Class diagram of agent/context association.....	60
Figure 4.9: A sequence diagram describing how an agent requires a service via the agent context	60
Figure 4.10: A simple policy pool implementation	67
Figure 4.11: A class diagram of a service-based implementation pattern	68
Figure 4.12: A simple agent - policy pool association	72
Figure 4.13: An agent-policy pools structure	73
Figure 4.14: A layered Structure of policy pool structure in an agent	74
Figure 4.15: A separated policy conflict-solving structure	75
Figure 4.16: An association between the LayerObject and PolicyPool class	78

Figure 4.17: A class diagram of the Agent Policy Delegation Pattern	79
Figure 4.18: A class diagram for the Hashtable Pattern	83
Figure 4.19: A sequence diagram for the Hashtable Pattern.....	84
Figure 5.1: The architecture of the DBOS.....	87
Figure 5.2: Layer structure of application-agent framework.....	88
Figure 5.3: High-level interaction diagram of DBOS.....	90
Figure 5.4: A class diagram of DBOS.....	91
Figure 5.5: Class diagram of GUI module.....	92
Figure 5.6: Class diagram of Broker module.....	93
Figure 5.7: Class diagram of server module.....	94
Figure 5.8: Design of the distributed agent environment using pattern language...	95
Figure 5.9: A high-level design of the distributed agent environment using Agent Pattern.....	97
Figure 5.10: A class diagram of the distributed agent environment.....	99
Figure 5.11: The use of Layer Structure Pattern in base agent design	100
Figure 5.12: A use of the Agent Abstraction Pattern in generic agent design	101
Figure 5.13: Use of Service-Based Implementation Pattern	104
Figure 5.14: Use of the Agent Policy Pattern.....	105
Figure 5.15: The Hashtable Pattern.....	106

LIST OF TABLE

Figure 4.1: An overall relation between agent policy pattern.....	45
---	----

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my inexpressible gratitude to my supervisor, Professor Dwight Deugo, for his technical guidance, enthusiastic support and constant patience during the course of this thesis work. I would also like to thank my wife, Samantha Liu, and my parents for their constant supports and encouragement.

AGENT POLICY PATTERNS

CHAPTER 1: INTRODUCTION

1.1. PROBLEM

The exponential growth of the Internet makes the world instantly more connected. It has been estimated that there were over 30 million Internet users spread across 96 different countries in 1998 and this figure will be over one billion at the beginning of the new millennium^[1]. The growth of Internet and institution intranets during the past five years has led to more electronic information becoming available to people than ever before. This trend is set to progress rapidly with the increasing penetration of networks into homes. Information that was once primarily text-based has quickly branched into the areas of images, sound and video clips. While enjoying the rapid growth of information accessibility, users are often faced a serious problem, i.e., information segregation and management.

The increase in awareness of electronic information has led to the introduction of a wide range of communication protocols to support the volume of information that is becoming accessible. This has led to the electronic information segregation and formation of information islands according to its communication protocol^[2]. This problem is directly related to the growth of the distributed information in the Internet. Apparently, there is a greater need for a powerful information navigator that can help users to manage and navigate through the information that is available. Due to the heterogeneity of presentation formats associated with different information systems, the information

navigator will have to process certain characteristics to assist users in dealing with and managing distributed information. The characteristics include ability to communicate with disparate information systems, extensibility to deal with new systems and dynamism to react to unpredictable changes in the information and in the environment.

1.2. MOTIVATION

Agent technology is an important solution to the information segregation and management problem. The important attributes of agent-based systems are abstraction, inter-operability and indirect-management. Agents can inter-operate at many different levels; with the user at the desktop, with information and legacy systems in both local and distributed manners, and with other agent-based systems. For example, the characteristic of mobility allows agents to address information distributions, especially those that involve large networks of machines that are spread over a wide area. Agents can undertake tasks on behalf of their users and tasks can be carried out in the background without explicit user intervention. Agent technology will provide the next generation of management approaches that integrate with existing and future information systems to assist the users in the management of their distributed information.

In fact, agent technology has provided an exciting computing and problem-solving paradigm. It has been predicted that agents will be the most important computing paradigm in the next ten years^[3]. The world wide market for agent software will grow from an estimated value of \$3M in 1994 to about \$3B by the year 2000^[3]. There is a wide range of applications that are making the use of agent-based systems. Agent applications are being developed for fields as varied as manufacturing, entertainment,

electronic commerce, user assistance, service and business management, information retrieval, and research. Multi-agent systems including IBM Aglet^[4], Grasshopper^[5], Voyage^[6] and JAFMAS^[7] are among the most well known agent frameworks. As the number of agent frameworks continues to grow, it is important to consider the reusability of the agent software. The motivation of this thesis work is to extract and document the existing, well-proven and reusable design practices via the investigation of various agent frameworks.

1.3. APPROACH

Pattern research has played an important role in the object-oriented design and programming. Software patterns have become an important part in the development of high performance, robust and reusable computer software. This is because patterns address recurring problems that arise in specific design situations and document well proven design experience.

Agent software development is a fast growing field and many good designs and architecture patterns are embedded in the existing agent frameworks. These patterns can be extracted and documented since they represent the valuable experience of many agent developers. They can be reused in new agent-based applications. This thesis represents the search and documentation of design and architecture patterns that have been applied in agent frameworks.

There are many topics, including agent functionality, collaboration, interaction and policy structure, that are of the interests in the search for agent patterns. However, this thesis focuses on the topic of agent policy patterns.

1.4.OBJECTIVES

The objective of this thesis is to investigate various agent frameworks and to document patterns that are relevant to policy structures in agents and agent framework.

1.5.OUTLINES OF THIS WORK

Chapter 2 addresses the fundamental background and terminology of agent and multi-agent systems. A brief introduction of some commercial agent frameworks is provided. Chapter 3 provides a background of patterns; the common formats of a pattern; the roles a pattern plays in the software development. A discussion of agent patterns is also provided in this chapter.

In Chapter 4, six agent policy patterns that were extracted from existing agent frameworks are introduced. These patterns are Agent Abstraction Pattern; Use Service Pattern; Service-Based Implementation Pattern; Layered Structure Pattern; Agent Policy Delegation Pattern; Hashtable Pattern. The Agent Abstraction pattern provides a way to construct a generic base agent. The Service-Based Implementation pattern and Layered Structure Pattern provide different levels of policy grouping or partitioning. The former is a vertical partition and the latter is a horizontal partition. The Agent Policy Delegation Pattern and Hashtable Pattern provide more details on the concrete policy implementations. The Use Service Pattern serves as a bridge between the agent and policy structures.

In Chapter 5, A high-level design of the Distributed Book Ordering System (DBOS) is introduced to demonstrate the application of above agent policy patterns. The DBOS consists of a user interface, a book ordering broker and remote bookstores (servers), and

it allows users to create their agents. The DBOS broker agent accepts a book query from a user agent and dispatches a query agent to the remote servers using JAVA RMI. The query agent will migrate to various servers and return with the search results including the book availability, price and deliverable date. In this example, we focus discussions on how the agent policy patterns are applied rather than the detail implementation of a functional rich application in order to demonstrate the use of agent policy patterns in the high-level design of a distributed agent environment.

In Chapter 6, The contributions of this thesis work are summarized and some future developments of the agent policy pattern are also suggested.

CHAPTER 2: THE AGENT PARADIGM

2.1. WHAT IS AN AGENT?

Software agents can be traced back to the early days of artificial intelligence work, e.g., Hewitt's concurrent actor model^[8]. In his model, Hewitt proposed the concept of a self-contained, interactive and concurrent-executing object which he termed an actor. This object had some encapsulated internal state and could respond to messages from other similar objects. Today, when software agents are discussed, dimensions such as agency, intelligence and mobility, are used to measure the capabilities of an agent^[9]. Agency deals with the degree of autonomy of the software agent in representing the user to other agents, applications and computer systems. For example, an agent represents, helps and guides the user. In some cases, it takes unilateral actions on the user's behalf. Intelligence refers to the ability of the agent to capture and apply application-specific knowledge in solving problems. An agent is mobile if it can move between systems in a network. Mobility introduces additional complexity to an intelligence agent. In brief, we can describe agent using the following informal definition by Lecky-Thompson^[10]

"A piece of software which performs a given task using information gleaned from its environment to act in a suitable manner so as to complete the task successfully. The software should be able to adapt itself based on changes occurring in its environment, so that a change in circumstances will still yield the intended result."

Let us explore briefly what makes software designated an agent different from a software object in the views of inter-operability and indirect management. The difference between

object-oriented programming and agent software engineering lies in the language that one uses to describe the messages passing between their external interface. For the former, the meaning of a message can vary from one object to another. In agent-based software engineering, agents use a common language with agent independent semantics. The adoption of a universal agent communication language (ACL) is a higher level of interoperability than the object-oriented programming.

Traditionally, computer software programs are user-centric and required direct manipulation by users. However, with the advances in computer technology, the fundamental characteristics of computers are changing to support increased functionality, more complex applications and interactive interfaces. The concept of user-centric manipulation no longer supports the required and extended capacities. The software agent concept advanced the idea that a user no longer had to execute explicitly a given set of applications to complete a task. The user confers the details of what the task agent is expected to perform instead of actually how to perform it. It is the responsibility of the agent to interpret the task and achieve it accordingly. Thus the user is abstracted from the details of the individual tools that are being used.

2.2. AGENT CHARACTERISTICS

As defined previously, an agent is a self-contained software entity responsible for performing part of a programmatic process. It contains some level of intelligence, ranging from simple predefined rules to self-learning machines. It acts typically on behalf of a user or a process enabling task automation. More advanced agents may cooperate with other agents to carry out tasks beyond the capability of a single agent. Moreover, with

mobility, an agent may move from one system to another to access remote resources or to meet or cooperate with other agents. In principle, agents can be characterized by the following attributes^[11]:

Intelligence: This attribute indicates two aspects. One is the creation of the programmatic content of the agent. The other is the knowledge representation, which provides the means to express goals, tasks, preferences and vocabularies appropriate for various applications.

Asynchronous Operation: An agent may execute its tasks totally decoupled from its user or other agents. This means that agents may be triggered by the occurrence of a certain event, or by a time of a day. An agent placed within the network may operate totally asynchronous to the user, performing its task by talking to various system resource agents.

Communication: During their operation, agents may communicate with various system resources and users via messaging. For example, if a printing agent is looking for a printer to print a document on behalf a user, it may message the resource agent directly or it may broadcast the request to all the active resource agents.

Cooperation: This attribute indicates that an agent system allows cooperation between agent entities. The complexity of cooperation may range from a client/server style of interaction to negotiations and cooperation.

Mobility: In order to perform specific tasks, agents may be transported through a network to remote sites, usually in a specific run-time environment. Generally, two levels of agent mobility can be identified:

- a) Remote Execution: An agent is transferred to a remote system, where it is activated and executed in its entirety. The transport mechanisms used for agent transport vary from TCP/IP to electronic mail.
- b) Migration: During its execution, an "active" agent may move from node to node in order to progressively accomplish its task. In other words, agents are capable of suspending their execution, transporting themselves (i.e. program code, data, and execution states) to another node in the network, and resuming execution from the point at which they were suspended. In addition, agents may launch new agents during their journey, e.g. for delivering acquired information to the client or to perform specific sub-tasks.

Reactivity and Proactivity: Agents can perceive their environment and respond to the changes that occur around them in a timely fashion. Their actions are performed as the result of triggering rules or of executing stereotyped plans^[12]. Agents can also exhibit goal-oriented behavior by taking the initiative. They can reason about their intentions and beliefs according to their planned course of actions^[13].

The above lists only some main characteristics of agents. Depending on the notions and functional descriptions, agents can have many other characteristics such as social ability^[13], temporal continuity^[13], risk and trust^[14].

2.3.SERVICES OF AGENCY

Agency, sometimes also called agent platform, is an agent's execution context or environment that provides many common services. Normally, the environment supports agent execution, management, security, mobility and communication. The following describes some of the core services supported by an agency^[15]:

Agent Execution: Taking into account agent-specific requirements regarding the runtime environment, an agent platform must provide the capability to create agents. Before the creation, the platform has to retrieve the agent's code that may either be delivered during the creation request or downloaded separately from an external code base.

Management: It is necessary for agent administrators to monitor and control their agents. The control is comprised of an agent's task interruption and modification, and agent premature termination. The monitoring of an agent is associated with its localization in the scope of the distributed environment.

Security: The privacy and integrity of agents as well as agent systems must be guaranteed throughout their entire lifetimes. This requirement comprises the encryption and decryption of agents during migration, the authentication and authorization of agents and agent systems, and access control regarding the resources of an agent system or even of an agent offering some functionality.

Mobility: Mobility must be provided by the platform, supporting remote execution as well as migration.

Identification: Agents and agent systems have to be uniquely identifiable in the scope of the agent environment. Thus, special support is required for the generation of unique agent identifiers.

Communication: Agents should be able to communicate with each other as well as with platform services. Several mechanisms are possible, such as messages, method invocation or blackboard mechanism. Communication through messages may be done by point-to-point, multicasting or broadcasting.

The core services provide the mandatory functionality that is required for the general support of agents and for the maintenance of the agent platform itself. Additionally, enhanced services can be plugged into the agency to offer more functionality.

2.4. THE NOTION OF AGENT

The notion of agent depends on the characteristics and behavior of the agent. An agent whose nature is well defined and whose behavior is predictable is more likely to be trusted by the user. There are two notions of agent, i.e., weak and strong agents, developed by Wooldridge and Jennings^[16].

2.4.1. The notion of a weak agent

A weak agent may possess the following properties:

- ♦ autonomy: agents operate without the direct intervention of humans or other agents and have some kind of control over their actions and internal state;
- ♦ social ability: agents interact with other agents and humans via some kind of agent communication language;

- ◆ reactivity: agents perceive changes that occur in their environment and respond to the changes in a timely fashion. This may entail that an agent spends most of its time in the sleep state from which it will awake if certain changes in its environment take place;
- ◆ proactivity: agents do not simply act in response to their environment; they are able to exhibit goal-directed behaviour by taking the initiative;
- ◆ temporal continuity: agents are continuously running processes either running/active in the foreground or sleeping/passive in the background;
- ◆ goal-orientedness: an agent is capable of handling complex, high-level tasks. The decision how such a task is best split up in to smaller sub-tasks, and in which order and in which way these sub-tasks are best performed, should be made by the agent itself.

