About the Authors

Dr. Joseph P. Bigus is a member of the extended architecture team for the IBM Agent Builder Environment and an architect and development team leader on the IBM Intelligent Miner product. He works in the IBM AS/400 programming laboratory in Rochester, MN. Joe has more than ten years of experience working with artificial intelligence applications. Dr. Bigus has authored a book on data mining as well as articles and technical papers on neural networks and their applications. He currently holds sixteen U.S. patents on neural network technology. He holds a B.S. in Computer Science from Villanova University, and an M.S. and Ph.D. in Computer Science from Lehigh University in Bethlehem, PA.

Jennifer Bigus is a software engineer in the IBM AS/400 programming laboratory in Rochester, MN. She is currently a leader of Java development for AS/400 systems and has led other object-oriented development projects in the past. Jennifer has worked in the computer industry for more than 15 years. She holds a B.S. in Computer Science from Winona State University, and an M.S. in Computer Science from Lehigh University.

Preface

The term artificial intelligence (AI) has gone in and out of style over the past forty years. At first, it represented the promise of computers and the ability of humankind to create intelligent machines with comparable and possibly superior abilities to humans. But then, after years of unfulfilled expectations, it came to be associated with failed approaches. Although the term may have fallen out of favor, the basic desire to make computers act smarter or more intelligent has not gone away. Indeed, as we have built more and more complex systems out of layers of conventional software, the need for intelligence, whether real or artificial, has only increased.

In this book, we approach artificial intelligence not as a holy grail where only parity with human abilities signals success, but as a set of techniques for making software that is more intuitive, easy to use, and which makes users more

productive. In short, artificial intelligence allows us to build software which is smarter than it otherwise would be, and which is qualitatively better than software that does not use artificial intelligence techniques.

Whether viewed as a failure or success, the fact remains that artificial intelligence research has made major contributions to the design and development of mainstream commercial software. Perhaps the biggest impact has been in the area now known as object-oriented programming. The idea of software objects with associated data and procedures is a refinement of the AI concept of frames, which had slots or attributes for holding data related to a concept, and attached procedures (methods) for processing that data. The Smalltalk programming language was the first "pure" object programming language, but object extensions to Lisp, Pascal, and C were also made. The C++ language, based on C, is the most widely used object-oriented programming language today, primarily because it was an extension of the extremely popular C language that already had thousands of programmers fluent in its use.

While once a topic of heated argument in software development circles, the object-oriented approach to commercial software design and development is clearly the preferred method today. The recent introduction of the Java programming language by Sun Microsystems and its spin-off JavaSoft has not only confirmed this preference, but has shown how deeply this conviction is held. The industry interest and uptake of Java has been astounding to most experienced software developers, because it usually takes decades before new programming languages gain any significant following. However, Java, with its C++-like syntax, and Smalltalk-like semantics and run-time behavior, has received unprecedented buy-in from commercial software developers, even though it has been in commercial use for only a short time. Part of the reason for its popularity can be attributed to Java's "write once, run anywhere" cross-platform portability, built-in graphical user interface (GUI) framework, and strong support for both client/server and network-centric applications. As we describe later, this same collection of language features makes Java idea for writing intelligent agent applications.

How to Use This Book

This book is targeted at object-oriented programmers who are interested in learning how to make their programs behave more intelligently. Programmers with experience in C++ or Smalltalk should have no trouble following the Java

examples presented in this book. While we provide an introduction to the Java language and programming environment, this is not a Java primer. If you are completely new to Java programming, you will also want to get a book written explicitly for beginning Java programmers.

Readers can use this book to achieve two different but overlapping goals. For those most interested in the underlying artificial intelligence programming techniques, we provide an introduction to the relevant AI concepts as well as concrete Java code examples. This material, presented in the first half of the book, could be used as a text for an undergraduate course in AI programming. People who are most interested in creating their own intelligent agents or in understanding the issues related to intelligent agent applications can focus on the second half of the book, where we present our design and implementation of a Java intelligent agent framework, along with several applications. Taken in its entirety, this book provides the background and fundamentals of AI programming techniques as a base, and builds on this foundation to construct a complete framework for developing intelligent agent applications, whether on the desktop or across the Internet.

In addition to providing a discussion of the technical issues related to intelligent agents, we provide citations to a number of related papers and articles. At the end of each chapter, a set of exercises and questions are provided to test your understanding of the material. A set of Java applications and applets is included for hands-on experimentation and for exploring (and possibly extending) the behavior of the various AI techniques.

Organization of the Book

This book is divided into two major parts. Part 1 focuses on the artificial intelligence algorithms and techniques used to make applications and agents intelligent. Part 2 builds on Part 1 by taking the AI algorithms and using them in an intelligent agent framework and example applications. All programs and examples are written entirely in Java.

In the Introduction, we discuss some of the history of artificial intelligence research and the basic premises of both the symbol processing and neural network (connectionist) schools. We explore the evolution of artificial intelligence systems into today's intelligent agents and the simultaneous emergence of network computing and the World Wide Web. We discuss the

many ways that artificial intelligence can be used to add value to commercial software systems.

Chapter 1 presents a brief overview of the Java programming language and development environment, including data types, control structures, the differences between Java applications and applets, and basic object-oriented programming support such as objects, classes, and inheritance. Next we highlight the classes and functions provided in several of the major Java packages, including java.lang, jav.awt, java.beans, and java.sql. We examine Java from an intelligent agent perspective in terms of support for autonomy, intelligence, and mobility. We compare and contrast Java with C++ and Smalltalk as a general-purpose, object-oriented programming language. This chapter is meant to get experienced object-oriented programmers up to speed quickly on the unique aspects of the Java language and to introduce the specific language features used in our agent applications.

In Chapter 2, we show how to solve problems using search and state-based definitions of the world. We describe how problems can be mapped onto a state-space representation, and how common AI search strategies such as breadth-first, depth-first, and best-first search can be used to find solutions to problems. We implement these search techniques in Java and develop an applet to help us examine their behavior on sample problems. Other more advanced heuristic search techniques are also discussed.

Chapter 3 deals with the major types of knowledge representation used in AI systems. We start with a general discussion of the types of knowledge we need to represent and then explore propositional and predicate logic, frames, and semantic networks. An emerging standard knowledge representation, the Knowledge Interchange Format (KIF), is described in detail. Finally, we discuss the issues when building a domain-specific knowledge base. Knowledge representation is a key element in any artificial intelligent application, so we pay particular attention to the strengths and weaknesses of the various techniques.

Reasoning systems, specifically rule-based systems, are the focus of Chapter 4. The basic elements of a rule are described, including antecedent and consequent clauses, certainty factors, and sensor and effector (action) rules. An example rule base is used throughout the chapter to illustrate the concepts related to forward chaining and backward chaining with rules. We develop a Java applet and classes to implement rules and rule-based inferencing. Fuzzy rule systems that

combine rules and fuzzy logic are described. We also explore the issues related to planning, a specialty in AI that is particularly relevent to intelligent agent systems.

In Chapter 5, we discuss learning and adaptive techniques and the advantages such behavior provides in intelligent agents. We describe the fundamental issues in building learning systems, and the major paradigms including supervised, unsupervised, and reinforcement learning. An introduction to neural networks and the fundamentals of back propagation networks and self-organizing feature maps are described. Next, we show how information theory is used to build decision trees using induction. These three learning algorithms are then implemented in Java, along with an applet to serve as the user interface.

Chapter 6 provides a bridge from the artificial intelligence techniques described in Chapters 2 through 5 to the intelligent agents and the applications developed in the rest of the book. We look at the agent attributes of perception and action and how AI provides the solutions. The issues related to multiagent systems are also discussed. Classic blackboard agent architecture is described as an introduction to the general topic of communications between agents. The Knowledge Query and Manipulation Language (KQML) is examined as the leading approach for communicating between agents. Finally, we discuss how agents can cooperate and compete in the emerging electronic commerce world.

In Part 2, we change our focus from the underlying artificial intelligence issues and techniques to their application to major intelligent agent paradigms. Part 2 consists of a set of application chapters, where we design and build an intelligent agent architecture and then apply it to several common application domains. We build on the artificial intelligence functions developed and described in Part 1.

Chapter 7 is a key chapter in this book, where we develop the CIAgent intelligent agent architecture, going from requirements through specifications and design. We describe the major Java classes, the functions they provide, and their Java implementations. The Java Beans software component model is evaluated as it relates to intelligent agent requirements. Finally, we extend the rule-based processing capabilities introduced in Chapter 4 by adding support for Facts, and Sensors and Effectors in rules.

Chapter 8 illustrates how we can use our CIAgent framework to construct an application that assists a user with PC management and local application

activities. We develop two CIAgent intelligent agents, one to handle time-based alarms, and one to watch file states. The user can set alarms to go off at specific times or at regular intervals, or can use a watch to monitor changes to file contents. Whenever alarms or watches go off, the application can alert the user, or run an arbitrary system command or application specified by the user.

The Internet is the focus of our intelligent agent application in Chapter 9. We design and develop a simple Internet news reader application and enhance it by providing information filtering. The user can specify a set of keywords that are used to score news articles for their potential relevance to the user. Three different types of filtering are provided. Basic keyword filtering orders the articles based on the total number of keyword matches. Neural segmentation is used in cluster filtering, where the article profiles are clustered and the scores in each cluster are averaged. In feedback filtering, articles are assigned a relevance value ranging from useless to interesting. These values are used as the target value to train a back propagation neural network model to predict the relevance scores of new articles.

Chapter 10 focuses on the issues involved when autonomous agents interact in multiagent systems. We develop an electronic marketplace application, with a CIAgent-based Facilitator as the market manager. BuyerAgents and SellerAgents communicate with the Facilitator using KQML-like messages and the Java Beans event framework. The sales negotiation strategies range from simple hardcoded logic to rule-based inferencing using action rules.

We conclude in Chapter 11 with an examination of several Java-based agent environments and applications. These include IBM's Aglets, FTP Software agents, ObjectSpace's Voyager, General Magic's Odyssey, Stanford University's JATLite, MCC's InfoSleuth, and IBM's Agent Builder Environment. We close with a discussion of the state of the art of Java intelligent agents and expected future developments.

Appendix A provides the source code for some of the example rule-bases used in Chapter 4. Appendix B contains the data sets used in the Learn applet in Chapter 5. The Bibliography is a resource list of artificial intelligence, intelligent agent, and Java papers, articles, and books.

Conventions Used in this Book

Italic is used to indicate filenames, file pathnames, and program names, and is used to identify important new terms as they are defined and introduced in the text. *Italic* is also used for packages, methods, variables, and HTML tags when they are referenced in the main text. **Boldface** is used for GUI buttons, menu items, user commands, and class names. Courier Font is used for Java and HTML code set off from the main body of the text.

Java Applets and Applications

This book includes a CD containing the Java code from the artificial intelligence applets developed in Part 1, as well as the CIAgent intelligent agent applications from Part 2. Some of the same *.java* files appear in both directories when enhancements or modifications were required for Part 2.

The applets can be run by opening a Web browser or applet viewer on the HTML files in the *ciagent/part1/* directory. The applets include:

SearchApplet.html
RuleApplet.html
LearnApplet.html

The applications can be run by going to the *ciagent/part2/* directory and running them from the command line. These include:

> PCManager
> NewsFilter
> Marketplace

Accessing On-Line Information

The World Wide Web and the Internet provide a large amount of resources on the Java programming language and intelligent agents. The Sun and JavaSoft Web sites contain complete documentation of the Java language and the Java APIs. Development tool vendors such as Symantec, Borland, IBM, and Microsoft also have sites devoted to Java. Intelligent agent information is easily obtained from one of several World Wide Web sites, such as:

UMBC AgentWeb—This academic site, sponsored by the University of Maryland, Baltimore County, contains many useful links to information about

intelligent software agents. The material ranges from basic introductory level to contacts to researchers doing advanced work in this area. The URL is http://www.cs.umbc.edu/agents/.

MIT Software Agents Group—This site, sponsored by the Massachusetts Institute of Technology's Media Lab, features a searchable database to other agents sites and agent conference information, as well as many online research papers. The URL is http://agents.www.media.mit.edu/groups/agents/.

Knowledge Sharing Effort—This Web site at Stanford University features links to papers and code related to the Knowledge Sharing Effort which produced both the Knowledge Interchange Format (KIF) and Knowledge Query and Manipulation Language (KQML) specifications. The URL is http://hpdce.stanford.edu/knowledge.html.

SIGART—The Association of Computing Machinery (ACM) Special Interest Group on Artificial Intelligence home page contains information on the International Joint Conference on AI as well as an excellent index based on artificial intelligence and related subject areas. The main URL is http://sigart.acm.org/ai/ and the subject index page is http://sigart.acm.org/ai/ontology.html.

IBM alphaWorks—This site, sponsored by IBM Research, contains much of the latest Internet technology developed by IBM. Packages are updated regularly and some products are removed when their status changes from research and product previews to commercial products. You can find information on the Agent Builder Environment (ABE), Web Browser Intelligence (Webby), and Aglets, the IBM mobile Java agent technology. The URL is http://alphaworks.ibm.com.

Acknowledgments

We would like to extend our thanks to the many people who helped make this book possible. First, thanks to our children, Sarah Jo and Alexander, who had to deal with not one, but both parents working on a book at the same time. We promise we will go camping soon.

Next, we would like to thank the people who read through early drafts of the material, who reviewed our design and Java coding practices, and who, through

their feedback, helped improve the quality of the book. These include Cindy Hitchcock, Jeff Pilgrim, Don Schlosnagle, Rich Burr, Paul Coffman, and Paul Monday from IBM.

We would like to thank Ed Kay, Glenn Blank, and Don Hillman at Lehigh University for their education in AI techniques. Joe would also like to thank some of the people at IBM who worked on the Agent Builder Environment project (before it was ABE). These include Manny Aparicio, Benjamin Grosov, Joe Resnak, David Levine, Joe Gdaniec, Betty Atkinson, and Don Gilbert. Discussions with Ted Selker and Rob Barret from IBM Almaden Research Center were also valuable.

Our managers at IBM, Debbie Kummer and Victoria Mathews, also deserve our thanks for their patience, understanding, and support as we worked on this project in parallel with our day-to-day responsibilities at IBM.

And, finally, it was our pleasure to work with Marjorie Spencer, Margaret Hendrey, and Angela Murphy at Wiley. They were always helpful and patient, even when we missed our deadlines for manuscript delivery.

Previous | Table of Contents | Next

Introduction

In this section, we present an introduction to the two major topics covered in this book: artificial intelligence and intelligent agents. We trace the history of artificial intelligence research and discuss the basic premises of both the symbol processing and neural network schools. We explore the evolution of artificial intelligence systems from an interesting but largely discredited technology into the basis for today's intelligent agent applications. The simultaneous emergence of network computing and the World Wide Web, and their requirements for intelligent software are also discussed. We present key attributes of intelligent agents such as autonomy, mobility, and intelligence and provide a taxonomy for classifying various intelligent agent applications.

Artificial Intelligence

The science of artificial intelligence (AI) is approximately forty years old, dating back to a conference held at Dartmouth in 1958. During the past forty years, the public perception of AI has not always matched the reality. In the early years, the excitement of both scientists and the popular press tended to overstate the real-world prowess of AI systems. Early success in game playing, mathematical theorem proving, common-sense reasoning, and college mathematics seemed to promise rapid progress toward practical machine intelligence. During this time the fields of speech recognition, natural language understanding, and image optical character recognition all began as specialties in AI research labs.

However, the early successes were followed by a slow realization that what was hard for people and easy for computers was more than offset by the things that were easy for people to do but almost impossible for computers to do. The promise of the early years has never been fully realized, and AI research and the term *artificial intelligence* have become associated with failure and overhyped technology.

Nevertheless, researchers in artificial intelligence have made significant contributions to computer science. Many of today's mainstream ideas about computers were once considered highly controversial and impractical when first proposed by the artificial intelligence community. Whether it is the WIMP

(windows, icon, mouse, pointer) user interface, which dominates human-computer interaction today, or object-oriented programming techniques, which are sweeping commercial software development, AI has made an impact. Today, the idea of intelligent software agents helping users do tasks across networks of computers would not even be discussed if not for the years of research in distributed AI, problem solving, reasoning, learning, and planning. So before we dive into the tools and tricks of the AI trade, let's take a brief look at the underlying ideas behind the intelligence in our intelligent agents.