2.4.2.The notion of a strong agent

Agents that fit the notion of a strong agent usually have one or more of the following characteristics:

- ◆ mobility: agents have the ability of moving around an electronic network;
- ◆ benevolence: agents do not have conflicting goals; therefore, they always try to do what is asked;
- ◆ rationality: an agent will act in order to achieve its goals and will not act in such a way as to prevent its goals being achieved;
- ◆ adaptivity: an agent should be able to adjust itself to the habits, working methods and preferences of its user;

- ♦ collaboration: an agent should not unthinkingly accept and execute instructions. It should take into account that the human user makes mistakes such as omitting important information and/or providing ambiguous information.

2.5. AGENCY AND AGENT INTELLIGENCE

The degree of autonomy and authority vested in the agent is provided by agency. It can be measured at least qualitatively by the nature of the interaction between the agent and other entities in the system. At a minimum, an agent must run asynchronously. A more advanced agent can interact with other entities such as data, applications and services, or collaborate and negotiate with other agents.

Agent intelligence is something different from agency. It has been defined as follows^[10]:

"Intelligence is the degree of reasoning and learned behaviour: the agent's ability to accept the user's statement of goals and carry out the task delegated to it. At a minimum, there can be some statement of preferences, perhaps in the form of rules, with an inference engine or some other reasoning mechanism to act on these preferences. Higher levels of intelligence include a user model or some other form of understanding and reasoning about what a user wants done, and planning the means to achieve this goal. Further out on the intelligence scale are systems that learn and adapt to their environment, both in terms of the user's objectives, and in terms of the resources available to the agent. Such a system might, like a human assistant, discover new relationships,

connections, or concepts independently from the human user, and exploit these in anticipating and satisfying user needs."

2.6.CATEGORIZING OF AGENTS

A common way to categorize agents is according to their job or functional descriptions. For example, if an agent provides information services, it may be categorized as information agent. There are many kinds of agents due to the variety of agent applications and it is impossible to introduce them all. In the following, therefore, only the most commonly used agents are discussed.

Artificial intelligence agent

Artificial intelligent agents, originating from the artificial intelligent community, provide intelligence and reasoning capabilities that are comparable to or in excess of those possessed by human beings. Artificial intelligence architectures are generally based on three core philosophies: deliberative, reactive and a hybrid of these^[17].

A deliberative model is to reduce the world to a representation of realizable symbols that can be combined to form structures. Upon the structures an agent can operate on the symbols through logical reasoning based on pattern matching and symbolic manipulation.

A reactive model is a hierarchy of behaviors designed to accomplish tasks. Each behavior competes with the other behavior in order to influence the actions of the agent. Lower layers in the hierarchy represent primitive behavior styles and higher layers represent more abstract behaviors. A hybrid model combines the best qualities of both the deliberative and reactive approaches. It consists of building an agent system out of two or more subsystems; a symbolic world model with which developers plan and make

decisions, and a reactive subsystem which is capable of responding quickly to events that happen without having to resort to the complex symbolic manipulation.

User Interface agent

User Interface agents assist the user in whatever tasks they are performing. For example, the MIT has developed a prototype interface agent called News Tailor that is a news filter. The agent can be trained by given a series of examples that show in which kind of article the user is interested^[18]. From this, the News Tailor can search all news articles to find articles that are similar to those initially indicated by the user. The agent presents the articles that it has found to the user and the latter, according to the applicability of the found articles, gives feedback to the agent. Thus, the News Tailor can widen or restrict its searching next time.

Information agent

An information agent is the one that has access to a number of information resources and is able to collect and manipulate information. Typically, it can communicate across the network to locate, query or manipulate the information. An example might be where an information agent is asked to find a particular paper; the information agent searches a number of information resources and presents the user with FTP sites and WWW addresses. The key qualities of information agents lie with their ability to communicate with a large range of information resources to ensure that the widest amount of information is processed in order to provide the user with the best results. Generally, an information agent will possess some of the characteristics of an interface agent and it will

have to develop and maintain a user profile to determine how to best deliver the information that a user needs to reduce information overload.

Believable agent

Believable agents provide the illusion of life and thus permit the audience's suspicion of disbelief. They have special ingredients such as emotion, perception, and reactive and goal-directed behavior. Perception is achieved by low-level sensory routines. The integrated sensory model maintains the agent's best guess about the physical structure of the whole world and is fed by the raw data of the sensory routines. The success or failure of goals generates different emotional states that model the emotion^[19]. For example, the fulfillment of a goal produces a corresponding happiness emotion; however, emotion fades with time as other emotions take priority. Reactive and goal-directed behavior (handled by happiness) continuously choose the next action of the agent based upon current exceptions, goals, and the emotional state.

Reactive agent

Reactive agents represent a special category of agents that do not possess internal, symbolic models of their environments. Instead they act or respond in a stimulus-response manner to the present state of the environment in which they are embedded^[20].

Collaborative agent

These agents emphasize autonomy and cooperation in order to perform tasks for their users. Their key attributes include autonomy, social ability, responsiveness and proactiveness. To achieve the user designated tasks, collaboration agents may have to negotiate with other collaborative agents to reach mutually acceptable agreements^[21].

Distributed artificial intelligence agent

Distributed artificial intelligence agents are collective agents which together sit at the social level rather than the agent level^[22]. Distributed artificial intelligence looks at how problems at the social level can be broken down to agents at the agent level and how those agents can be made to cooperate, negotiate and coordinate their activities to ensure that the problems are solved efficiently.

Mobile agent

Mobile agents are computational software processes capable of roaming wide area networks, such as WWW, interacting with foreign hosts and gathering information on behalf of its user^[9]. Mobile Agents are aimed primarily at large computer networks offering a huge number of sophisticated services. Examples range from advanced Internet filter and search agents^[23], to smart messaging, network connection^[24], intelligent communication and management^[25].

2.7.MULTI-AGENT SYSTEMS

There are many multi-agent frameworks that have been developed to provide agent programmers a platform to build agent-based applications. The following list briefly describes some well-known systems including IBM Aglets Workbench^[4], Concordia^[26] from Mitsubishi Electric Information Technology, Odyssey^[27] from General Magic and Voyager^[6] from ObjectSpace. For more information on the comparison of various agent frameworks, one may visit the web site "http://pent21.infosys.tuwien.ac.at/cetus/oo_mobile_agents.html".

IBM Aglets Workbench^[4]: Aglets Workbench is a visual environment for building network-based applications that use mobile agents to search for, access and manage data and other information. IBM Aglets are mobile Java programs which may travel and execute in specialized nodes in the network. The Java Aglet Application Programming Interface of the framework defines the methods necessary for aglet creation, message handling in the network and initialization, dispatching, retraction, deactivation/activation, cloning and disposing of the aglet. Aglets workbench also implements a configurable security manager that provides a high degree of security for the hosting computer system and its owner.

Concordia^[26]: Mitsubishi Electric Information Technology Center of America has developed a Java-based framework for development and management of network-efficient mobile agent applications for accessing information anytime, anywhere, and on any device. Concordia provides support for agent persistence and recovery and guarantees the transmission of agents across a network. It also provides certain security coverage. Within Concordia, an agent's travel plans are specified by its Itinerary. The Itinerary is a completely separate data structure from the agent itself. Concordia provides two forms of asynchronous distributed events: selected events and group-oriented events. The event selection paradigm enables agents to define the types of events they wish to receive. In contrast, group-oriented events are distributed to a collection of agents without any selection.

Odyssey^[27]: Odyssey is an agent system implemented as a set of Java class libraries that provide supports for developing distributed and mobile applications. Odyssey technology

implements the concepts of places for agents. It models a network of computers as a collection of places. A place offers services to the mobile agents that enter it. An agent can move from one place to another, thus occupying different places at different times. Odyssey provides Java classes for mobile agents and stationary places. Odyssey agents are Java threads and they rely on the same security services as all other Java applications.

Voyager^[6]: Voyager is a Java-based agent-enhanced Object Request Broker (ORB) developed by ObjectSpace Inc. It allows Java programmers quickly and easily to create sophisticated network applications using both traditional and agent-enhanced distributed programming techniques. It provides for the creation of both autonomous mobile agents and objects. Voyager agents roam a network and continue to execute as they move. Voyager can remotely construct and communicate with any Java class, even third party libraries, without source. It allows seamless support for object mobility. Once created, any serializable object can be moved to a new location, even while the object is receiving messages. Messages sent to the old location are automatically forwarded to the new location. Voyager is a very efficient tool for constructing agent-based distributed applications.

Grasshopper^[5]: Grasshopper is an agent platform that is also entirely implemented in Java. Moreover, it is built on top of CORBA. Similar to Voyager, it combines the client/server paradigm and mobile agent technology. Agents are transferred via CORBA IIOP, Java RMI or plain sockets. Apart from runtime environment, Grasshopper provides various plug-ins to enhance the core functionality of agent and agent systems.

Sophisticated communication and security mechanisms are integrated into the platform core as well.

JATLite^[28]: The Computer Science Department at Stanford University developed JATLite that provides a set of Java packages to facilitate agent framework development using the Java language. JATLite provides basic communication tools and templates for developing agents that exchange KQML messages through TCP/IP. It defines a special construct called an Agent Name Server (ANS) which stores all the names and addresses of existing agents. When an agent is created and connected to the network, it first registers with the ANS. In registering, the agent passes the ANS its name, port number and its local host. If an agent knowingly terminates, it first sends a remove address message to the ANS, which echoes the message to all the other agents. JATLite also provides special Agent Router functionality which allows Java applets to exchange messages with any registered agent on the Internet. The Agent Router allows any registered agent to send messages to any other registered agent by making a single socket connection to the Agent Router. Messages are forwarded without the sending agent having to know the receiving agents' address and making a separate socket connection with the ANS infrastructure.

JAFMAS^[7]: Java-based Agent Framework for Multi-agent Systems (JAFMAS) defines a generic methodology for multi-agent application development and provides a set of services that relieves the developer from the effort of programming cooperation mechanisms from scratch. It guarantees that essential interoperation, communication and cooperation services will always be there to support applications. Unlike other agent

building tools, JAFMAS is concerned with representing, developing, and analyzing coordination knowledge and protocols among a collection of intelligent agents forming the multi-agent system. It enables the agents to work together and coherently achieve their goals.

There are still many more agent platforms that have been developed including InteRRap^[29], dMars^[30], Cybele^[31], AgenTalk^[32], AgentTcl^[33], Telescript^[34], Swarm^[35], Echelon^[36], COOL^[37]. References for these systems are provided at the end of this thesis for those who are interested.

2.8.SUMMARY

This chapter described the basic terminology of agents and agent platforms. The agent definition, various agent characteristics, concept of agency, notions of an agent and different kinds of agents are introduced. Some prevalent agent-building platforms available today were briefly described including aglets Workbench from IBM, Odyssey from General Magic, Voyager from ObjectSpace and Grasshopper from IKV++.

CHAPTER 3: SOFTWARE PATTERNS

3.1.WHAT IS A PATTERN?

Experience makes the difference between a productive programmer and a less productive programmer because experience provides programmers with wisdom. As programmers gain experience, they can recognize the similarity between new problems and problems that they have solved before. With a knowledge of patterns, experienced programmers also can recognize the situations to which patterns apply and immediately use the solution without having to stop, analyze the problem or pose possible strategies.

Software patterns are reusable solutions to recurring problems that occur during software development. Some software pattern characteristics include the following^[38]:

- ◆ A pattern addresses a reoccurring design problem that arises in specific design situations and presents a solution to it;
- ◆ Patterns document existing well-proven design experience. They are not created or invented artificially rather than distilled. They provide the means to reuse the design knowledge gained by experienced practitioners.
- ◆ Patterns identify and specify abstractions that are above the level of the single class and instance of components. Typically, a pattern describes several components, classes and details of their responsibilities, relationships and cooperation.
- ◆ Patterns provide a common vocabulary and understanding for design principles. Pattern names, if chosen carefully, become part of a widespread design language. They facilitate effective discussion of design problems and their solutions.

- ◆ Patterns are a means of documenting software architectures. They can describe the vision a designer has in mind when designing a software system. This helps others to avoid violating this vision when extending and modifying the original architecture.
- ◆ Patterns support the construction of software with defined properties. They also provide a skeleton of functional behavior and help the functionality implementation of an application.
- ◆ Patterns help designers to build complex and heterogeneous software architectures. Every pattern provides a predefined set of components, roles and relationships between them.
- ◆ Patterns help designers to manage software complexity. Every pattern describes a proven way to handle the problem it addresses. When you encounter a concrete design situation covered by a pattern, there is no need to waste time inventing a new solution to your problem. If you implement the pattern correctly, you can rely on the solution it provides.

3.2. A BRIEF HISTORY OF PATTERNS

The idea of software patterns originally came from the field of architecture. In the late 1970s, an architect, Christopher Alexander, wrote two revolutionary books that describe patterns in building architecture and urban planning: “*A pattern Language: Towns, Buildings, Construction*”^[39] and “*The Timeless Way of Building*”^[40]. The ideas presented in these books are applicable to a number of fields outside of architecture including software design.

In 1987, Ward Cunningham and Kent Beck used some of Alexander's ideas to develop five patterns for user-interface design^[41]. Since then, a few books on software design patterns have been published including: "*Design Patterns*" by Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides, often referred to as the Gang of Four book^[42], "*Pattern Language of Program Design*" edited by James O. Coplien and Douglas C. Schmidt^[43], "*Pattern-Oriented Software Architecture: A System of Patterns*" by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal^[44].

In 1998, two volumes of "*Patterns in Java*" by Mark Grand were also published^[45-46]. Volume 1 is mainly a Java translation of many existing patterns and volume 2 introduces new patterns. Both volumes introduced the Unified Modeling Language (UML) notation.

3.3. PATTERN DESCRIPTIONS

Focusing on a particular object-oriented design problem or issue, a design pattern identifies participating classes, instances and their roles, collaborations and distribution of responsibilities. It describes when and where the pattern applies in the view of design constraints, consequences and trade-offs of its use. In general, a pattern has four essential components: name, problem, solution and consequence.