Basic Concepts

Throughout its history, artificial intelligence has focused on problems which lie just beyond the reach of what state-of-the-art computers could do at that time (Rich and Knight 1991). As computer science and computer systems have evolved into higher levels of functionality, the areas which fall into the domain of AI research have also changed. Invented to compute ballistics charts for World War II–era weapons, the power and versatility of computers were just being imagined. Digital computers were a relatively new concept, and the ideas of what would be useful AI functions included game playing and mathematics.

After 40 years of work, we can identify three major phases of development in AI research. In the early years, much of the work dealt with formal problems that were structured and had well-defined problem boundaries. This included work on math-related skills such as proving theorems, geometry, calculus, and playing games such as checkers and chess. In this first phase, the emphasis was on creating general "thinking machines" which would be capable of solving broad classes of problems. These systems tended to include sophisticated reasoning and search techniques.

A second phase began with the recognition that the most successful AI projects were aimed at very narrow problem domains and usually encoded much specific knowledge about the problem to be solved. This approach of adding specific domain knowledge to a more general reasoning system led to the first commercial success in AI—expert systems. Rule-based expert systems were developed to do various tasks including chemical analysis, configuring computer systems, and diagnosing medical conditions in patients. They utilized research in knowledge representation, knowledge engineering, and advanced reasoning techniques, and proved that artificial intelligence could provide real value in commercial applications. At this same time, computer workstations were

developed specifically to run Lisp, Prolog, and Smalltalk applications. These AI workstations featured powerful integrated development environments and were years ahead of other commercial software environments.

We are now well into a third phase of AI applications. Since the late 1980s, much of the AI community has been working on solving the difficult problems of machine vision and speech, natural language understanding and translation, commonsense reasoning, and robot control. A branch of AI known as connectionism regained popularity and expanded the range of commercial applications through the use of neural networks for data mining, modeling, and adaptive control. Biological methods such as genetic algorithms and alternative logic systems such as fuzzy logic have combined to reenergize the field of artificial intelligence. Recently, the explosive growth in the Internet and distributed computing has led to the idea of agents that move through the network, interacting with each other and performing tasks for their users. Intelligent agents use the latest AI techniques to provide autonomous, intelligent, and mobile software agents, thereby extending the reach of users across networks.

When we talk about artificial intelligence or intelligent agents, the question often arises, what do we mean by *intelligence*? Do we mean that our agent acts like a human? thinks like a human? that it acts or thinks rationally? While there are as many answers as there are researchers involved in AI work, we'll tell you what we think it means. To us, an intelligent agent acts rationally. It does the things we would do, but not necessarily the same way we would do them. Our agents may not pass the Turing test, proposed by Alan Turing in 1950 as a yardstick for judging computer intelligence. But our agents will perform useful tasks for us. They will make us more productive. They will allow us to do more work in less time, and see more interesting information and less useless data. Our programs will be qualitatively better using AI techniques than they would be otherwise. No matter how humble, that is our goal—to develop better, smarter applications.

Symbol Processing

There are many behaviors to which we ascribe intelligence. Being able to recognize situations or cases is one type of intelligence. For example, a doctor who talks with a patient and collects information regarding the patient's symptoms and then is able to accurately diagnose an ailment and the proper course of treatment exhibits this type of intelligence. Being able to learn from a

few examples and then generalize and apply that knowledge to new situations is another form of intelligence.

Intelligent behavior can be produced by the manipulation of symbols. This is one of the primary tenets of artificial intelligence techniques. Symbols are tokens which represent real-world objects or ideas and can be represented inside a computer by character strings or by numbers. In this approach, a problem must be represented by a collection of symbols, and then an appropriate algorithm must be developed to process these symbols.

The physical symbol systems hypothesis (Newell and Simon 1980) says that only a "physical symbol system has the necessary and sufficient means for general intelligent action." This idea, that intelligence flows from the active manipulation of symbols, was the cornerstone on which much of the subsequent AI research was built. Researchers constructed intelligent systems using symbols for pattern recognition, reasoning, learning, and planning (Russell and Norvig 1995). History has shown that symbols may be appropriate for reasoning and planning, but that pattern recognition and learning may be best left to other approaches.

There are several typical ways of manipulating symbols which have proven useful in solving problems. The most common is to use the symbols in formulations of If-Then rules which are processed using reasoning techniques called forward and backward chaining. Forward chaining lets the system deduce new information from a given set of input data. Backward chaining allows the system to reach conclusions based on a specific goal state. Another symbol processing technique is a semantic network, in which the symbols and the concepts they represent are connected by links into a network of knowledge that can then be used to determine new relationships. Another formalism is a frame, where related attributes of a concept are grouped together in a structure with slots and are processed by a set of related procedures called daemons or fillers. Changing the value of a single slot could set off a complex sequence of related procedures as the set of knowledge represented by the frames is made consistent. These reasoning techniques are described in more detail in Chapters 3 and 4.

Symbol processing techniques represent a relatively high level in the cognitive process. From a cognitive science perspective, symbol processing corresponds to conscious thought, where knowledge is explicitly represented, and the knowledge itself can be examined and manipulated. While symbol processing

and conscious thought are clearly part of the story, another set of researchers are examining a symbol-less approach to intelligence modeled after the brain.

Neural Networks

An increasingly popular method in artificial intelligence is called neural networks or connectionism. Neural networks have less to do with symbol processing inspired by formal mathematical logic, and more to do with how human or natural intelligence occurs. Humans have neural networks in their heads, consisting of hundreds of billions of brain cells called neurons, connected by adaptive synapses which act as switching systems between the neurons. Artificial neural networks are based on this massively parallel architecture found in the brain. They process information, not by manipulating symbols, but by processing large amounts of raw data in a parallel manner. Different formulations of neural networks are used to segment or cluster data, to classify data, and to make predictive models using data. A collection of processing units which mimic the basic operations of real neurons is used to perform these functions. As the neural network learns or is trained, a set of connection weights between the processing units is modified based on the perceived relationships in the data.

Compared to symbol processing systems, neural networks perform relatively low-level cognitive functions. The knowledge they gain through the learning process is stored in the connection weights and is not readily available for examination or manipulation. However, the ability of neural networks to learn from and adapt to their surroundings is a crucial function needed by intelligent software systems. From a cognitive science perspective, neural networks are more like the underlying pattern recognition and sensory processing that is performed by the unconscious levels of the human mind. We discuss neural networks and learning in Chapter 5.

Where artificial intelligence research was once dominated by symbol processing techniques, there is now a more balanced view where the strengths of neural networks are used to counter the weaknesses of symbol processing. In our view, both are absolutely necessary in order to create intelligent applications and intelligent autonomous agents.

The Internet and the World Wide Web

The Internet grew out of government funding for researchers who needed to collaborate over great distances. As a byproduct of solving those problems, protocols that allowed different computers to talk to each other, exchange data, and work together needed were developed. This led to TCP/IP as the de facto standard networking protocol for the Internet. The growth in the Internet is astounding, with the number of sites growing exponentially. Thousands of new sites are connected to the Internet each month.

While electronic mail (e-mail) was once the primary service provided by the Internet, information publishing and software distribution are now of equal importance. The Gopher text information service, which gained popularity in the early 1990s generated the first wave of information publishing on the net. The File Transfer Protocol (FTP) allows users to download research papers and articles as well as retrieve software updates and even complete software products over the Internet. But it was the HyperText Transfer Protocol (HTTP) that brought the Internet from the realm of academia and computer technologists into the public consciousness. The development of the Mosaic browser at the University of Illinois transformed the Internet into a general-purpose communications medium, where computer novices and experts, consumers, and businesses can interact in entirely new ways.