The pattern name is a handle that describes a design problem, its solution, and consequences in a word or two. The name of a pattern can be used as design vocabulary since it provides us a higher level of abstraction and it makes us easy to think about the design.

The problem component explains a particular design problem and its context. It might also describe class or object structures that are symptomatic of an inflexible design. Sometime the problem will include a list of conditions that must be met.

The solution describes the elements that make up the design, their relationships, responsibilities and collaborations. The solution doesn't provide a particular concrete design or implementation because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.

The consequences are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trades-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility and portability.

3.4. PATTERN FORMATS

Different forms have evolved to record patterns including Alexanderian, GOF, Portland, Coplien and Canonical forms. The latter three are very similar. Different authors tend to use slightly different formats to record their patterns; however, there are two formats that are most commonly used in practice: the GOF (Gang of Four) ^[42] and Canonical formats^[43].

3.4.1. GOF Format

Many software pattern researchers followed the GOF pattern format described in the book *Design Patterns* by Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides^[42]. The sections contained in this pattern format are briefly introduced as follows:

Pattern Name: the pattern's name conveys the essence of the pattern. A good name is vital because it will become part of design vocabulary.

Intent: a short statement that answers the questions of what the design pattern does what its rational and intent are and what particular design issue or problem the pattern addresses.

Also Known As: other well-known names for the pattern if any.

Motivation: a scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario helps software designers to understand the abstract description of the pattern that follows.

Applicability: this section describes the situations in which the design pattern can be applied and examples of poor designs that can be addressed.

Structure: it is a graphical representation of the classes in the pattern using a notation based on the object-orientated modeling.

Participants: this section describes classes and objects that participate in the design pattern and their responsibilities.

Collaborations: it describes how all the participants collaborate to carry out their responsibilities.

Consequences: this section addresses how the pattern supports its objectives, what the benefits and trade-offs are of using the pattern.

Implementation: it states what pitfalls, hints, or techniques one should be aware of when implementing the pattern.

Sample Code: code fragments that illustrate how one might implement the pattern in a software language.

Known Uses: examples of the pattern found in real systems.

Related Pattern: it describes what patterns are closely related to this pattern and what the important differences are.

3.4.2. Canonical Format

Today, more people tend to follow the Canonical format due to its relative simplicity.

The sections contained in the Canonical format are also briefly stated as follows^[47]:

Pattern name: A heading indicates the nature of the pattern.

Context: this section describes the situation of the problem that the pattern addresses. For most patterns, the context introduces the problem in terms of a concrete example and suggests design requirements.

Forces: this section summarizes the different possible solutions that pull the problem in different directions and their design trade-offs. It has the format of “if we do this...it may solve the problem, but ...”

Problem: the problem is often stated as a question or design challenge that the pattern addresses.

Solution: this section is the core of the pattern. It describes a general-purpose solution to the problem that the pattern addresses.

Consequences: this section explains the implications, good and bad of using the pattern.

Implementation issues: this section describes the important considerations to be aware of when executing the solution. It may also describe some common variations or simplifications of the solution. Some patterns may not have an implementation section because these considerations are not relevant to every pattern.

Code example: it contains a code example that shows a sample implementation for a design that uses the pattern, For some patterns, the code example may not be relevant.

Related Patterns: this section contains a list of patterns that are related to the pattern described.

Resulting Context: this section tells which forces are resolved and which forces are remained unresolved by the pattern; it points to more patterns that might be the next ones to consider.

There are some differences between the GOF and Canonical formats since they have different subtitles. A brief comparison of the two formats as follows:

- The context: In the Canonical form, the context is clearly defined to provide a clear situation to the reader. In the GOF form, the context is introduced through the motivation and applicability. It is hard, sometimes, to interpret the context clearly.
- The problem: The Canonical form defines the problem explicitly that allows the reader "to get to the point" immediately. In the GOF form the problem is implicitly given in the motivation.
- The forces: The forces are given clearly in the Canonical form, which allows the reader to understand the "pushes" behind the scheme. However, in the GOF form, the forces are not explicitly given. They are included in the motivation and the applicability and are difficult to identify.
- The solution: In the Canonical form, the solution is explicitly given after the introduction of the context, problem and forces. The solution provides the reader a "release" effect after viewing how the forces push everything in to a "box". In the GOF form, the problem is revealed in the intent, which allows the reader to know what the pattern is for at the very beginning and it does not produce the extra "release" effect on reading through the pattern.

Generally speaking, the Canonical form is easy to read and understand since it follows a simple context-forces-solution logic. A well-written pattern in Canonical form can

provide the reader a clear contrast between using and not using the pattern. In this thesis, the Canonical form is adopted in documenting the agent policy pattern in Chapter 4.

3.5. PATTERN LANGUAGE

A pattern language is a collection of patterns that build on each other to generate a system^[47]. A pattern in isolation solves an isolated design problem; a pattern language builds a system. It is through pattern language that patterns achieve their full power. In other words, software patterns only go so far. They show you how to build pieces of an application, but not an entire application. To build a complete application, you need a language of patterns -- a pattern language. The pattern language consists of individual patterns plus a network of connections among the patterns showing how the patterns may be combined together. Just as the English language, with its vocabulary and rules of grammar, one can generate an infinite variety of sentences. A software pattern language, with its individual patterns and rules for combining them, one can generate an infinite variety of software applications^[48].

For example, Figure 3.1 illustrates the Graphical User Interface pattern language^[49]. The GUI pattern represents the core and provides the overall architecture for connecting models, views and controllers. The Subform patterns help to describe how to structure the views. The View Event Handler pattern describes how a view should handle an event message such that its update process is independent from the visual component generating the event and from the models that have changed state. The Complete Update pattern describes how to structure a view's update code such that other developer can quickly and easily understand the logic of the update. The Multiple Updates pattern

describes how to manage the update of visual components arranged in multiple subforms. To use a software pattern language to design an application, you start at the topmost pattern in the network of patterns. You grow your software design marginally by doing what the pattern suggests. Then you grow each part of the design incrementally using the next lower patterns in the network. The highest patterns in the network give very broad design guidance; lower and lower patterns in the network give more and more detailed design guidance. Working through the pattern language from top to bottom causes your software design to grow like an unfolding flower.

Figure 3.1. GUI pattern language^[49].

3.6. AGENT PATTERNS

Agent-based applications, in both academic research and commercial product development ranging from electronic commerce to network management, has dramatically increased in recent years. This trend has stimulated the research in agent patterns. In fact, various agent pattern works have been published^[50,51-52,54]. These

patterns can be categorized into three levels, e.g., multi-agent system, agent architectural and agent activity.

3.6.1. Multi-agent System Pattern

The Agent Pattern was suggested by Alberto Silva and Jose Delgado^[50]. The intent of this pattern is to define an easy way to build dynamic and distributed applications using autonomous and active objects called agents.

In an agent-based approach, users interact with their own agents and delegate their specific tasks. Agents interact between themselves in order to perform these respective tasks. For example, Figure 3.2 shows a scheme in which a user wants to buy a book and interacts with his agent to pass specific information. All the subsequent actions such as accessing and interacting with the broker, obtaining the addresses of the relevant bookseller agents, accessing and negotiating with some bookseller agents; deciding whether to buy or not and buying the book, are executed indirectly by the respective agent.

Figure 3.2: Global view of an agent-based application.

Agent-based applications built using the Agent Pattern will have a number of characteristics including autonomous, heterogeneous, open, dynamic, robust and secure. All these characteristics make the applications potentially very difficult to implement if they are not using the agent approach. Figure 3.3 shows a generic static structure of the Agent Pattern^[50].

Figure 3.3. Generic structure of the Agent pattern – class diagram.

The User manipulates agents through the AgentView which is an adaptation of the Proxy^[42] and Remote Proxy^[42] patterns. The aim of AgentView is to provide location transparent and secure access to agents. UserID is identified by a unique identity, which may contain the user's name, a public key, a set of certificates, the organization and country the user belongs to and users e-mail.

The Agent class is the visible and extensible part of the Agent Pattern. Basically, programmers derive the Agent class in order to build their own concrete classes. The agent class has three main groups of methods: public final, callbacks, and helper

methods. The agent instance provides transparently services such as persistence, communication, mobility, naming and access control. Additionally, the Agent instance may keep related information, such as the current and native place identities, security policy object, a reference to the concrete agent itself and other references of objects involved. The ConcreteAgent classes are Agent subclasses that are used by agent programmers to build their specific agents.

The SecurityManager class enables the agent to access security policy. The ExecutionPlace provides the agent a computational environment, which corresponds to the context or place where agents are created and reside. This class offers specific functions provided by the agent support system. The notion of execution context or place is a component supported by agent frameworks.

The Agent Pattern provides a clean and easy way to develop agent-based applications, mainly in open and large-scale distributed environments such as the Internet and application areas such as Electronic Commerce. The Agent Pattern encapsulates a business specific class with some user identification and a specific security policy.

3.6.2. Agent Architectural Pattern

The Agent Pattern provides an overall design structure for an agent system but it does not get into the detail of how to structure an agent. Kendall et. al.^[51-52] proposed an Agent Layer Structure Pattern, which suggests that agents should be decomposed into layers for the following reasons: i) higher level behavior depends on lower level capabilities, ii) levels only depend on their neighbors, and iii) there is two way information flow between neighboring levels.

Figure 3.4. Agent Layer Pattern.

In Kendall model as indicated in Figure 3.4, the agent has seven layers of abstraction as follows:

1. Sensory: the agent gathers sensory input from its environment.
2. Beliefs: the agent's models of itself and its environment.
3. Reasoning: the agent determines what to do next, based on its goals, plans, capabilities, and models. The plans may generically specify requests for services or responses to requests.
4. Action: the agent's intentions (instantiated plans from the reasoning layer) are carried out. Pending actions for the agents are queued, and these are derived from incoming requests that have been accepted by the collaboration layer.

5. Collaboration: the agent determines how to collaborate with another agent. This includes accepting or rejecting an incoming request for services.
6. Translation: the agent formulates a message for another agent, translating it into another language or semantics (ontology), if necessary.
7. Mobility: when collaborating agents are in different locations, this layer is required for messaging

In Figure 3.4, a top down information flow is shown on the left side while a bottom up is on the right. On the right side, an agent's beliefs are based on sensory input. When presented with a problem, an agent reasons about its beliefs to determine what to do. When deciding on an action, the agent can carry the action out directly, but an action that involves other agents requires the collaboration layer. Within this layer, the agent decides how to negotiate with another agent. Once the approach to collaboration is determined, the actual message is formulated at the translation layer if the collaborating agents have different semantics. The mobility layer delivers messages to distant societies.

On the left side, an incoming message is translated into the agent's semantics. The collaboration layer stores contacts and conversations, and determines whether the agent should process a message. If yes, the message is passed to the actions layer. When the actions are dequeued for processing, they are passed to the reasoning layer. Depending on the type of request, processing may continue to the beliefs and sensory layers. Once a plan is instantiated as an intention and placed in the action layer, it does not require the services of any lower layers. The concept of agent layer structure has been used by many agent frameworks^[51-53]; however, the number of layers varying.

3.6.3. Agent Activity Patterns

The agent patterns discovered by Danny Lange are mainly patterns of agent activities and they are categorized into three groups: traveling, task and interaction^[54]. The traveling patterns deal with various aspects of managing the movements of agents, such as routing and quality service. Patterns in this group are Itinerary, Forward and Ticket patterns. The Itinerary pattern is concerned with routing among multiple destinations. An itinerary maintains a list of destinations, defines a routing scheme, handles special cases such as what to do if a destination does not exist, and always knows where to go next. Objectifying the itinerary allows you to save it and reuse it later in much the same way that you save URLs as bookmarks. The Forward pattern allows a given host to mechanically forward agents to another host. The Ticket pattern, an enriched version of the concept of a URL, embodies requirements concerning quality of service, permissions, and other data. For example, it can include time-out information for dispatching an agent to a remote host.

The task patterns are concerned with the break down of tasks and how these tasks are delegated to one or more agents. In general, tasks can be dynamically assigned to general-purpose agents. A given task can be accomplished either by a single agent or by multiple agents working in parallel. There are two fundamental patterns in this group: the Master-Slave and Plan patterns. The Master-Slave pattern allows a master agent to delegate a task to a slave agent and the latter performs the task and returns the result to the former. The Plan pattern adopts a workflow concept to organize multiple tasks to be performed in sequence or in parallel by multiple agents. The plan hides the task flow

from the agents and it promotes reusability of tasks and dynamic assignment of tasks to agents.

The Interaction patterns deal with locating agents and facilitating their interactions. Patterns in this group are Meeting, Locker, Messenger, Finder and Organized Group. The Meeting pattern provides a way for two or more agents to initiate local interaction at a given host. It abstracts the synchronization in time and place that is required for local interaction. Agents can dispatch themselves to specific destination, call a meeting place, where they are notified the arrival of their counterparts and can engage in local interaction. The Locker pattern is to temporarily store data in private. In this way, an agent avoids bringing along data that are not needed at the moment. The agent can, later, return and retrieve the private data stored in the locker. The Messenger pattern defines a surrogate agent to carry a remote message from one agent to another. The Finder pattern defines an agent that provides services for naming and locating agents with specific capabilities. The Organized Group pattern composes agents into groups in which all the members of a group travel together.

Design patterns have proven to be highly useful with the object-oriented field and have helped developers to achieve good design of applications through reusability of validated components. It is expected that the agent patterns play a similar role in agent-orientated application development.

3.7. AGENT POLICY PATTERNS

So far we have been viewing agent patterns from the perspective of agent design architecture, interactions and collaborations. In next section, we introduced the policy

concept. Policy defines an agent's behavior and actions and it provides agents the ability to think. For example, when receiving a request, an information agent may examine the requester to decide whether it is entitled to have the requested information. The agent checks its internal permission policy and provides the information only to those who qualify. In fact, the concept of policy is embedded in almost every area of an agent system. One obvious area is the security.

One objective of this thesis is to extract and document agent patterns related to policy since policy implementation is an important part of an agent framework and there is very little study focusing on agent policy patterns. In Chapter 4, we introduce the definition of agent policy and document the agent policy patterns.

3.8. SUMMARY

In this chapter, a brief view of pattern definitions, history, description and formats are provided. In addition, an introduction of agent pattern research in three levels, e.g., multi-agent system, agent architecture and agent activity, are presented. The Agent Pattern provides a clean and easy way to develop multi-agent-based applications. The Agent Layer Structure describes a useful structure for agent construction and the agent activity patterns summarized many agent interactions and collaboration patterns.

CHAPTER 4: AGENT POLICY PATTERNS

4.1.WHAT IS POLICY?

In general, a policy has many different meanings depending on where it is applied. It may mean rule, constraint and requirement for a service to be granted. In the agent world, policy defines security measures, communication language and transportation means. It also defines an agent's behavior, action/reaction plans, rules of collaboration and knowledge representation. A policy normally has an applicable domain that provides a means of grouping objects to which it applies and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority. This is because, in large-scale systems, it is not practical to specify policies for individual objects. Grouping objects into a domain to which a policy applies is more efficient. For example, an authorization policy may specify that all members of a department (the domain) have access to a particular service.

The general format of a policy contains policy identifier, mode and other attributes of a policy such as trigger, subject, action, target, constraint and cross reference as expressed below^[55]:

Identifier mode [trigger] subject { action } target [constraint][parent][child][xref];

The identifier is a label used to refer to the policy. The mode of the policy distinguishes between obligations and authorizations. The trigger is an internal or external event, which is detected by a monitoring service. The subject of a policy, defined in terms of a domain scope expression, specifies to which the policies apply. The target of a policy specifies the objects on which actions are to be performed. The actions specify what must be

performed and what is permitted for authorizations. Multiple actions in a policy may imply that they are performed sequentially after the policy is triggered. The constraint, normally defined by the when clause, limits the applicability of a policy or makes a policy valid after a particular date. Constraints must be evaluated every time a policy is triggered. A policy can be expanded from a higher level policy and can also be refined into other detailed policies. In order to record this relationship, policies automatically contain references to their parent and children policies. In addition, a cross-reference (xref) from one policy to another can be inserted.

Consider security measures as an example. In open distributed environments, agent systems face a number of serious security issues including system integrity and access control. To address these issues, the following policies must be implemented^[6].

- Confidentiality: When an agent transports confidential data, the data should not be of any potential use by anybody other than the communication partners. Disclosure or eavesdropping of the data can be fatal and must be prevented.
- Integrity: If the agent was modified or corrupted, it must be obvious to the agent system upon reception whether the cause was by means of transmission errors or by intentional vandalism. An attacker must not be able to modify agents without the agent system noticing it.
- Authentication: In a communication session, authenticity of the exchanged data requires authentication of the communication parties. If important data is transmitted, each communication party should be aware of the real identity of the communication peer.

- ❑ Access control: An agent system generally has access to resources, such as a file system, network, CPU, time and memory. These resources must be protected from unauthorized access.
- ❑ Auditing: In order to get an overview of relevant security events, the security policies mentioned above, especially authentication and the access control policies, should be able to keep track of whenever agents try to access system resources or the system itself, as well as authentication failures or when transmission faults occur. These events should be logged to a file so they can be analyzed later.

A communication language also requires policies. When agents talk to each other, they can do so in a variety of ways. They can talk directly to each other, provided they speak the same language. They can talk through an interpreter or facilitator, provided they know how to talk to the interpreter and the interpreter can talk to each other. No matter which way agents communicate, there is a level of basic policy (normally called grammar) that defines the syntax and format of the language. One agent communication language is the knowledge query and manipulation language (KQML), which provides a framework for agents to exchange information and knowledge^[56]. KQML defines operations that agents may attempt on each other's knowledge bases, and provides a basic architecture for agents to share knowledge and information through special agents called facilitators.

It is easy to understand policies in terms of security, communication and collaboration, but it is not so obvious to see the relation between policy and knowledge representation. Let us examine the following example to see how knowledge can be represented by rules

(rules are policies and we use the term rule here to be consistent with the terminology used by the artificial intelligence community).

Consider a declarative “fact”: “an automobile has four wheels and a motor”. It can be represented by the if-then rule as: “**if** number_of_wheels = 4 and motor = yes **then** vehicle_type = automobile”. In fact, if-then rules have become the most popular form of knowledge representation used in artificial intelligence applications^[57]. This is because knowledge represented as if-then rules is easy to understand. Each rule can be viewed as a stand-alone piece of knowledge or unit of information in a knowledge base. New knowledge can easily be added simply by creating a new rule or an existing knowledge can be changed by modifying individual rules. However, there are some common problems that are encountered in the large-scale commercial artificial intelligence systems^[58]. One of them is that the intuition of the if-then rules is diminished as the number of rules grows since the if-then approach loses its effectiveness from a readability viewpoint. Commercial rule-based systems apply grouping or partitioning of rules so that they can be treated as logical blocks of knowledge to overcome the weakness. Another common problem in rule-based systems is that rules that may have been true before become false as more complete information is known or things change in the outside world. In such cases, we have to remove rules that are no longer valid. Keeping a rule-based system consistent by maintaining dependencies between rules requires a truth maintainable system.

4.2.POLICY PATTERN LANGUAGE

The proposed agent policy pattern language (shown in Table 4.1) summarizes the problems addressed by the policy patterns in our pattern language and their solutions. The Agent Abstraction pattern provides a way to construct a generic base agent. The Service-Based Implementation pattern and Layered Structure Pattern provide different levels of grouping or partitioning in policy structure. The former is a vertical partition while the latter is a horizontal partition. The Agent Policy Delegation Pattern and Hashtable Pattern provide more details of the policy structures. The Use Service Pattern serves as a bridge between the agent and policy structures.

An overall relation between these agent policy patterns is described in Figure 4.1. Using the Agent Abstraction Pattern, we can construct a generic base agent that can be customized by agent developers. The agent should be able to access various services including security, communication, collaboration and transportation. The access to these services can be obtained by applying the Use Services Pattern that constructs an agent execution environment. For each service that the agent environment supports, it requires a set of policies to maintain and govern its operation. To do that, we can apply the Service-Based Implementation Pattern, Layered Structure Pattern, Agent Policy Delegation Pattern and Hashtable Pattern. These patterns provide different levels of structures for policies.

Figure 4.1:-- An overall relation between agent policy patterns.

Pattern Name	Problem	Solution
Agent Abstraction	How do you construct a generic agent class that can be used as a base class for customization?	Combine the abstract and proxy patterns to construct a generic base agent.
Use Service	How do you structure an agent that would be able to change its behavior, plans and services at run-time?	Structure an environment that allows an agent to rely on services provided rather than on its own.
Service-Based Implementation	How do you categorize or partition policies to facilitate service management?	Categorize and structure policies into service domains.
Layered Structure	How do you structure policies to accommodate layered objects in an agent?	Use a layer structure for policy objects that associate directly with the layer objects in an agent.
Agent Policy Delegation	How do you structure different policies for a layered object in an agent?	Allow policy substitution and delegation.
Hashtable	How do you organize policies that provide fast access?	Use a hashtable structure for the policies

Table 4.1. – Agent Policy Pattern Problems and Solutions.

4.3. AGENT POLICY PATTERNS

4.3.1 Agent abstraction patterns

Context:

An agent-based application can be developed on top of an agent framework that provides base agents, such as stationary and mobile agents. These base agents are generic and associated with an agent context through which different services can be accessed, and they allow agent developers to build their own version of agents by subclassing them. Base agents should provide methods that can be overridden in their subclasses by the agent developers customizing agent behavior (Figure 4.2). However, the detailed implementation of the base agent must be concealed or encapsulated to protect the integrity of the agent system.

Figure 4.2-- Agent system contains context and base agent.

There are a few issues that must be considered when you are constructing base agents:

- ❑ Encapsulation: The term encapsulation describes the fact that objects are thought of as black boxes. An object encapsulates the methods and data inside it. Encapsulation

implies that an object is self-governing, which means that the variables contained in an object can only be modified within the object. Only the methods within an object should have the access to variables in that object. The agent context object must encapsulate its variables and private methods, and its implementation details must be hidden from other agent developers. Only public methods or services are accessible by the customized agents via subclassing base agents. The most common services provided by the agent context include communication, registration, management, transport and security. Without encapsulation, agent developers would be able to modify agent context variables and alter the provided services, which is a serious issue.

- ❑ **System integrity:** Agent system provides two kinds of methods through which agents can access common services. The first kind is called a system method. System methods provide basic services such as communication and transportation and can not be overridden by other agent developers. The other is a user-defined method that allows developers to customized agents, e.g., agent developers can override these methods to make their agent specific. For example, a customized agent can migrate to a designated host using the transport method provided by the agent system. Upon arriving, the customized agent can execute its run method to perform specific tasks instructed by the agent developer in the foreign host. However, the agent developers must not be allowed to override system methods otherwise the integrity of the agent system will not be preserved.
- ❑ **Access Control—**Direct access to the agent (both base and customized agents) must not be allowed to ensure the integrity and safety of the agents. This is because there

are many dangerous events that can happen with direct access to the agent. For example, a malicious agent can review your agent's confidential data, use your agent's resources or ask your agent to perform a suicidal mission.

- Location Transparent --A mobile agent provides location transparency to the user. This means that the user has no knowledge when and where the mobile agent is dispatched. For example, an Internet information search agent needs travel to many locations before it finds the information required by its user. The user however has no idea where his agent has traversed.

Problem:

How do you structure a generic agent class that can be used as a base class for customization without giving away details of the implementation of your agent system?

Forces:

One simple approach is to allow agent developers to subclass directly from the BaseAgent as indicated in Figure 4.3. Direct subclassing allows the CustomizedAgent to inherit all the functionality of a BaseAgent. However, it violates the encapsulation principles since agent developers can review the implementation details and modify or override methods of the BaseAgent.

4.3 – A direct inheritance from the base agent.

Another approach is to put an abstraction layer between the BaseAgent and the agent developers as indicated in Figure 4.4. An AbstractAgent class is associated with the BaseAgent class and agent developer can only create CustomizedAgent by subclassing the AbstractAgent class. This implementation has merit since it provides good encapsulation and preserves the system integrity. At the same time, the CustomizedAgent still inherits all the functionality of the base agent. However, this way neither prevents the CustomizedAgent from direct access to the BaseAgent nor provides a location transparent feature.

Figure 4.4 – An indirect inheritance from the base agent.

Another alternative implementation is to associate a ProxyAgent with the BaseAgent as depicted in Figure 4.5. The agent developer may create a CustomizedAgent by direct subclassing from the BaseAgent. Since the BaseAgent is associated with a ProxyAgent, the latter prevents the BaseAgent from any direct access and provides a location transparent feature. This approach fails to comply with the encapsulation principle and to

ensure the integrity of the system since the user agent class inherits the BaseAgent directly.

Figure 4.5 – An association between a base agent and a proxy agent.

In summary, neither indirect inheritance nor the proxy agent approach balances all the requirements. The former satisfies encapsulation and preserves system integrity, but fails to provide access control and a location transparent requirement. The latter is the opposite, it satisfies the access control and location transparent feature, but violates encapsulation principles and compromises system integrity.

Solution:

The solution is to combine the abstract and proxy patterns^[42] and construct a generic base agent, as illustrated in Figure 4.6. This approach is to have an **AbstractAgent** class associated with a **ProxyAgent** class. The **CustomizedAgent** class inherits from the **AbstractAgent** and is defined by the agent developer. Use of the abstract class provides good encapsulation to the agent system and ensures its integrity. Other agent developers that use the **CustomizedAgent** will not be able to review any of the implementation detail of the agent system since they can only access the abstraction class. Moreover, since system methods for the common services provided by the agent can be defined as final methods in the abstraction class, agent developers will not be able to modify the final

methods. They can however override non-final methods. The proxy class is also an abstract class and it controls any direct access to an agent. When a malicious agent tries to access an agent, it actually accesses the proxy agent first. Any malicious request will be denied by the proxy agent. The proxy agent also provides location transparency. Using the Internet search agent as an example, when the user interacts with his search agent, it actually interacts with the search agent's proxy. The proxy agent continues interacting with the user while the search agent traverses different locations. This provides the impression that the user is still interacting with his search agent locally.

Figure 4.6 – A combination of the abstract and proxy patterns.

Consequences:

Applying this pattern produces the following consequences:

- + It ensures the integrity of the agent system and prevents hostile agent developers from modifying or overriding the system methods for common services. Your agent system will provide generic base agents for agent developers to build their customized agents and applications. Moreover, your agent system enables agent developers to customize their agents via overriding standard methods.

- + It hides all the implementation details of an agent from other agent developers and it provides a well-defined interface to ease the further agent application developments. This has a great benefit since it provides a platform to allow other applications to be built without worrying about anything below.
- + It provides access controls to the agents. If any other object wishes to access the agent, it must go through the proxy agent first, which applies the security measure.
- + It provides location transparency. When an agent performs a task on behalf of its user in a remote host, the user will have no knowledge of where the agent goes. The proxy agent provides frequent interactions with the user.
- The application of this pattern involves more classes, complicating the implementation. It also requires the implementation of an agent environment (context) that supports services for agents.

Implementation Issues:

The **AbstractAgent** class provides a layer of abstraction to protect the **BaseAgent** from direct subclassing. This is important since the abstraction layer encapsulates the implementation detail of the **BaseAgent**. The **AbstractAgent** class also defines methods that can be overridden in its subclasses by other agent developers. Methods that are not allowed to be overridden include cloning, dispatching, deactivating and disposing. Methods that are allowed to be overridden are necessary hooks to customize the agent behavior such as the **run()** method.

To create a customized agent, one can:

```
public class MyAgent extends AbstractAgent{
```

```
        // put myAgent's methods here.  
    }
```

When a customized agent has been created, it is given its own thread of execution through the system's invocation of its run method. This invocation gives the customized agent a degree of autonomy. The run method is called every time the customized agent is activated in a context. By overriding this method, the agent developer can customize his agent's autonomous behavior.

```
    public void run(){  
        //do something here...  
    }
```

The agent can get its own proxy by using the `Agent.getProxy()` method. The agent proxy acts as the handle to an agent and provides a common way of accessing the agent. Because an agent has several methods that should not be accessed directly from other agents for security reasons, the agent proxy acts as a shield protecting the agent from other malicious agents. When a method is invoked, the agent proxy consults the security policy to determine whether the current execution context is permitted to perform the method. Another important role of the agent proxy is to provide the agent with location transparency.