The World Wide Web, with its publishing and broadcasting capabilities, has extended the range of applications and services which are available to users of the Internet. The now-ubiquitous Web browser provides a universal interface to applications regardless of which server platform is serving up the application. In the browsing or “pull” mode, the Web allows individuals to explore vast amounts of information in one relatively seamless environment. Knowing that all of the information is out there, but not knowing exactly how to find it, can make the Web browsing experience quite frustrating. The popular search engines and Web index sites such as AltaVista, Excite, Yahoo, and Lycos provide an important service to users of the Web, by grouping information by topics and keywords. But even with the search engines and index sites, Web browsing is still a hit-or-miss proposition (with misses more likely than hits). In this environment, intelligent agents will emerge as truly useful personal assistants by searching, finding, and filtering information from the Web, and bringing it to a user’s attention. Even as the Web evolves into “push” or broadcast mode, where users subscribe to sites which send out constant updates to their Web pages, this requirement for filtering information will not go away. Unless the broadcast sites are able to send out very personalized streams of information, the user will still

have to separate the valuable information from the useless noise.

While the Internet and World Wide Web have captured the public's attention, businesses are quickly adapting the way they use information technology through their internal networks or intranets. Companies use intranets for internal and external e-mail, to post information, and to handle routine administrative tasks. Intranets allow a wide variety of client computers to connect to centralized servers, without the cost and complexity of developing client/server applications. Intranets serve the same purpose and have the same advantages for companies that the Internet has for individuals. A standardized client application, the Web browser, running on standard personal computers or low-cost network computers, can provide a single point of access to a collection of corporatewide network-based applications.

From AI to Intelligent Agents

As is often the case when a technical field provokes commercial interest, there has been a large movement and change of focus in the AI research community to apply the basic artificial intelligence techniques to distributed computer systems, companywide intranets, the Internet, and the World Wide Web. Initially, the focus was limited to word searches, information retrieval, and filtering tasks. But as more and more commercial transactions are performed on networks, there is more interest in having smart agents which can perform specific actions. By taking a step back and looking at what the Internet has become, many researchers who had been looking at how intelligent agents could cooperate to achieve tasks on distributed computer systems have realized that there is finally a problem in search of a technology (as opposed to the other way around). Intelligent agents can provide real value to users in this new, interconnected, and networked world.

Up to this point, we have discussed artificial intelligence and its evolution into software agents at an abstract level. In the following sections, we explore some of the technical facets of intelligent agents, how they work, and how we can classify them based on their abilities and underlying technologies.

Events-Conditions-Actions

Suppose we have an intelligent agent, running autonomously, primed with

knowledge about the tasks we require of it and ready to move out onto the network when the opportunity arises. Now what? How does the agent know that we want it to do something for us, or that it should respond to someone who is trying to contact us? This is where we have to deal with events, recognize conditions, and take actions.

In the context of intelligent agents, an event is anything that happens to change the environment or anything of which the agent should be aware. For example, an event could be the arrival of a new piece of mail. Or it could be a change to a Web page. Or it could be a timer going off at midnight—time to start sending out the faxes that are queued up. Short of having our agent constantly running and checking or polling all the devices and computer systems we want it to monitor, having events signal important occurrences is the next best thing. Actually, it may be the best thing, because our agent can sleep, think about what has happened during the day, do housekeeping tasks, or anything else useful while it is waiting for the next event to occur.

When an event does occur, the agent has to recognize and evaluate what the event means and then respond to it. This second step, determining what the condition or state of the world is, could be simple or extremely complex depending on the situation. If mail has arrived, then the event will be self-describing—a new piece of mail has arrived. The agent may then have to query the mail system to find out who sent the mail, and what the subject or topic is, or even scan the mail text to find keywords. All of this is part of the recognize component of the cycle. The initial event may wake up the agent, but the agent then has to figure out what the significance of the event is in term of its duties. In the mail example, suppose the agent recognizes that the mail is from your boss, and that the message is classified as URGENT. This brings us to the next and perhaps most useful aspect of intelligent agents—actions.

If intelligent agents are going to make our lives easier (or at least more interesting), they must be able to take action, to do things for us. Having computers do things for us is not a new idea. Computers were developed to help people do work. However, having the computer initiate an action on our behalf is something totally different from entering a command on the command line and pressing *Enter* to run the command. While the results of our typing the command and pressing *Enter* may not always be exactly what we had in mind when we typed it in (it always seems that we realize what we should have typed after we press *Enter*), we know that whatever happens, it is our doing. Having an agent

(intelligent or not, human or computerized) take an action for us requires a certain leap of faith or at least some level of trust. We must trust that our intelligent agent is going to behave rationally and in our best interest. Like all situations where we delegate responsibility to a third party, we have to weigh the risks and the rewards. The risk is that the agent will mess things up, and we will have to do even more work to set things right. The reward is that we are freed from having to worry about the details of getting that piece of work done.

So, events-conditions-actions define the workings of our agent. Some researchers feel that an agent must also be proactive. It must not only react to events, but must be able to plan and initiate actions on its own. We agree. However, in our view, this action (signaling some event, or calling some application interface) is the result of some earlier event which caused our agent to go into planning mode. We want our intelligent agents to be able to initiate transactions with other agents on our behalf, using all of the intelligence and domain knowledge they can bring to bear. But this is just an extension of the event-condition-action paradigm.

Taxonomies of Agents

While intelligent agents are still somewhat new in commercial computing environments, they have been the focus of researchers for years. In that time, many different ways of classifying or categorizing agents have been proposed. One way is to place the agent in the context of intelligence, agency, and mobility. Another approach is to focus on the primary processing strategy of the agent. A third is to categorize the agent by the function it performs. In the following sections we explore all three perspectives on viewing agent capabilities.

Agency, Intelligence, and Mobility

When we talk about software agents, there are three dimensions or axes which we use to measure the capabilities: agency, intelligence, and mobility (IBM 1996). Agency deals with the degree of autonomy the software agent has in representing the user to other agents, applications, and computer systems. An agent represents the user, helps the user, guides the user, and in some cases, takes unilateral actions on the user’s behalf. This progression from simple helper to full-fledged assistant takes us from agents which can be hardcoded, to those, which out of simple necessity, must contain more advanced intelligence

techniques.

Intelligence refers to the ability of the agent to capture and apply application domain-specific knowledge and processing to solve problems. Thus our agents can be relatively dumb, using simple coded logic, or they can be relatively sophisticated, using complex AI-based methods such as inferencing and learning.

An agent is mobile if it can move between systems in a network. Mobility introduces additional complexity to an intelligent agent, because it raises concerns about security (the agent's and the target system's) and cost. Intranets are a particularly ripe environment for mobile intelligent agents to roam because they require less security than in the wide-open Internet.

Processing Strategies

One of the simplest types of agents are *reactive* or *reflex* agents, which respond in the event-condition-action mode. Reflex agents do not have internal models of the world. They respond solely to external stimuli and the information available from their sensing of the environment (Brooks 1986). Like neural networks, reactive agents exhibit *emergent behavior*, which is the result of the interactions of these simple individual agents. When reactive agents interact, they share low-level data, not high-level symbolic knowledge. One of the fundamental tenets of reactive agents is that they are grounded in physical sensor data and are not operating in the artificial symbol space. Applications of these agents have been limited to robots which use sensors to perceive the world.

Deliberative or *goal-directed* agents have domain knowledge and the planning capability necessary to take a sequence of actions in the hope of reaching or achieving a specific goal. Deliberative agents may proactively cooperate with other agents to achieve a task. They may use any and all of the symbolic artificial intelligence reasoning techniques which have been developed over the past forty years.

Collaborative agents work together to solve problems. Communication between agents is an important element, and while each individual agent is autonomous, it is the synergy resulting from their cooperation that makes collaborative agents interesting and useful. Collaborative agents can solve large problems which are beyond the scope of any single agent and they allow a modular approach based

on specialization of agent functions or domain knowledge. For example, collaborative agents may work as design assistants on large, complex engineering projects. Individual agents may be called upon to verify different aspects of the design, but their joint expertise is applied to insure that the overall design is consistent. In a collaborative agent system, the agents must be able to exchange information about beliefs, desires, and intentions, and possibly even share their knowledge.