Figure 4.7 illustrates an example of how the agent proxy provides location transparency to the user. Assume a search agent resides in "host A". The user application passes a request (a message object) of a search job to the agent. Upon receiving a message, the proxy agent does the security check then forwards the message to the agent. To perform a

search job, agent migrates to “host B” but still communicates with its proxy in “host A”. Meanwhile, the user application continuously interacts with the proxy as if it was interacting with the agent.

Figure 4.7: A scenario of location transparent with the Proxy agent.

Code Example:

Both Agent and AgentProxy classes are defined as abstract and they keep a handle to each other. An agent can only be associated with one proxy. When a method of the agent is called, the proxy is called first. This way, the direct access to the agent is prevented. All the methods that are not allowed to be overridden are declared as final methods (the clone method is one such example). However, there are also methods such as run() that

must be overridden in order to customize an agent. Fragments of code samples are given below^[4]:

```
public abstract class Agent {
    /*
     * declare a handler of a proxy.
     * the proxy instance variable is transient to prevent the whole
     * execution context from getting serialized whenever we serialize
     * the agent.
     */
    transient AgentProxy _proxy = null;

    protected Agent() {
    }
    /*
     * Sets the proxy for the agent. This uses the singleton pattern to
     * ensure that proxy can't be set twice.
     */
    public final void setProxy(AgentProxy proxy) {
        if (_proxy != null) {
            throw new SecurityException();
        }
        proxy.setAgent(this);
        _proxy = proxy;
    }

    /* the use of final is to prohibit this method being overridden by
     * user
     */
    public final Object clone() throws CloneNotSupportedException {
        return _proxy.clone();
    }

    /* many other final methods are defined here
     * .....
     * .....
     */
    /* methods that can be overridden are here */
    public void run() {
    }
    /**
     * This method is called once in the lifecycle of an agent.
     * This method is supposed to be overridden in subclasses.
     */
    }
    /* many other methods are defined here
     * .....
     * .....
     */
}

abstract public class AgentProxy {
    /*
```

```

    * Constructs AgentProxy
    */
protected AgentProxy () {

}

/**
 * Gets the agent that the proxy manages.
 */
abstract public Agent AgentProxy() throws InvalidAgentException;

/**
 * Clones the agent proxy. Note that the cloned agent will get
 * activated.
 * If you like to get cloned agent which is not activated, throw
 * ThreadDeath exception in the onClone method.
 */
abstract public Object clone() throws CloneNotSupportedException;

/**
 * many other methods declaration
 */
}

```

Known Uses:

This pattern is used by the IBM Aglet agent framework^[4]. The IBM Aglet consists of AbstractAgent and AgletProxy classes. The AgletProxy provides the access control for the AbstractAgent and both abstract classes support another layer of abstraction to protect the detailed implementation of the agent from being reviewed. The Aglet has a handle to its current context where many services are provided.

This pattern is also used in the Grasshopper agent framework^[5]. The Grasshopper provides two kinds of abstract agents, stationary and mobile agents, for customization. Both stationary and mobile agents inherit from the service class where agent methods are implemented. The user may create his agents by subclassing either stationary or mobile agents.

Related Patterns:

This pattern contains the proxy pattern of the GOF^[42]. It introduces an abstraction characteristic to the proxy pattern in a unique way to construct a generic agent.

Resulting Context:

This pattern constructs agents that rely on the services supported by an agent context; therefore, it requires the use of the Use Service Pattern to structure an agent context.

4.3.2. Use Service Pattern**Context:**

After having built a base agent, you need to construct an agent environment from which agents can access services such as execution, communication and transportation. The agents should be able to access these services not only from the local host, but also from foreign hosts. To structure such an agent environment, the following must be considered:

- ❑ **Management ability:** the agent environment should provide registration and security management for the agent framework, and facilitate an agent's creation, activation, termination and services access. The agent environment registers an agent when it arrives or is created, de-registers an agent when it is terminated or dispatched, and applies security measures to ensure the safety of the agent framework.
- ❑ **Inter-operability:** the agent must be able to execute in different hosts. In other words, the foreign hosts must either support the same execution environment as the home host or provide a transportation service to receive the incoming agents.
- ❑ **Available services:** it is important that an agent can access services that are provided locally rather than obtain execution code from a remote host. This permits an agent to

be a reasonably small, which reduces network transmission overhead cost if its migration is required.

Problem:

How do you structure an agent environment that allows agents to access services from both home and remote hosts?

Forces:

One approach is to have the base agent associated with the services, which implies that the agent can access the services directly. For example, to create a child agent, the parent agent can access the creation method of the execution services via the association. This association appears simple but there is no agent management ability. Moreover, when it migrates, the agent has to migrate with services due to the strong agent-service association, which will incur a high network transportation overhead. Also, this approach does not support the inter-operation between hosts since some services that the agent brought with it may not be considered as legal operations in the foreign host.

Another approach is to have the base agent associated with the agent context (or agent environment). This structure provides a degree of service management for the agent. When an agent migrates to a foreign host, only the agent context is transported with the agent. In this approach one agent requires one agent context. The agent context does not provide any agent management measures. Also, if an agent requires access to a service, the source code for the service has to be loaded from the home host. This may cause response time delays while the agent is waiting for the service code, especially when the network traffic is heavy.

Apparently, both approaches fail to support inter-operation since there is no common interface among the network hosts. Without the common interface, agents will not be able to use services provided by the foreign hosts.

Solution:

The solution is to provide an agent execution context with a common interface that allows agents to inquire or access its existing services in a common way. This approach permits an agent to rely on the services provided by the context rather than on its own. An agent lives in the context and keeps a handle to the context. When an agent migrates to a foreign host, it will be received by the foreign context of that host. The agent that resides in the foreign context can obtain necessary services locally via the common interface.

Figure 4.8 is a class diagram of the Use Service Pattern in which the agent class is associated with a context interface. The context interface is defined commonly among all the hosts and the agent context that implements the interface can be specific to an individual host. This structure allows a host to have more flexibility. For example, a host can implement security measures that are different from other hosts. To obtain a service, the agent can get a list of the services provided from the context, then request a particular service. If a security check on the agent is passed, the agent context returns a handle to the requested service to the agent as shown in Figure 4.9.

Figure 4.8: Class diagram of agent/context association.

Figure 4.9: A sequence diagram describing how an agent requires a service via the agent context.

Consequences:

Applying this pattern has the following consequences:

- + The agent context provides a means of agent management and maintenance. When agents arrive from other hosts or are created at a host, the agent context registers all the agents and monitors their execution status. When agents terminate, the agent context removes their registration status.
- + The agent context facilitates the security measures in the host. When a mobile agent arrives, the agent context receives it and security policies can be applied to examine the arrival agent. If the new arrival agent threatens the security of the host, the agent context can terminate the execution of the hostile agent.
- + The agent context provides services to agents that are active in the context. When an agent executes services in a foreign host, it can use services locally and avoid loading classes from its home host. This decreases the network overhead. Since agents rely on the services provided by the context, the size of agent can be kept small. When agents migrate, they do not need to carry or load required classes from the home host.
- + It allows an agent to change its execution plan dynamically. For example, a mobile agent can pick up different execution plans when it arrives at different hosts.
- + The context interface defines common methods but leaves the implementation details to individual host, which provides more flexibility to the agent context implementation.
- It requires the implementation of an agent context and a common interface, which may increase the complexity of the agent system.

Implementation Issues:

Construction of agent context: The agent context is an environment where agents spend most of their lives. Agents are created and destroyed in the context. When an agent travels in a network, it migrates from one context to the other. The context is a uniform execution environment that provides fundamental services such as communication, management, transport and security. Policies must be implemented to ensure smooth operation of each service.

Construction of context interface: The context interface represents the common services that are provided by all the agent hosts so that the agent can be executed in different hosts. A proxy object can implement context interface to prevent the agent context from being accessed directly by agents. An agent can use the agent context to get information about the environment where it resides.

Use of services: The agent class is associated with a context and it has a method for gaining access to its current context:

```
context = getAgentContext();
```

With access to the context, the agent can access services provided by the context. For example, to create a new agent, the following code is used:

```
context.createAgent(childAgentName);
```

Code Example:

The following code segment is a brief sample of the agent context interface. It shows methods that cope with agent management, basic agent operations and context service accessing^[4].

```

public interface AgentContextInterface{
    /**
     * Gets the agent proxies in the current execution context.
     */
    public Enumeration getAgentProxies();

    /**
     * Gets the proxy for an agent specified by its identity.
     */
    public AgentProxy getAgentProxy(AgentIdentifier identity);

    /**
     * Gets the context service indicated by the key.
     */
    public String getService (String key);

    /**
     * Gets the context service indicated by the key and default
     * value.
     */
    public String getService (String key, String def);

    /**
     * Enumerates all the service keys.
     */
    public Enumeration serviceKeys();

    /**
     * Creates an instance of the specified agent located at the
     * specified URL.
     */
    public AgentProxy createAgent(URL url, String name,
        Arguments args) throws IOException, AgentException;

    /**
     * Retracts the Agent specified by its url:
     */
    public AgentProxy retractAgent(URL url) throws
        IOException, AgentException;

    /**
     * Activate an agent. This is a forced activation of a deactivated
     * agent.
     */
    public AgentProxy activateAgent(AgentIdentifier aid)
        throws AgentNotFoundException;

    /**
     * Returns the URL of the daemon serving all current execution
     * contexts.
     */
    public URL getHostingURL();

    /**

```

```

        * other methods etc.
        */
    }

    public class AgentContext implements AgentContextInterface {

        /**
         * implementation of getAgentProxies method
         */
        public Enumeration getAgentProxies()
        {
            //detail implementation of the method;
        }

        /**
         * other detail implementation of methods defined in the context
         interface.
         */
    }
}

```

Known Uses:

This pattern is used by the IBM Aglet agent framework^[4], which implements a context and a context interface to provide different services. The Aglet has a handle to its current context through which agent system services are accessed.

The Grasshopper agent framework^[5] also used this pattern. The Grasshopper provides a Service class to allow stationary and mobile agents to access system services via subclassing.

Related Pattern:

This pattern is closely associated with the Agent Abstraction Pattern by Silva and Delgado[54].

Resulting Context:

This pattern provides agents an environment to services access and management. Applying this pattern will require the use of the Service-Based Implementation Pattern that categorizes policies according to their association with the services.

4.3.3. Service-Based Implementation Pattern

Context:

Suppose that you are constructing a multi-agent system and you have structured it with the Use Service Pattern. The system contains an agent context and many services. Agents live in the agent context and they are managed by it. The most common services that your agent context needs to provide include the following:

- ❑ **Communication Service:** this service is responsible for all remote interactions that take place between the distributed components.
- ❑ **Registration Service:** each agent context must be able to know about all currently hosted agents.
- ❑ **Management Service:** This service permits the monitoring and controlling of agents of an agency by external users.
- ❑ **Transportation Service:** this service supports the migration of agents from one agency to another. At the destination agency, the agent continues its task exactly at the point where it has been interrupted before the migration.
- ❑ **Security Service:** this includes both external and internal security to protect the host from hostile and illegal remote interactions, and to protect agency resources from unauthorized access by agents.

- ❑ **Persistence Service:** this service enables the storage of agents, e.g., the internal information maintained inside an agent on a persistent medium. In this way it is possible to recover agents when needed.

Each of the above services must be guarded by corresponding policies. In order to provide these services, you need to construct policy sets to define the prerequisites for each service. If an agent requires a communication service, a set of policies is required to define how the communication takes place and how a common language is applied. If there are security policies to apply, policies must be defined on agent to agent and agent to host interactions to prevent both agent and host from hostilities. For example, policies must be placed for an agent to prevent confidential data that it carries from being reviewed by other agents; whereas the host must have policies implemented to stop hostile agents from overusing system resources. One must consider the following when structuring policies.

- ❑ **Scalability:** policies must be able to grow. The policy structure must permit adding of new policies, deleting or replacing of existing policies.
- ❑ **Priority:** policies should have a priority and the structure must have a mechanism for solving conflicts.
- ❑ **Retrieval efficiency:** the policy structure must facilitate rapid access of policies. When an agent utilizes a service, policy lookup should be completed within a reasonably amount of time.

- ❑ **Applicable Scope:** a policy must be associated with an application scope, which draws boundaries of where the policy is applicable.

The management of the services becomes the management of service policies and raises the issue of how various policies are structured and categorized.

Problem:

How do you structure and categorize policies to facilitate service management?

Forces:

We can structure all the policies together in a policy pool as indicated in Figure 4.10. The policy pool can be an object or a table containing if-then policies. All the services are associated with the same policy pool. This structure is simple since there is no structuring or categorizing of policies. A policy is identified by its name (such name must be unique). However, this structure is not scalable. For example, to remove a policy, we need to find the policy first and set the corresponding entry to null. To add a new policy, we have to search for an empty entry. If an empty entry is found, we add the new policy to that entry otherwise we add the new policy to the end of the table (assuming the policy pool is to be structured by a table). These operations will become inefficient when the policy pool table is large. Moreover, the policy pool will likely have to handle priority conflict since some of the policies may have the same priority. A simple policy pool structure will obviously not be sufficient since there is always a potential of misuse of a policy in a wrong service.

Figure 4.10: A simple policy pool implementation.

Solution:

The solution is to categorize policies into services as indicated in Figure 4.11. This means that policies are grouped according to their association with services. The service is the applicable scope of the policies within it. The growth or alternation of policies is done within the service. An example of this kind of policy structure is security service. External and internal security policies are structured and managed within the scope of security service.

Figure 4.11: A class diagram of a service-based implementation pattern.

Consequences:

The categorization of policies according to services will have the following consequences:

- + It clearly defines the policy applicable scope. Since each policy set is defined within a particular service scope, there will be no confusion or misuse of policies. Policies for different services are separate entities. For example, communication policy and security policies are structured independently.
- + It provides a scalable structure for policies. This pattern suggests that the service is associated with a policy pool that defines how the service is provisioning. This association allows the policy pools for different service to grow independently. When

adding a new, or removing an existing policy to a service, only the corresponding policy pool is subjected to modification. Such a structure will facilitate the service policy management.

- + It avoids the possible priority conflict since priority is defined for policies in the same service policy pool. Resolving policy conflict can be done within the same service policy pool. If it is a conflict happening between different policy pools that belong to different services, the conflict can be solved in the service level.
- + It allows an agent to retrieve policies effectively. The independent structure of service policy pool allows fast policy access because the search is service orientated. When a security policy is retrieved, the search is made only in the security policy pool.
- It requires good categorization techniques and experience to associate a policy to the correct service.

Implementation Issues:

This pattern provides the first step to structuring effective policies for an agent system. It suggests a “divide and conquer” strategy. The division of policies starts at the service level in a vertical manner. To further decouple the service and service policy, a policy manager can be set up to facilitate the policy management. However, to implement policies even for a service could be very complicated and sometimes, different layers of policies may be engaged.

Known Uses:

The Grasshopper agent framework^[5] partitions its services into various domains including communication, management, persistence, registration security and transport. Policies are

implemented within each domain. For example, in the security domain, policies of confidentiality, integrity, authentication, access control and auditing are implemented.

Voyager agent frameworks^[6] groups its services through the use of this pattern. Services include communication, Messaging, Naming, Directory and Gateway. Other utilities are also provided.

Related Pattern:

This pattern relates to the Whole-Part pattern^[48] where the Whole encapsulates its constituent components (the Part), organizes their collaboration, and provides a common interface to its functionality.

4.3.4. Layered Structure Pattern

Context:

Agents can be constructed in layers^[51-53]. The higher level or more sophisticated behavior depends on lower level capabilities. Information flows only between neighboring layers.

One known example is the seven-layer model suggested by Kendall and Malkoun^[51]:

- ❑ Sensory Layer gathers sensory input from the agent's environment.
- ❑ Beliefs Layer models the agent and its environment.
- ❑ Reasoning Layer determines what an agent should do next based on its goals, plans, capabilities and modes.
- ❑ Action Layer carries out the agent's intentions derived from incoming requests that have been accepted by the collaboration layer.

- ❑ Collaboration Layer determines how to collaborate with other agents. This includes accepting or rejecting an incoming request for services.
- ❑ Translation Layer formulates a message for another agent, translating it into another language or semantics.
- ❑ Mobility Layer performs message transmission and reception for an agent.

Suppose that you are structuring policies for an agent constructed using the layered structure. The behavior of each layer of the agent and the information exchanged between layers must be defined by corresponding policies. Using the Sensory layer as an example, it must have policy to determine what and how to sense the agent's context environment. Similarly, policies must be provided for other layers. Other policy requirements that need to be addressed within the layers include the following:

- ❑ Application Scope: policies structured for a layer must be used within the corresponding layer. For example, if a policy is structured for the Sensory layer, its validation only remains within the Sensory layer.
- ❑ Association: The layer should have a clear awareness of the existence of the associated policy. However, this association should not be a strong coupling between the layer and the actual policy structure. In other words, the policy structure should be relatively independent to facilitate policy management.
- ❑ Retrieval Speed: policy retrieval must be fast. Agents offer operation in real-time and must complete certain tasks within short time constraint.
- ❑ Encapsulation: the policy details for a layer must be hidden and independent from the policies of other layers.

- Conflict Resolving: the mechanism for resolving policy conflict between layers should be simple and efficient.

Problem:

- How do you structure policies to accommodate the layered structure in an agent?

Forces:

A simple way to structure policies for an agent is to have an agent policy pool (Figure 4.12). An agent-policy pool puts all policies into a large pool regardless of the application scope of each policy. As depicted in Figure 4.12, all the policies are contained in the AgentPolicyPool class. Such a structure shares the beauty of simplicity and policy retrieval speed is fast if there are a small number of policies in the pool. However, such an structure does not provide a direct layer object-policy relation. The retrieval performance will decrease if there are many policies. This is because when a layer object requires accessing one policy, it must search through the AgentPolicyPool object. The structure also violates encapsulation principle since the policies for different layered objects reside in the same policy object.

Figure 4.12: A simple agent - policy pool association.

Another structure is to allow the **BaseAgent** to contain all the policy pools for layer objects as shown in Figure 4.13. Policy pools for different layers will be structured separately. This approach does provide good encapsulation between the policy pools for different layers and it also defines a policy's application scope. However, it does not allow the layer to have direct access to its policy pool. For example, if the Sensory object wants to access a policy, it has to go through the **BaseAgent** to access the corresponding policy pool. Clearly, such a structure is not compatible with the layered structure of the agent.

Figure 4.13: An agent-policy pools structure.

Solution:

The solution is to use a layered structure for policy pools, which are associated directly with the layer objects within the agent. Such a layered structure is illustrated in Figure 4.14. Each layer of the agent is directly associated with a policy pool object that defines the behavior, service, interaction and security of that layer. For example, the Sensory layer is associated with the **SensoryPolicyPool** class in which the sensing plans are

defined. The layered policy pool objects define mechanisms for solving conflicts within their layered objects.

Figure 4.14: A layered Structure of policy pool structure in an agent.

Consequences:

Using the policy layered structure pattern has the following consequences:

- + It is compatible with the agent layered structure. If you structure your agent with a layered structure, then it is an advantage to structure policy pools into layers because each layer owns its policies and each policy has a clear application scope.
- + It allows the layers to access the corresponding policy objects directly and rapidly. Since this pattern provides a one-to-one association between the layer and its corresponding policy object, the layer can access its policies without going through the agent.
- + It provides a scaleable policy structure. Since the policy structure for a layer is relatively independent, it facilitates the growth of the policy. When adding, modifying or removing a policy from a layer, only the related policy class is affected and it does not alter the policy structure in other layers.

- + The structure supports conflict-resolving mechanisms within a layer. This is because one can define conflict-resolving methods in each policy pool.
- The implementation of the structure involves more classes and relations that may compromise the performance efficiency.

Implementation Issues:

This pattern provides a horizontal view of policy partition. The number of layers in the policy implementation depends on the layered structure of the agent. A seven-layer model has been discussed in a number of papers^[51-52], but it is not mandatory. For example, a Mitel agent has a four-layer structure^[53].

If the policy conflict-resolving mechanism is simple, it is convenient to implement the mechanism inside the policy pool object. However, if the mechanism is large and complicated, then it is better to separate it from the policy pool object as indicated in Figure 4.15.

Figure 4.15: A separated policy conflict-solving structure.

The actual implementation of each layer's policy can use the Agent Policy Delegation Pattern and/or Hashtable Pattern.

Known Uses:

The layered partitioning of an agent and agent policy approach has been applied in various applications. The Mitel Agent Framework was implemented with four layers including network, services logic negotiation and resource layers, each of which is associated with its own policies^[53].

The Layered Structure Pattern was also observed in Kendall and Malkoun's RMIT agent framework where seven-layer agent structure was used and each layer was associated with a knowledge set^[51].

Related Pattern:

This pattern is related to the Layers Pattern by Buschmann et al.^[44]. The Layers Pattern is for applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

4.3.5. Agent Policy Delegation Pattern**Context:**

Agents and multi-agent systems can span several levels of abstraction. Using a layer structure appears to be an effective approach to encompass autonomous, social, reactive and proactive behavior of an agent. Each layer has its distinguished responsibilities and behavior that are defined by policies associated with the layer.

Suppose that you have constructed a multi-agent system using a layered architecture and you have also partitioned the policies using the Layered Structure Pattern described previously. Now, you want to structure policies for each layer. In most cases, there is

going to be more than one policy set associated with a layer. Using the Sensory layer as an example, there may be sensing plans A, B and C required by your agent as it moves from one destination to another. Each sensing plan contains policies that instruct the Sensory object what to sense. New policies may be added to the existing plan or existing policies may be modified at runtime. In the structure of policies for each layer, the following must be considered:

- ❑ Decoupling: a strong dependency between the layer and the concrete policy implementation should be avoided since a strong coupling to a specific policy will make the agent less portable and dynamic in terms of changing policies.
- ❑ Substitution at runtime: a change of policy is essential at runtime. For example, the Sensory layer should be able to change sensing plans at runtime.
- ❑ Scalability and manageability: the structure should allow new policies to be added and should provide an efficient way to manage existing policies for layer objects.

Problem:

How do you structure different policy sets for an agent that has a layer structure?

Forces:

One approach is to embed policies inside the layered objects. For example, we can put sensing plans inside the sensory object. This approach makes policy structuring easy, but it introduces a strong binding between the Sensory object and the sensing policies. As a result, it will make sensing policies hard to manage, to scale and to change at runtime. For example, to add new sensing policy, we need to modify the Sensory object. In an extreme case, the layered object may be big that makes the agent less portable (in the

case of migrating the agent to a foreign host), as well as makes the layered policies less manageable.

Another structure is to have a policy pool class aggregated inside the LayerObject class (Figure 4.16). This structure is also considered simple and it provides a relatively loose coupling between the LayerObject and the PolicyPool object. However, there exist some potential problems. It makes policies hard to manage since there is no boundary between different policy sets. Also, this structure does not support policy changes at runtime.

Figure 4.16: An association between the LayerObject and PolicyPool class.

Solution:

The solution is to further decouple the dependency between the layered object and the policy pool object by introducing a containment relation. A class diagram of the Agent Policy Delegation Pattern is shown in Figure 4.17. This design pattern not only decouples the layered object and its corresponding policy pool object, but also provides independence among different sets of policies. As depicted in Figure 4.17, the LayerObject is associated with a PolicyPool that contains various PolicySet objects. A clear separation between the LayerObject and PolicyPool allows various concrete PolicySets to be structured independently.

Figure 4.17: A class diagram of the Agent Policy Delegation Pattern.

Consequences:

The application of the agent policy delegation pattern will result in the following consequences:

- + It provides a flexible policy structure. This pattern disjoins the tight coupling between the layer object and its corresponding policies. Such a loose coupling provides room for dynamic changes to policies. An agent can change its behavior by executing new policies obtained from a new host.
- + It provides a scaleable solution. Since policies are structured independently, it facilitates policy modification. For example, adding, modifying or removing a policy affects only policy set classes.
- It introduces a more complicated class structure and increases to the class numbers, which may compromise performance.

Implementation Issues:

There are different approaches related to how layered objects inside an agent can obtain the required policy.

A layered object can make a request explicitly to its policy pool object to obtain a copy of the required policy object, for example, through `newPolicySet = PolicyPool.getPolicySet(policySetName)`.

Another way is that the layered object uses the policy service by passing itself to the policy manager (through `PolicyPool.applyPolicy(layerObject, policySetName)`). In this case, the layered object does not keep a copy of the policy that it requires.

Sample Code:

The `LayerObject` class has a handle to the `PolicyPool` that contains many `PolicySet` classes. The `PolicySet` is a policy class in another level of details. When the `LayerObject` wants to access `PolicySetA`, it simply gets the required policy set through `policySet = PolicyPool.getPolicySet(A)`.

```
public class PolicyPool
{
    protected List policySet;
    public PolicyPool();

    // methods are defined here;
    public getPolicySet(PolicySetName);
    //....
}
public class PolicySet
{
    //private attributes
    String PolicySetName
    //constructor
    public PolicySet(name){policySetName = name;}
    //other methods
}

public class LayerObject
{
    private PolicyPool policyPool;
```

```

        private PolicySet policySet
        public LayerObject(){}
        //get a policy set
        policyPool = new PolicyPool();
        policySet = policyPool.getPolicySet(A);
        //others .....
    }

```

Known Uses:

This pattern is used by the RMIT agent framework^[51]. An example of it is in the implementation of reasoning plans for the reasoning layer object. Another example is in the implementation of contract policy selection for the resource layer of the Mitel agent framework^[53].

Related Pattern:

This pattern is similar to the Strategy Pattern described by Gamma et. al.^[46].

4.3.6. Hashtable Pattern

Context:

Suppose that you are implementing a policy selection mechanism. A policy is selected using a predicate or a policy name in a simple case. For instance, if the predicate is "A", then policy "A" will be selected. In such a policy mechanism, the followings issues are of are concern:

Strong Association: A strong association relation between the predicate (policy name) and policy must be embedded in the policy database.

Access Speed: Fast access to the policy database is essential. The policy database must be structure to allow agents to retrieve a policy quickly.

Scalability: the policy database structure must be scaleable and must allow the addition of new policies and the deletion of existing policies at runtime.

Problem:

How do you organize policies so that efficient access can be achieved?

Forces:

An obvious structure to use is tables. Predicates (policy names) and corresponding policies can be placed in the columns within the same row and different policies can be placed in different rows. This implementation provides a strong relation between predicates and policies. When an agent retrieves a policy, a linear search of the policy table is performed. A new row can be created if a new policy is to be added; an existing row can be deleted if a policy is to be removed. Multiple columns of predicates can be arranged in the same table. This implementation is simple. However, its performance will decrease if the table is large. This is because linear search becomes ineffective in a large table.

Another alternative is to implement policies using if... then... statements. For example, assuming a user agent makes an inquiry to a book broker agent, the latter agent may return the lowest price or earliest deliverable date according to the desire of the former. The policy implementation in this case using can be expressed as follows:

If price, then apply lowest price policy;

If deliverable date, then apply earliest deliverable date policy;

.....

The if... then... statements method is easy to implement. However, similar to the table approach, it has the disadvantages of policy retrieval, scalability and maintainability when the number of policies becomes large. This is due to the fact that search for policies is done in a linear way.

Solution:

The solution is to use a hashtable structure for the storing different policies as shown in Figure 4.18. A `PolicyActionInterface` is defined (for simplicity, only one method called `applyPolicy(String policy_key)` is defined). All the `Policy` classes implement the interface. Also, the `PolicyHashTable` class that stores `ConcretePolicyAction` classes is associated with the `PolicyActionInterface`. The `ConcretePolicyAction` classes are accessible through the `PolicyHashTable`. The `PolicyHashTable` object is responsible for associating the policy-required object to the corresponding `ConcretePolicyAction`. When a call to the `applyPolicy(policy_key)` method is invoked, the policy-key is passed to the `PolicyHashTable` which compares the policy keys that are already stored in the hashtable. If it is a valid key, a `ConcretePolicyAction` object is returned as indicated in Figure 4.19.

Figure 4.18: A class diagram for the Hashtable Pattern.

Figure 4.19: A sequence diagram for the Hashtable Pattern.