As mentioned earlier, a *mobile* agent is a software process (a running program's code and its state) which can travel across computer systems in a network doing work for its owner. An advantage of mobile agents is that the communications between the home system and the remote systems are reduced. By allowing the agent to go to the remote system and access data locally on that system, we enable a whole new class of applications. For example, mobile agents could provide an easy way to do load balancing in distributed systems. "Oh, the processor is heavily loaded here, better hop on over to System X for a while."

As Nwana (1996) says, "mobility is neither a necessary nor sufficient condition for agenthood." Sometimes it makes sense to have your agent go out on the network; other times it does not. For some people, the idea of sending a mobile agent off to work is comfortable and natural. For others, the lack of a familiar computing model (it is neither server-based nor client/server) makes mobile agents hard to fathom. For example, if your agent contains some exclusive domain knowledge or algorithms, then sending that intellectual property out on the network to reside on foreign hosts, may not be a good idea. Better to keep that agent at home in a safe, secure system, and send out messenger collaborative agents to do the traveling. Perhaps the biggest inhibitor to widespread use of mobile agents is security. We have a name for software that comes unbidden onto our systems and starts executing. We call them *viruses*. How do we make sure that only "good" mobile agents can run on our system, but not "bad" agents? And how can we tell the difference?

Processing Functions

Perhaps the most natural way of thinking about the different types of agents is based on the function they perform. Thus, we have user interface agents, which try to "do what you mean" rather than what you say when interacting with an piece of application or system software. We have search agents, which go out on the Internet and find documents for us. We have filter agents, which process

incoming mail or news postings, ferreting out the stuff of interest from the more mundane or uninteresting rubbish. We have domain-specific assistants, which can book a business trip, schedule a meeting, or verify that our design does not violate any constraints. In short, any combination of intelligent agent attributes can be combined and applied to a specific domain to create a new, function-specific intelligent agent. All we need are the software tools and computing infrastructure to be able to add the domain knowledge and reasoning or learning capabilities to our agents, and the comfort level that is required to trust the agent to do our bidding.

Interface agents work as personal assistants to help a user accomplish tasks. Interface agents usually employ learning to adapt themselves to the work habits and preferences of the user. Patti Maes (1994) at MIT identifies four ways that learning can occur. First, an agent can learn by watching over the user's shoulder, observing what the user does and imitating the user. Second, the agent can offer advice or take actions on the user's behalf and then learn by receiving feedback or reinforcement from the user. Third, the agent can get explicit instructions from the user (when this happens, then do that). Finally, by asking other agents for advice, and agent can learn from their experiences. Note that interface agents collaborate primarily with the user, not with other agents (asking advice is the one exception). Using various learning mechanisms, interface agents offer the promise of customizing the user interface of a computer system or set of applications for a particular user and their unique working style. If interface agents can collaborate and share their knowledge about how to do a task, then when one person in a workgroup figures out how to do something, that skill could be transferable to all other users in that workgroup through their interface agents. The productivity gains could be enormous.

Another generic class of agents is *information* agents. In some ways, information agents are the Dr. Jekyll/Mr. Hyde of software programs. Some information agents go out on the Internet or the Web and seek out information of interest to the user. Others filter streams of information coming in e-mail correspondence and newsgroup postings. But either way, information agents try to help with the core problem of getting the right information at the right time. The question is not whether there is too much information or too little, but making sure that you see the right information. Of course, what is "right" depends on the context in which you are working. This information overload problem is one of the prime factors in the emergence of intelligent software agents as viable commercial products. Whether routing particularly important e-mail messages, or actively

constructing a personal newspaper (consisting only of interesting articles and advertisements), information agents promise relief from the overwhelming amount of data we are exposed to each day. Information agents called Spiders are already being used to index the Web. In general, information agents can be static, and sit on one system using search engines and Web indexes to gather information for the user, or they can be mobile and go out and actively search for information. Either way, their function remains the same—to deliver useful information to the user.

In the preceding sections, we have described three of the major taxonomies for classifying agents, but there are others. Some feel that collaboration with other agents is a major distinction. Others feel that whether the agent interacts directly with a human user or not is important. Still others feel that learning ability should be a primary basis for classifying agents. Whether the agent works across a network or only locally on a PC or workstation is another distinguishing attribute. Because intelligent agents exist in a multidimensional space, characterizing agents by two or three of those dimensions is somewhat risky (Nwana 1996). However, we feel that the loss of some precision is more than offset by the clarity of the two- or three-dimensional approach. People find it hard to think in six- or seven-dimensional space. In our opinion, agency and intelligence are the fundamental underlying capabilities on which agents should be classified. Mobility or some other characteristic could be used as the third axis, as required.

To be sure, there are software agents which are autonomous, but not intelligent. These agents often act as simple machine performance monitors in distributed system management applications. Simple Network Management Protocol (SNMP) agents are an example of this kind. On the other hand, there are programs or applications which use artificial intelligence techniques such as learning and reasoning, but which have relatively little autonomy. Classic expert systems are an example. They are not intelligent agents in the sense used in this book. We are interested in programs at the intersection of two domains, intelligence and agency. All other characteristics are secondary and may or may not be present for us to use the term *intelligent agent*.

Intelligent agents are software programs, nothing more and nothing less. Sometimes this has a negative impact when someone new to the topic comes to the realization that there is no magic here, just programming. However, intelligent agents, at least as we define and refer to them, are software programs

with an attitude. They exist to help users get their work done. You may say that that is what application software is supposed to do. This is true. However, many applications today assume a level of familiarity and sophistication in the users which many users are incapable or unwilling to achieve. People just want to get their job done. Most don't care if the font is TrueType or Adobe, if the component model is ActiveX or JavaBeans, or whether the code is client/server or Web-based. Toward this end, intelligent agent software is practical software. It just gets the job done. If intelligent agent software introduces another level of complexity that the user has to deal with, then it will be a failure. Intelligent agents must be enabling and automating, not frustrating or intrusive.

Summary

In this chapter, we presented an introduction to artificial intelligence and intelligent agents. The major points include:

- The field of artificial intelligence is approximately 40 years old. During that time, AI has evolved from trying to build general problem solvers using search, to building expert systems with deep but narrow domain knowledge, to combining knowledge, reasoning, and learning to solve difficult real-world problems.
- A software agent exhibits *intelligence* if it acts rationally. It uses knowledge, information, and reasoning to take reasonable actions in pursuit of a goal or goals.
- Most artificial intelligence techniques are based on *symbol processing*. A set of tokens or symbols is used to represent knowledge and the state of the world, and algorithms manipulate those symbols to mimic high-level cognitive processing functions.
- *Neural networks* are another school of artificial intelligence, based on a brainlike, adaptive parallel computing model which does not use explicit symbols. Neural networks learn or adapt when exposed to data and correspond to cognitive functions such as low-level sensory processing and pattern recognition.
- The Internet, intranets, and the Web have caused an explosion in the amount of information available to people. The easy-to-use Web browsers have also attracted many new computer users to the on-line world. These factors have contributed to create a huge opportunity for *intelligent agents*, software that helps users do complex computing tasks.
- Intelligent agents must be able to recognize events, determine the

meaning of those events, and then take actions on behalf of a user.

- Agents can be categorized by placing them in a three-dimensional space, where the axes are *agency*, the amount of autonomy an agent has, *intelligence*, the knowledge, reasoning, and learning capabilities of the agent, and *mobility*, being able to move between systems in a network.
- *Reactive agents* are relatively simple agents that sense their world, and respond reflexively to external stimuli. *Deliberative agents* are more complex, using knowledge, reasoning, and even learning to plan and achieve their goals. *Collaborative agents* work together as a team, combining their knowledge and specialized skills to solve large problems.
- Intelligent Agents can also be classified by the functions they perform. *Interface agents* work as personal assistants to help user's accomplish tasks. *Information agents* work to prevent information overload. They can either actively search out desired information on the Web, or filter out uninteresting or unwanted data as it arrives. *Domain-specific agents* can do on-line shopping, make travel arrangements, suggest a good book or CD, or help schedule a meeting.
- In the end, the success or failure of intelligent agents will depend on how much value they provide to their users.