Consequences:

Application of this pattern has the following consequences:

- + It is a scalable solution since new policies can be easily added to the corresponding policy action class without a serious impact on the entire policy database. When a policy action class gets too large, it can be split into two and simply registers the new policy action into the hashtable.
- + It facilitates policy management and maintenance. This is because all the policies can be managed and maintained in the policy class and there is a strong association between the policy key (the predicates) and the policy. Each policy action can be treated as an independent object.
- + It provides fast retrieval of a policy since a hashtable searching is much more efficient than linear searching.
- It involves a more complicated class structure. This means an increase in complexity in both system design and code implementation.

Implementation Issues:

A hashtable data structure associates key objects with value objects. This allows the associated value objects to be found quickly by knowing the key objects. In the actual hashtable structure, it is generally possible to have more hash values than places in the data structure. Hashtable algorithms resolve this by associating each place in the data structure with multiple hashed values. This means multiple objects may be stored in the same place in a hashtable. When this happens it is called a collision. It is impossible to prevent collisions in a hashtable. However, one can arrange to handle this situation.

Code Example:

The following code fragments illustrate a small part of the Hashtable pattern implementation. The ConcretePolicyAction implements the PolicyActionInterface and it is the class where the concrete policy is implemented.

```
public class ConcretePolicyAction implements PolicyActionInterface
{
    String policyName ;
    /*
    concrete policy implementation is defined here
    */
}

public class PolicyHashTable
{
    /*
    a hashtable to store all the policy objects
    */
    Hashtable variableList ;

    /*
    methods to add, delete and access a policy are defined here
    */
}

public interface PolicyActionInterface
{
    applyPolicy(String key);

    /*
    other method prototypes;
    */
}
```

Known uses:

This pattern is commonly used in many agent framework implementations. It can be found in IBM aglet system^[4], Grasshopper agent system^[5] and JAFMAS^[7].

Related Pattern:

This pattern is similar to the Hashed Adapter Pattern described in Grand's book "Patterns in Java" Vol.2^[46]

4.4. SUMMARY

In this Chapter, six agent policy patterns, e.g., Agent Abstraction Pattern, Use Service Pattern, Service-Based Implementation Pattern, Layered Structure Pattern, Agent Policy Delegation Pattern and Hashtable Pattern were introduced. These patterns offer a way to structure policies for an agent environment. In Chapter 5, the use of these patterns in the high-level design of an agent-based application will be discussed.

CHAPTER 5: AGENT POLICY PATTERN APPLICATION

5.1. INTRODUCTION

This chapter demonstrates the use of the six agent policy patterns in the design of the Distributed Book Ordering System (DBOS). The DBOS is an agent-based application that consists of a GUI, a book ordering Broker and remote Servers. An agent, created by a user through the GUI, can interact with the DBOS to carry out tasks on behalf of the user. For example, a user agent can make an inquiry to the DBOS broker agent. The broker agent accepts the inquiry and dispatches a query agent to traverse all the remote bookstores (servers). The query agent returns the query results including the book availability, price and deliverable date. If the book is found at multiple locations, the Broker agent may return the lowest price or earliest deliverable date according to the user's desire. The overall architecture of the DBOS is given in Figure 5.1.

Figure 5.1: The architecture of the DBOS.

We could implement DBOS directly starting from Java RMI or CORBA. However, this implementation is not a most efficient way if other distributed applications must be built.

The idea here is to build the application on top of a distributed agent environment that provides the common mobility service as indicated in Figure 5.2. In the following sections, we will introduce the design of the DBOS system and the distributed agent environment.

The objective of the chapter is to illustrate the use of agent policy patterns with a simple example. Therefore, many detail of the implementation of DBOS will not be discussed. We focus the discussions on an overall design of both DBOS and the distributed agent environment, as well as where and how the agent policy patterns are applied.

Figure 5.2: Layer structure of application-agent framework.

5.2. REQUIREMENTS

The DBOS must handle the following requirements:

- The user is allowed to order a book by author, title or ISBN using a graphic user interface (GUI). Users may submit queries concurrently.

- ❑ Bookstores (served as the remote servers) have their own database containing book inventory information.
- ❑ The DBOS Broker agent accepts a book inquiry, determines the book availability at distributed bookstores (remote servers) by dispatching query agents to different bookstores.
- ❑ The query agent returns with a message containing the searching result, such as the book price and deliverable date to the broker agent. If the search fails, then a "not found" message is returned.
- ❑ The Broker agent presents the searching result to the user agent after receiving a message from the query agent. If a book is found at multiple locations, the DBOS Broker agent returns the best price or earliest deliverable date to the user agent.

5.3. SCENARIO

In this application, we consider two basic scenarios: single user book inquiry and multi-user book inquiries. The latter is particularly important since it involves multi-agent processing and synchronization issues.

Single User Book Inquiry:

A user is allowed to create a user agent using the DBOS's GUI and to create a book inquiry, which includes the submission of parameters such as the book title. When an inquiry is submitted, the user agent contacts the DBOS agent who accepts the query and creates a query agent. The latter will carry out the search and return the query result, i.e., the book price, deliverable date and available quantity of the book. The broker agent, in accordance with its policy, returns the best-fit information to the user agent.

Multi-user Book Inquiries:

Multiple users are allowed to create agents using the DBOS's GUI concurrently. The broker agent handles all user agents in the same way as in the case of single user book inquiry. To avoid getting into any race conditions, a query queue is used to implement a first-come first-served policy.

5.4. HIGH-LEVEL INTERACTION DIAGRAM

Figure 5.3 illustrates how the external world interacts with the system, as well as the system's internal interactions between the modulus. In the DBOS, the users interact with the system by creating their user agents. For example, when a user makes an inquiry of a book, he creates a user agent and instructs it which book to locate. The user agent communicates with the broker agent by sending a request message. The Broker agent responds to the user agent and dispatches a query agent. The query agent migrates to all book servers to conduct a database search, and returns the search results to the broker agent. The broker agent selects the best price or earliest delivery date and returns the information to the user agent.

Figure 5.3: High-level interaction diagram of DBOS.

Figure 5. 4: A class diagram of DBOS

5.5. CLASS DIAGRAM OF DBOS

A class diagram describes the types of objects in the system and various kinds of static relationships that exist among them. The DBOS design involves about fifteen classes (classes related to RMI are not shown here), which are distributed in the GUI, Broker and Remote Server packages. An overall class diagram of DBOS is given in Figure 5.4.

5.5.1. GUI

There are five classes in the GUI package including the UserApplet, UserAgent, Inquiry, Order and Payment classes as indicated in Figure 5.5.

Figure 5.5: Class diagram of GUI module.

The UserApplet is an interface allowing users to access the system and to create user agents. The UserAgent instance can accept query, order or payment input from the user and can carry out a user's task. The Inquiry, Order and Payment are objects holding user's request information.

5.5.2. Book Ordering Broker Agent

The broker agent listens to the user agents and processes in coming requests. When an inquiry is received, the broker agent creates and dispatches a query agent to search remote servers. Figure 5.6 shows three classes in Book Ordering Broker Package, including BrokerAgent, QueryAgent and OrderAgent (for simplicity classes relevant to distributed agent environment and policy implementation are shown).

Figure 5.6: Class diagram of Broker module.

5.5.3. Database Servers (Bookstores)

The database module contains seven classes to handle the information on books, such as title, price, quantity and deliverable date as shown in Figure 5.7. BookStoreAgent

receives query agents and examines their identities. Once the query agent passes the security check, it can access the BookInventory by invoking the latter's book search methods. The Book class represents book, which has attributes including bookName and price. A book also has a state that indicates whether a book is available, on-hold or out-of-stock. BookState class is an abstract class, which is inherited by classes Available, OnHold and OutOfStock. The returnState() returns a BookState object.

Figure 5.7: Class diagram of server module.

5.6. DESIGN APPROACH USING AGENT POLICY PATTERNS

The key functions of DBOS are that the user agent must be able to send message to the broker agent and the query agent must be able to traverse all the remote servers. These functions are supported by the distributed agent environment. In this section, we provide

a high-level design picture of how to implement the distributed agent environment using the agent policy patterns introduced in the previous chapter. Since the purpose of this chapter is to demonstrate the use of agent policy patterns, we will not describe all the features of a mobile agent system. Instead, our discussions will be focused on how to apply the agent policy patterns in the process of designing a distributed agent environment.

Figure 5.8 depicts an agent policy pattern language for constructing a distributed agent environment. The Agent Pattern^[50] provides an overall view of an agent system and it consists of three sub-systems: agent, agent context and system services. The Agent Layer Structure Pattern^[51-52] and Agent Abstraction Pattern can be applied to implement the agent sub-system. The Use Service Pattern and the Service-Based Implementation Pattern can be used in agent context and service sub-systems, respectively. The Layered Structure Pattern, Agent Policy Delegation Pattern and Hashtable Pattern can be applied for more detailed policy implementations.

Figure 5.8: Design of the distributed agent environment using pattern language.

5.7. THE AGENT PATTERN

A high-level design diagram of the distributed agent environment using the Agent Pattern language is given in Figure 5.9. Package "A" is the agent sub-system, "B" is the agent context and "C" is the services that provided by the distributed agent environment. A brief description of each component is given below:

Package A: this package consists of Agent View, Base Agent and Abstract Agent. The Agent View provides user an interface to monitor his agents. The Base Agent is protected by the Abstract Agent. The latter allows agent developers to customize their agents by subclassing. The Base Agent is associated with the agent context where it obtains services provided by the agent system. The Agent Layer Structure Pattern and Abstract Agent Pattern are used to structure this sub-system.

Package B: The System View is an interface that allows interactions between users and the agent system, through which users can cerate agents. The Context is an execution environment where user agents live and execute their tasks. The Context contains various services provided by the agent system including registry, security, mobility and execution. This part is implemented using Use Service Pattern.

Package C: this package includes the implementation of common services that the agent environment provides, such as registration, security, mobility and agent execution. The Service-Based Implementation Pattern is applied to allow services to be implemented independently.

Figure 5.9: A high-level design of the distributed agent environment using Agent Pattern.

Based on the Agent Pattern, a simplified class diagram of the distributed agent environment is constructed as shown in Figure 5.10. The arrows represent a relationship existing between the two blocks. The descriptions of each class are provided as follows:

AgentView Class: this class provides an interface that represents the user agent through which user can pass instructions to his agent to perform either query or order.

BaseAgent Class: this class contains the implementation of agent methods including accesses to agent system services. In our example, this class is structured using the Agent Layer Structure Pattern.

ProxyAgent Class: this class provides access control for the agent and it also handles the interactions between the user and his agent.

AbstractAgent Class: this class introduces a layer of abstraction to the BaseAgent so that it encapsulates the details of BaseAgent class. With the ProxyAgent class together, they implement the Agent Abstraction Pattern.

SystemView Class: this class provides an interface that allows user interaction with the system. Users can create his agents within this interface.

ContextInterface Class: this is an interface that is implemented by the AgentContext. Such an interface is important since it defines common methods for agent context in different locations.

AgentContext Class: this class implements all the methods defined in the ContextInterface. It also provides an execution and management environment for agents. This class contains many services that are used to maintain the agent system's normal operations. Agents can obtain services from the AgentContext.

Services Class: this class represents many service classes that are required by the agent context.

BaseAgent, AbstractAgent, ProxyAgent and Agentview classes are the main elements of the agent package and ContextInterface and AgentContext classes reside the agent context package. The Service class symbolically represents the agent services.

Figure 5.10: A class diagram of the distributed agent environment.

5.8. AGENT PACKAGE--LAYER STRUCTURE AND AGENT ABSTRACTION PATTERNS

The agent package contains **BaseAgent** and **AbstractAgent**. The **BaseAgent** construction includes the detail design of an agent and its associations with the agent context package. The Layer Structure Pattern is used to structure the **BaseAgent**. The **AbstractAgent** encapsulates the **BaseAgent** insides from the agent developers and also provides a generic

base from which the agent developers can build their application agents. The Agent Abstraction Pattern is applied to construct the AbstractAgent.

5.8.1. Layer Structure Patterns

The Kandall's Agent Layer Structure Pattern^[51-52] is applied in the design of the base agent. As indicated in Figure 5.11, the base agent consists of four layers including Mobility, Action, Beliefs and Sensory layers. The agent policy implementation for each layer uses the Layer Policy Structure Pattern, e.g., each layer is associated with its corresponding policy.

Figure 5.11: The use of Layer Structure Pattern in base agent design.

Assuming the user agent makes a payment to the broker agent using a credit card, the Sensory Layer object of the broker agent reads in the credit card information including user name, card number, card type and expiration date. The SensoryPolicy then examines whether the information is correct. If it is, the information will then be passed to Beliefs layer for further processing, otherwise the information will be rejected and the user agent will be asked re-enter information. In the Beliefs Layer, BeliefsPolicy examines possible

mismatch of card type and card number. If no fault is found, the Action Layer will handle the payment and an order agent will be dispatched to corresponding location (book server) to update the inventory and to confirm the success of the order.

5.8.2. Agent Abstraction Pattern

The base agent is developed as a generic agent so that other agent developers can subclass it to build customized agents. In order to encapsulate the details of the BaseAgent, the Agent Abstraction Pattern is applied as shown in Figure 5.12. The AbstractAgent class and the AgentProxy class provide a wrapper around the BaseAgent, and customized agents can be constructed by subclassing of the AbstractAgent. In DBOS, there are three customized agents: UserAgent, BrokerAgent and QueryAgent.

Figure 5.12: A use of the Agent Abstraction Pattern in generic agent design.

5.9. AGENT CONTEXT PACKAGE -- USE SERVICE PATTERN

This package is designed and implemented using the Use Service Pattern, which help to provide manageable links between the agent and service package. The association

between BaseAgent class and the ContextInterface (Figure 5.10) represents the Use Service Pattern. The AgentContext class that contains various service classes implements the context interface. Such an arrangement allows agent to use services locally. The agent context acts as the centric manager managing agent's execution, registration, communication, transportation and security services.

5.10. SERVICE PACKAGE – SERVICE-BASED IMPLEMENTATION PATTERNS

This package provides system's services and services policies. Categorizing services and their corresponding service policies is a complicated task. Using the Service-Based Implementation Pattern, Agent Policy Delegation Pattern and Hashtable Pattern simplify this task.