Previous | Table of Contents | Next

Chapter 1
The Java Programming Language

Chapter 1 presents the Java programming language developed by Sun Microsystems and the functions provided by the Java Development Kit (JDK) 1.1. This chapter provides a quick overview of the language, meant to get experienced object-oriented programmers up to speed quickly on the unique aspects of the Java language. We briefly compare and contrast Java against C++ and Smalltalk as a general-purpose, object-oriented programming language and discuss how Java satisfies the requirements for intelligent agents, including autonomy, intelligence, and mobility.
Complete documentation of the Java language specifications and application programming interfaces (APIs) is available on the Web at http://www.javasoft.com. There are many excellent books on programming in Java. *Java in a Nutshell* (Flanagan 1996) and *Core Java* (Cornell and Horstmann 1996) are two that we found particularly useful.

Overview of the Java Language

Java is an object-oriented programming language developed by Sun Microsystems. It was originally designed for programming real-time embedded software for consumer electronics, particularly set-top boxes to interface between cable providers, broadcasters, and televisions or televisionlike appliances. However, the effort was redirected to the Internet when the market for set-top boxes did not develop quickly enough, while the Internet exploded in popularity.

Originally the developers of Java intended to use C++ for their software development. But they needed a language that could execute on different sets of computer chips to accommodate the ever-changing consumer electronics market. So they decided to design their own language which would be independent of the underlying hardware.

It is this “architecture-neutral” aspect of Java that makes it ideal for programming on the Internet. It allows a user to receive software from a remote system and execute it on a local system, regardless of the underlying hardware or operating system. An interpreter and runtime are called the Java Virtual Machine which insulates the software from the underlying hardware.

Unlike more traditional languages, Java source code does not get translated into the machine instructions for a particular computer platform. Instead, Java source code (.java) is compiled into an intermediate form called bytecodes which are stored in a .class file. These bytecodes can be executed on any computer system that implements a Java Virtual Machine (JVM). This portability is perhaps one of the most compelling features of the Java language, from a commercial perspective. In the current era of cross-platform application development, any tool that allows programmers to write code once and execute it on many platforms is going to get attention.

The portable, interpreted nature of Java impacts its performance. While the performance of interpreted Java code is better than scripting languages and fast enough for interactive applications, it is slower than traditional languages whose source code is compiled directly into the machine code for a particular machine. To improve performance, Just-In-Time compilers (JITs) have been developed. A JIT compiler runs concurrently with the Java Virtual Machine and determines what pieces of Java code are called most often. These are compiled into machine instructions on-the-fly so that they do not need to be interpreted each time they are encountered within a program. Static compilers are also being developed to compile the Java source code into machine code that can be executed without interpretation. It is important to note that, unlike bytecodes, the machine code generated is not portable and will not execute on other platforms.

The bytecode portability is what enables Java to be transported across a network and executed on any target computer system. Java applets are small Java programs designed to be included in an HTML (HyperText Markup Language) Web document. HTML tags specify the name of the Java applet and its Uniform Resource Locator (URL). The URL is the location on the Internet at which the applet bytecodes reside. When a Java-enabled Web browser displays an HTML document containing an applet tag, the Java bytecodes are downloaded from the specified location and the Java Virtual Machine interprets or executes the bytecodes. Java applets are what enable Web pages to contain animated graphics and interactive content.

Because Java applets can be downloaded from any system, security mechanisms exist within the Java Virtual Machine to protect against malicious or errant applets. The Java runtime system verifies the bytecodes as they are downloaded from the network to ensure they are valid bytecodes and that the code does not violate any of the inherent restrictions placed on applets. Java applets are restricted from communicating with any server other than the originating host, the one from which they were downloaded. They cannot run a local executable program or access local files. The restrictions are in place to prevent a Java applet from gaining access to the underlying operating system or data on the system. These restrictions can be eased, however, through the use of digital signatures and alternate **SecurityManager** implementations.

But Java can be used for more than programming applets to run within a browser. Java is a full-function programming language which can be used to write standalone applications. These applications are not placed under the same security restrictions as applets and therefore can access data and underlying operating system function.

Java is an object-oriented programming language, borrowing heavily from Smalltalk, Objective C, and C++. It is characterized by many as a better, safer C++. Java uses C++ syntax and is readily accessible to the large existing C++ development community. Java, however, does not drag along the legacy of C. It does not allow global variables, functions, or procedures. With the exception of a few primitive data types like integers or floating-point numbers, everything in Java is an object. Object references are not pointers, and pointer manipulation is not allowed. This contributes to the general robustness of Java programs since pointer operations tend to be particularly nasty and bug-prone. Java also manages memory itself, thereby avoiding problems with allocation and deallocation of objects. It does not allow multiple inheritance like C++ does, but supports another type of reuse through the use of formal interface definitions.

Java is similar enough to C and C++ that it already feels familiar to most of the existing programming community. But it is different enough in important ways (memory management and cross-platform portability) that it is worth it for programmers to switch to a new language.

The next section examines the Java language features in more detail.

Java Language Features

This section does not include the full language specification for Java but is intended to provide enough information for a C or C++ programmer to get a feel for Java. For a full description of the language, see the JavaSoft Web site at http://www.javasoft.com/.

	Table 1.1 Primitive Data Types	
Primitive Data Type	**Size**	**Content**
boolean	1 bit	true or false
byte	8 bits	signed integer
short	16 bits	signed integer
int	32 bits	signed integer
long	64 bits	signed integer
float	32 bits	IEEE 754 floating point
double	64 bits	IEEE 754 floating point
char	16 bits	Unicode character

Data Types

Although Java is an object-oriented language, the primitive data types are not objects. Equivalent objects are provided as part of the language, but the primitives allow faster performance by eliminating the overhead associated with objects. Table 1.1 describes the primitive (non-Object) data types available in Java.

Variable declarations are similar to those in C or C++. Java is a strongly typed language, so every variable must have a type. The type can be either a primitive data type, or a reference data type. Primitive data types in Java are passed by value. This means that the actual value is stored in a variable, and the value itself is passed on method calls. Objects, on the other hand, are reference data types. This means that the variable contains a reference to the address at which the object is stored, and the reference is passed on methods calls. Variables in Java can be initialized when declared and can be declared at any point in the code. Here are some examples of variable declarations in Java:

```
boolean flag;
char answer = 'N';
int[] arrayOfIntegers = new int[10];
String myName;
```

As seen in the example above, arrays are first-class objects in Java, and, like other objects, must be explicitly created using the *new* operator.

Control Structures

Many of Java's control structures are identical to C and C++. These include *if/else* conditional branching, *while* loops and *do* loops. Java is more restrictive than C in that the conditional expression must be a Java boolean data type. A boolean value cannot be cast to other types, so you must use comparison operators or methods which return boolean values. The following code snippet illustrates this point:

```
int i = 0;
while ( i < 10) {
  if ( i == 0)  {
```

```
    ... // first time
  } else  {
    int j = 0 ;
    do {
      ... // all other times
      j++ ;
    } while ( j < 10) ;
    i++ ;
  }
```

The Java *switch* statement also uses standard C syntax, right down to support for the *default* label. The *for* loop syntax has been slightly modified in Java. Like C++, you can define loop variables in the initialization section, but unlike C++, they are scoped to the inside of the loop only. Java allows multiple statements in the initialization and increment sections, but they must be delimited by commas.

```
for (int i=0, long sum=0 ; i < 10 ; i++) {
      sum += i ;
}
// note i and sum are out of scope
```

Java supports the *break* and *continue* statements for breaking out of enclosing loops and *switch* statements. But Java adds a twist to these functions. You can add a label to a loop by adding the label and a colon before the loop specification. This allows you to specify which enclosing loop you want to *break* or *continue* out of.

Objects, Classes, and Methods

Java is an object-oriented programming language. Thus, the way you get things done is by defining classes, instantiating objects, and calling methods on those objects. A *class* is a collection of data (called members) and methods (functions or procedures) that manipulate that data. An *object* is an *instance* of a class and we use the term interchangeably. In Java, there are no global variables or functions; everything must be defined as part of a class. Below we define a simple Java class called Agent:

```
public class Agent {
 String name ;
 Agent contact ;
 public String getName() { return name ;}
 public Agent getContact() { return contact; }
 Agent(String aName) { name = aName; ) ;
```

```
}
```

In this example, *name* and *contact* are members and *getName()* and *getContact()* are methods. *Agent(String aName)* is a *constructor* which is used to create an instance of **Agent** with the *name* member initialized to the value of *aName*. An object can be created in Java using either the *new* keyword (most common) or the *newInstance()* method on the class. For example:

```
Agent msmart,jbond;
msmart = new Agent(“Maxwell”);
jbond = Agent newInstance();
```

Here **Agent** is a class, and *msmart* and *jbond* are variables that reference or refer to instances of **Agent.** Remember, Java has no pointers that can be manipulated (and possibly misused). Variables hold elementary data types or object references, which are always valid. Note that before the second line is executed, variable *msmart* refers to no object and has a special value called *null*. A null reference will cause an exception (error) if you try to use it to refer to an object.

We can get and set data members in the object either directly, if the object permissions allow, or indirectly through methods, again if the permissions allow. The *public*, *private*, and *protected* keywords determine whether other objects can access data member or call methods on an object. For example, we could get the name of *jbond* by coding *jbond.name* to access the name directly, or by using the method call *jbond.getName()*.

In our **Agent** class, *name* and *contact* are called *instance variables*. Each instance of an **Agent** will have its own *name* and *contact* data members. We can create *class variables*, which apply to all instances of **Agent** by using the *static* keyword.

Java does not support multiple inheritance, where a class can be derived from two or more parent or base classes. Instead, it introduces the notion of an interface. An *interface* is a set of methods which define a specific behavior that is supported by an instance of the class. In some ways, the interface construct is more powerful than multiple inheritance, because it allows any class to mimic the behavior of other classes without regard to their underlying data members or implementation strategies. Smalltalk only allows single inheritance, but doesn’t have anything like the Java interface construct. C++ allows multiple inheritance, but the complexity makes using it error prone. The Java interface seems like a

reasonable compromise between these two approaches.

Applets and Applications

Java code can be run as stand alone applications or as applets, which are run in a browser as part of a Web page. Let's start with applications because they are the most familiar. Any Java class can be turned into an application by providing a *main()* method with the following signature:

```
public class AgentApp {

  public static void main(String[] args) {
     // this is an application
  }
...
}
```

A Java application can be invoked from the command line by specifying the keyword *java* followed by the name of the class: *java AgentApp*. Remember that Java is an interpreted language, so we need another program to read and interpret the bytecodes in the *AgentApp.class* file. This is the Java runtime, in the executable file java. Any parameters passed in on the command line can be accessed through the *args* String array, much like the *argv* parameter in a C or C++ *main()*.

In order to run as an applet, a class must be a subclass of the **Applet** class. An applet is designed to run under the control of another program, usually a Web browser, and make use of the browser's window to display itself and interact with the user. The *init()* method is used to process parameters and do other onetime initializations for the applet. The *start()* and *stop()* methods are called whenever the user brings up or leaves the Web page. A sample applet is shown below :

```
public class AgentApplet extends Applet {

   String name = "Double ought seven" ;
   public void Paint(Graphics g) {
   g.drawString("Hello, my name is " + name + "!", 100,100) ;
   }
   public void init() {
      // for an applet, init is like the main()
      //  it is the first applet method called
   }
```

```
    public void start() {
        // called after init, and when user returns to Web page
    }
    public void stop() {
        // called when user moves off the Web page
    }
}
```

Previous | Table of Contents | Next

Because the Web browser actually calls the applet, we need an HTML file that references the applet and specifies its parameters, such as:

```
<APPLET; code="AgentApplet.class" width=300 height=200>
<PARAM;  name="contact" value="chief">
</APPLET>
```

Notice that along with the name of the applet *.class* file, you specify the height and width of the display area which the applet will require on the current Web page. You can also specify additional parameters in the HTML using the *<PARAM>* tag, which the applet can retrieve via the *getParameter()* method. The applet can retrieve this value by calling :

```
String param = this.getParameter("contact");
```

The Java runtime provides an applet viewer which can be used to run applets as standalone applications. The applet viewer takes the role of the browser in this case. Applets should also implement the *getAppletInfo()* and *getParameterInfo()* methods to provide information about the applet and its parameters to users.

Applets are one of the prime reasons people refer to Java as an Internet programming language. Because they were first used to spice up dull Web pages with animated text and graphics, Java applets are the most well-known feature of the Java language. Applets are now being used to create dynamic data entry panels for browser-based Internet applications. They have also emerged as a key technology behind the development of *network computers* as possible replacements for dumb terminals.

Java Native Interface

The Java Native Interface (JNI) provides a standard technique for Java to call out to other language environments and for the Java Virtual Machine to be invoked from programs written in other languages. A common use of JNI is to provide Java *native methods* or "wrappers" for existing C or C++ APIs. This allows Java programmers to treat those APIs as simple extensions to the Java environment. But what actually occurs under the covers is that the JVM transfers control into the other language environment (a DLL or UNIX shared library) and passes a pointer to the running Java environment.

When a Java program calls a *native* method, the parameters (either primitive data types such as int, long, or float, or references to Java objects) are passed into the C or C++ API function. The *native* method can then use the Java environment pointer and a set of JNI functions to call methods on those Java objects. Thus, a Java method call is turned into a set of parameters and the underlying C or C++ API function is called. The API results are then returned to the Java environment, either directly or by called methods to set data members on Java objects. The JNI also provides an interface so that a C or C++ program can start up the JVM and have it load and run a Java program.

Other Java Language Features

An interesting feature of Java is that it uses the 16-bit Unicode character set, rather than the 8-bit ASCII encoding. Unicode defines over 34,000 coded characters covering the major written languages from around the world. This built-in national language support (NLS) capability, combined with some of the locale() or location sensitive processing introduced in Java 1.1, means that Java applets could be written for a worldwide audience.

Because the Java runtime supports multiple threads of control, the language specification includes a *synchronized* statement which can be used to mark critical sections of code to prevent corruption of objects. Threading is an important feature for network computing applications, because it allows server programs to service multiple clients by spinning new threads for each request. This approach gives much better performance than starting a new process (like CGI-BIN programs) when a request comes in.

Packages

Sets of related classes are grouped together into Java *packages*. In the same way a Java class name maps to identically named source (.java) and class (.class) files, Java packages are mapped to identically named directories. All .class files which are part of a package must reside in the same directory. A package such as *java.awt.image* maps to a directory structure of *java/awt/image*.

Java supports grouping collections of Java packages and classes into zip files in uncompressed format. A new packaging format called JAR (Java ARchives) can also be used to group collections of Java packages into a single file so that a

complete set of Java classes can be downloaded from a Web server in a single transaction.

java.lang

The *java*.*lang* package contains the basic Java language classes. These include classes for objects and classes themselves (**Object, Class)**, some basic data types (**Boolean, Byte, Character, Double, Float, Integer, Long, Math, Number, Short, String, StringBuffer, Void)**, security (**SecurityManager**), threads (**Thread, ThreadGroup**), and some other classes related to Java compilation and runtime. Note that these classes define only the very essential behavior of classes in the Java language. More complex and powerful classes based on the classes in *java*.*lang* exist in other packages. The *java*.*lang* package also includes the **Cloneable** interface which is used to indicate whether an object can be cloned, and the **Runnable** interface which indicates that a class should be executed by a thread.

This the only package that is automatically imported into every Java program. The **Object** class is the root of all class hierarchies. Because every other Java object is a subclass of **Object**, all objects can invoke the *public* and *protected* methods of the **Object** class. These include tests for equality, thread synchronization on objects, hash code generation, and identification of the **Class** to which an object belongs. Some of these methods should be overridden in subclasses to provide the appropriate behaviors. A **Class** object represents a Java class and can provide information about the class itself including the class name, the superclass of the class, and the interfaces implemented by the class.

The majority of the data type classes are fairly self-explanatory. The **String** and **StringBuffer** classes are of some interest because **String** objects in Java are immutable, which means that strings have constant values that cannot be changed once they are defined. **StringBuffer** objects are used for modifiable string values.