Figure 5.13 shows the design of Service classes using the Service-Based Implementation Pattern. In the distributed agent environment, the system services are categorized into five groups. They are security, mobility, communication, registry and execution.

Security: this service provides three security checks: (a) agent security, which ensures that only the agent owner who can manipulate or terminate his agent; (b) agent context security, which ensures the integrity of the agent context. The agent context can terminate an agent if the latter threatens the safety of the agent context; and (c) the system security, which provides authentication on users.

Mobility: this service allows agents to travel various locations in the network. The service is implement using JAVA RMI. This service contains two components: client and server.

The client side deactivates and serialized an agent that requires to be transferred. The server side receives and activates the agent.

Communication: this service provides agent communication by sending formatted messages. For example, the user agent sends a query to the broker agent using this service. When receiving the message, the broker agent examines and processes the message.

Registry: this service registers an agent when it is created, and removes agent from the registry list when it is terminated. It also provides various methods to facilitate the agent management in an agent context.

Execution: this service provides the fundamental methods for an agent execution in agent context. Methods include `receive()`, `activate()`, `createAgent()` and `terminate()`.

Each of these services is associated with a corresponding policy class that defines the service as indicated by Figure 5.13.

Figure 5.13: Use of Service-Based Implementation Pattern.

5.11. DETAIL IMPLEMENTATION OF AGENT POLICY

The Service-Based Implementation Pattern provides a way to categorize the policy for each service. The Agent Policy Pattern or Hashtable Pattern can be used to carry out the detail implementation of each policy class.

5.11.1. Agent Policy Delegation Pattern

An example of applying the Agent Policy Delegation Pattern is given in Figure 5.14. The implementation of security policy is done by the SecurityPolicyPool, which contains three different security policies internal, external and resource permission policies.

Figure 5.14: Use of the Agent Policy Pattern.

5.11.2. Hashtable Pattern

Figure 5.15 illustrates an example of applying the Hashtable Pattern in the design of the action policy classes. One task that the Action Layer of the broker agent has to do is to determine what book information should be returned to the user agent when receiving book inquiry information from the search agent. According to what is required by the user agent, the returned information could be the earliest delivering date, the closest location or the lowest price. If the user agent requires the lowest price, a policy keyword “price” is passed to the broker agent. The ActionLayer object requests the PricePolicy from the PolicyHashTable by calling the method `applyPolicy(“price”)`. The PolicyHashTable returns the PricePolicy object to the caller. If the PricePolicy does not exist, the PolicyHashTable returns a “null” object indicating that no policy is applicable in this case.

Figure 5.15: The Hashtable Pattern.

5.12. SUMMARY

Given an agent application system, e.g., distributed book-ordering system, we demonstrated the usefulness of the agent policy pattern language in designing a distributed agent environment. The design process adopted a “top-down” approach beginning with the Agent Pattern, and then applying the Use Service Pattern, Service-based Implementation Pattern and Layer Structure Pattern. By applying three patterns, we could also use the Agent Abstraction Pattern, Agent Policy Delegation Pattern and Hashtable Pattern. Through the application of these patterns, we developed a design skeleton of a distributed agent environment and its application.

CHAPTER 6: CONCLUSION

6.1. CONTRIBUTIONS

In this thesis, six agent policy patterns were extracted from various available agent frameworks. These patterns are Agent Abstraction Pattern, Use Service Pattern, Service-Based Implementation Pattern, Layered Structure Pattern, Agent Policy Delegation Pattern and Hashtable Pattern.

1. The Agent Abstraction pattern combines the abstract and proxy patterns to construct a generic base agent. Agent developers can create customized agent by subclassing the base agent. Use of this pattern one can achieve good encapsulation practice.
2. The Use Service Pattern implements an agent execution context with a common interface. The agent context provides agents with an execution environment and an interface to allow agents to inquiry or to access the existing services. This means that an agent can rely on services provided by the agent context rather than on its own.
3. The Service-Based Implementation pattern categorizes policies according to their association with services. This pattern defines that the service is the applicable scope of all the implemented policies within it. The growth or alternation of policy is done also within the service.
4. The Layered Structure Pattern suggests a layer structure policy implementation for agents that have layer architecture. Each layer of the agent is directly associated with a policy object that defines the behavior, service, interaction and security of that layer.

5. The Agent Policy Delegation Pattern decouples the dependency between the layer object and concrete policy objects by introducing a substitution relation into the layer policy implementation.
6. Hashtable Pattern offers an efficient way for the policy implementation using a hashtable structure.

An agent policy pattern language binding the existing agent patterns, i.e., Agent Pattern and Agent Layer Structure Pattern, and the new agent policy patterns was also developed. This pattern language was applied in a high-level design of a distributed agent environment for agent-based application.

6.2 FUTURE WORK

Policy Patterns that are introduced in this thesis are related to the high-level policy categorizing and structuring for a distributed agent environment. Many areas of detail concrete policy implementations are of interest for agent policy pattern research. For example, possible future study can be continued in the following topics.

- ❑ Security Policy Patterns
- ❑ Registry Policy Patterns
- ❑ Communication Patterns
- ❑ Agent Interaction and Collaboration Patterns

Patterns extracted from these areas will offer more details in terms of policy design and implementation for building a distributed agent environment and agent-based applications, and will enhance the agent policy pattern language developed in this work.

Reference:

1. D. Chaum, "Achieving Electronic Privacy", Scientific American, August, pp96-101, 1992.
2. A., Van Dam, "Hypertext'87 Keynote Address", in Communications of the ACM, 31(7), pp887-895, 1988.
3. Aparicio, M., "IBM Intelligent Agents," FIPA Opening Forum Proceedings, Yorktown, New York, 1996.
4. IBM Aglets, IBM Aglets: Programming Mobile Agents in Java, A White Paper, <http://www.ibm.co.jp/trl/aglets/whitepaper.htm>, IBM Tokyo Research Laboratory, 1996.
5. Grasshopper Development System, release 1, IKV++ GmbH, URL: <http://www.ikv.de>.
6. Voyage, URL: <http://www.objectspace.com/voyage>. JF of 104
7. JAFMAS, "A Java-based Agent Framework for Multi-Agent Systems", URL: <http://www.eecs.uc.edu/~abaker/JAFMAS/>
8. C. Hewitt, "Viewing Control Structure as Patterns of Passing Messages", Artificial Intelligence, 8, No3, pp323-364, 1977.
9. Nwana, H.S., "Software Agents: An Overview," Knowledge Engineering Review, Vol. 11, No 3, Cambridge University Press, pp.1-40, Sept 1996.
10. B. Hermans, "Intellegent Software Agents on the Internet: an inventory of currently offered functionality in the information society and a prediction of future developments", URL: <http://www.hermans.org/agents>.

11. T. Magedanz, K. Rothermel and S. Krause, "Intelligent Agents: An Engineering Technology for next Generation Telecommunications?", URL:
<http://www.ikv.de/products/grasshopper>.
12. Chaib-draa, B., Moulin, B., Mandiau, R. & Millot, P., "Chapter 1 - Trends in Distributed Artificial Intelligence," Foundations of Distributed Artificial Intelligence, G. M. P. O'Hare and N. R. Jennings (eds.), John Wiley & Sons Inc., pp. 3-55, 1996.
13. FIPA Rationale, http://www.cselt.stet.it/fipa/fipa_rationale.htm, 1996.
14. Foner, L., "What is an Agent, Anyway? A Sociological Case Study," Agents Memo 93-01, MIT Media Lab, Cambridge, MA, 1993.
15. M. Breugst, I. Busse, S. Covaci and T. Magedanz, "A Mobile Agent Platform for IN (Intelligent Network) Based Service Environments", URL:
<http://www.ikv.de/products/grasshopper>.
16. M. Wooldridge and N. Jennings, "Intelligent Agents: Theory and Practice", Knowledge Engineering Review, 10, No2, pp115-152.
17. J. Dale, "A mobile Agent Architecture for Distributed Information Management", URL: <http://www.cnri.restm.va.us/home/keo/bib/mobile-abs.bib.html>.
18. P. Maes, "Agents that Reduce Work and Information Overload" in Communications of the ACM, 37(7), pp31-40, 1994.
19. J. Bates, "The Role of Emotion in Believable Characters", in Communications of the ACM, 37(7), pp122-125, 1994.
20. (a) Mayfield, J., Labrou Y., & Finin, T., "Evaluation of KQML as an Agent Communication Language," Intelligent Agents Volume II -- Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages, Lecture Notes in

- Artificial Intelligence, M. Wooldridge, J. P. Muller and M. Tambe (eds.), Springer-Verlag, pp. 347-360, 1996. (b) Muller, J. P., "A Conceptual Model for Agent Interaction," Proceedings of the 2nd International Working Conference on Cooperative Knowledge Based Systems (CKBS-94), Deen, S. M. (ed.), Keele University: Dake Centre, pp. 213-234, 1994.
21. Jennings, N. R, "Chapter 6 - Coordination Techniques for Distributed AI," Foundations of Distributed Artificial Intelligence, G. M. P. O'Hare and N. R. Jennings (eds.), John Wiley & Sons, pp. 187-210, 1995.
 22. A.H. Bond and L. Gasser, "Readings in Distributed Artificial Intelligence", Morgan-Kaufmann, Los Angeles, 1988.
 23. K. Indermaur, "Baby Steps", BYTE Magazine, pp97-104, March 1995.
 24. P. Wayner, "Agent Away", BYTE Magazine, pp113-118, May 1994.
 25. A Reinhardt, "The Network with Smarts", BYTE Magazine, pp51-64, October 1994.
 26. Concordia, Mobile Agent Computing - A White Paper,
<http://www.meitca.com/HSL/Projects/Concordia/MobileAgentsWhitePaper.html>,
 Mitsubishi Electric Information Technology Center of America, 1997.
 27. Odyssey, Odyssey Frequently Asked Questions,
<http://www.genmagic.com/agents/odyssey-faq.html>, General Magic Inc., 1997.
 28. JATLite, JATLite overview,
http://java.stanford.edu/java_agent/htmlJATLiteOverview.html/, Stanford University,
 1997.

29. InteRRap, InteRRap Agent Architecture, <http://www.dfki.uni-sb.de/~pischel/interrap.html>, German Artificial Intelligence Research Institute (DFKI), 1994.
30. dMars, <http://www.aaii.oz.au/proj/dMARS-prod-brief.html>, by Australian Artificial Intelligence Institute (AII), 1996.
31. Cybele Agent Infrastructure, <http://www.i-a-i.com/projects/cybele/index.html>, 1996.
32. AgenTalk, AgenTalk: Describing Multiagent Coordination Protocols, <http://www.cslab.tas.ntt.jp/at/>, NTT Communication Science Laboratories and Ishida Laboratory, Department of Information Science, Kyoto University, Japan, 1996.
33. AgentTcl, Transportable Agents System, <http://www.cs.dartmouth.edu/agenttcl.html>, Computer Science Dept., Dartmouth University, 1995.
34. Telescript, Telescript Technology: The Foundation for the Electronic Marketplace, <http://www.genmagic.com/Telescript/Whitepapers/wp1/whitepaper-1.html>, General Magic Inc., 1996.
35. Swarm, The Swarm Simulation System, <http://www.santafe.edu/projects/swarm/overview.html>, The Santa Fe Institute, 1997.
36. Echelon, About LonWorks Control Technology, <http://www.lonworks.echelon.com/>, Echelon Corporation, 1996.
37. (a) Barbuceanu, M & Fox, M.S., "The Design of COOL: A Language for Representing Cooperation-Knowledge in Multi-Agent Systems," <http://www.ie.utoronto.ca/EIL/ABS-page/ABS-intro.html>, 1993. (b) Barbuceanu, M. & Fox, M.S., "The Architecture of a Generic Agent for Collaborative Enterprises," EIL Working Paper, Nov. 1994.

38. J.O. Coplien, "Software Patterns", ISBN 1-884842-50-x SIGS, Books & Multimedia, 1997.
39. C. Alexander, "A pattern Language: Towns, Buildings, Construction", Oxford University Press, 1977.
40. C. Alexander, "The Timeless Way of Building", Oxford University Press, 1979.
41. W. Cunningham and K. Beck, "Window per Task", URL:
<http://www.c2.com/cgi/wiki?WindowPerTask>, 1997,
42. Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns",
43. James O. Coplien and Douglas C. Schmidt, "Pattern Language of Program Design", published by Addison Wesley Longman, Inc., 1995.
44. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal "Pattern-Oriented Software Architecture: A System of Patterns", Wiley Computer Publishing, 1998.
45. Mark Grand, "Patterns in Java", Vol 1, Wiley Computer Publishing, 1999.
46. Mark Grand, "Patterns in Java", Vol 2, Wiley Computer Publishing, 1999.
47. 95.590 course notes, Department of Computer Science, Carleton University, 1998.
48. Jim Doble, "Pattern Language
49. D. Deugo and D. Sandu, "A Pattern Language for Developing Graphical User Interfaces", 1999.
50. A. Silva and J. Delgado, "The Pattern Agent: A Perspective from the Mobile Agent System Point of view", URL: <http://bruxelas.in.esc.pt/~alb/publicat.html>.
51. Liz Kendall, "A Java Application Framework for Agent Based Systems", URL:
http://www.cetus-links.org/oo_mobile_agents.html

52. Liz Kendall, "The Layered Agent Pattern", URL: http://www.cetus-links.org/oo_mobile_agents.html
53. M. Weiss, "Runtime Feature Interaction Management" Discussion Paper, May 1999, Mitel Corporation.
54. D. B. Lange and M. Oshima, "Programming and Developing Java Mobile Agents with Aglets", published by Addison Wesley Longman, Inc., 1999.
55. M. Sloman and E. Lupu, "Policy Specification for Programmable Networks", Extended version of paper in Proceedings of First International Working Conference on Active Networks (IWAN'99), Berlin, June 1999, ed. S. Covaci, published by Springer Verlag Lecture Notes in Computer Science
56. D. Kuokka and L. Harada, "On Using KQML for Matchmaking" in Proceedings of First International Conference on Multi-agent Systems", Menlo Park, CA, AAAI Press.
57. M. R. Gensereh and R.E. Fikes, "Knowledge Interchange Format", Version 3.0, reference manual, 1992.
58. J.P. Bigus and J. Bigus, "Constructing Intelligent Agent with Java", Wiley Computer Publishing, 1997.