The **SecurityManager** class is used primarily by Web browsers and applet viewers when loading untrusted code. A **SecurityManager** object implements a given security policy. Before an operation is invoked, a **SecurityManager** object can check to see if the caller is allowed to perform the operation. The default applet security policy assumes that the caller does not have permission to perform all but the most benign operations.

java.lang.reflect

The *java.lang.reflect* package includes classes to support introspection about the classes and objects currently running in a Java Virtual Machine. These include classes to create and access arrays (**Array**) and to provide information and access to classes, constructors, fields, methods, and modifiers (**Class, Constructor, Field, Method, Modifier**). Note that **Class** supports reflection methods but is actually part of *java.lang* for backward compatibility with earlier versions of Java.

java.io

The *java.io* package contains a large number of classes, most of which are descended from **InputStream** or **OutputStream**. The stream classes define methods for reading and writing data from a variety of streams. In addition to stream I/O, *java.io* includes classes for file I/O and filename filtering (**File, FilenameFilter, RandomAccessFile**). The *java.io* package also includes a number of interfaces. Two of the more interesting interfaces are **Serializable** and **Externalizable**. The **Serializable** interface allows an object to write its state to a stream and read it back again. This is useful for passing objects as parameters in a distributed architecture (like RMI) and for making an object's state persistent. The **Externalizable** interface is also used for streaming objects, but the implementer has more control over the format and the contents of the stream.

Previous | Table of Contents | Next

java.util

The *java.util* package contains many of the data structures commonly used in application programs. These include classes and interfaces for date and time *manipulation* (**Calendar, Date, GregorianCalendar, SimpleTimeZone, TimeZone**), common data *structures* (**BitSet, Dictionary, Enumeration, Hashtable, Random, Stack, StringTokenizer, Vector**), notification (**EventListener, EventObject, Observable, Observer**), and properties, locales, and resource *bundles* (**ListResourceBundle, Locale, Properties, PropertyResourceBundle, ResourceBundle**). While most of these are self-explanatory, a few of these deserve a closer look.

The notification classes and interfaces can be used to implement the observer design pattern (Gamma et al., 1995), the most familiar of which is the model/view paradigm. An **Observable** object is one whose changes need to be observed by other objects in an application. Objects that wish to be notified when the **Observable** object changes must implement the **Observer** interface and must register themselves with the **Observable** object.

The **Properties** class allows key/value pairs to be read from and written to a stream. **Properties** objects can be used to look up customizable values based on a key, similar to environment variables in many operating systems. If an application is to be run in different locales, properties can be loaded into a **PropertiesResourceBundle**, which is a subclass of **ResourceBundle**. The **ResourceBundle** class contain locale-specific objects which can be loaded as appropriate for the current **Locale**. This allows an application to be written independent of the user's **Locale**. These classes are very useful when writing code that can be downloaded from the Web and executed anywhere in the world. The **ResourceBundle** and **Locale** classes, along with classes in the *java.text* package, provide Java's internationalization support.

java.util.zip

The *java.util.zip* package provides support for ZIP files compression and decompression. This includes classes to compress and decompress the data (**Deflater, DeflaterOutputStream, GZIPInputStream, GZIPOutputStream,**

Inflater, InflaterInputStream, ZipEntry, ZipFile, ZipInputStream, ZipOutputStream) as well as classes and interfaces for doing checksums on those files (**Adler32, CRC32, CheckedInputStream, CheckedOutputStream, Checksum**).

java.net

The *java.net* package provides a set of classes for communications across networks. This includes classes for working with Internet addresses and Uniform Resource Locators (URLs) (**INetAddress, URL, URLConnection, URLEncoder, URLStreamHandler**, and **URLStreamHandlerFactory**) and sockets (**Socket, ServerSocket, DatagramSocket, SocketImpl**, and **SocketImplFactory**) as well as data (**ContentHandler, ContentHandlerFactory, DatagramPacket**). These classes allow Java programs to work at any of three logical levels over networks, at the URL read an HTML page level, at the raw sockets level, and at the even lower datagram level. This power and flexibility is not surprising given Java's network-centric focus.

java.awt

The Abstract Windowing Toolkit (**awt**) is an object-oriented framework for creating graphical user interfaces (GUI) in Java. It provides classes for basic window components, for drawing graphics, and for automatically arranging and resizing components. Unlike C and C++, which did not address GUI functions, Java provides this standard package so that Java applets and applications can run on multiple operating systems with very different underlying GUI support and have a reasonably common look and feel.

The *java.awt* package is quite large with over 100 classes and interfaces in the base *awt, java.awt.event, java.awt.image,* and *java.awt.datatransfer* packages. The awt package supports most standard GUI controls (**Window, Panel, Dialog, Frame, Canvas, ScrollBar, Menu, MenuBar, MenuItem, Button, Checkbox, Choice** (listboxes), **Label, TextField**, and **TextArea**). The graphics support includes (**Font, FontMetrics, Graphics, Image, Point, Polygon**, and **Rectangle**).

The JDK 1.1 event model supports a Smalltalk-like model-view-controller

paradigm, where the underlying data is stored in the model, the view is the graphical controls, and the controller is a class which listens for events and keeps the model and view synchronized. This Delegation Event Model is also used as the basis for the JavaBeans event model and is extremely important for building Java GUIs and reusable software components.

java.applet

The Java applet package contains a single class, **Applet**, which is a subclass of **awt.Panel**, and three interfaces, the **AppletContext, AppletStub**, and **AudioClip**. The AppletContext defines the environment the applet is running in, the HTML document it was loaded from, and other applets in that document. The **AppletContext** provides a way for one applet to get references to other applets on the same Web page, and to interact with them. The **AppletStub** is the interface between the applet and the browser or applet viewer. Applets were discussed in an earlier section on applets and applications.

java.text

Internationalization support for applets and applications is provided by the *java.text* package. This package contains classes and interfaces for handling text in a locale-specific way. Classes are provided for formatting dates, decimals, messages, and numbers in a way that conforms to local language conventions (**ChoiceFormat, DateFormat, DateFormatSymbols, DecimalFormat, DecimalFormatSymbols, FieldPosition, Format, MessageFormat, NumberFormat, ParsePosition, SimpleDateFormat**). Interfaces and classes are also provided for collating and iterating over internationalized text strings (**BreakIterator, CharacterIterator, CollationElementIterator, CollationKey, Collator, RuleBasedCollator, StringCharacterIterator**).

java.security

Security is a major concern with Internet applications and Java provides built-in architectural support for applet and application security. The *java.security* package supports signed applets and JAR (Java archive) files, key encryption, and access control lists. The **Security** class manages so-called security providers and centralizes all security related functions. A security **Provider** is a named implementation of some or all of the Java security APIs. The SUN provider is

the default and implements the Digital Signature Architecture (DSA) encryption algorithm.

Previous | Table of Contents | Next

java.beans

The *java.beans* package contains the JavaBeans reusable software component model, a collection of approximately 20 classes and interfaces. JavaBeans can range from small, lightweight GUI controls to relatively large components such as spreadsheets or word processors. Besides providing a component specification for Java applications, JavaBeans will be able to interoperate with Microsoft ActiveX components. Platform portability was an important design goal for the JavaBeans architecture.

"A Java Bean is a reusable software component that can be manipulated visually in a builder tool" (JavaSoft 1996). Each Bean provides sets of properties, methods, and events for interaction with other Beans. Beans support introspection, which allows visual application builders to analyze how each Bean works, and customization, so that the Bean's behavior can be extended in a visual builder environment. Once a Bean is customized and combined with other Beans in an application, it supports persistence, so that it can be saved and restored later.

JavaBeans interact through a sophisticated event model that features **EventSource** objects which can fire events and **EventListener** objects which can receive those events. Through introspection, Beans can determine what events other Beans can generate or listen to. There is an event registry which allows Beans to dynamically change their roles as event generators or listeners. **EventAdapter** objects can be inserted between an event source and an event listener to filter events, queue events, or to act as a centralized "switchboard" for inter-Bean communications.

java.rmi

The *java.rmi* package is used to create distributed object applications or applets in Java. Remote Method Invocation (RMI) uses a local surrogate or stub to manage the invocation of methods on a remote object. Any object that can be accessed remotely must implement the **Remote** interface. A Naming object provides the initial bootstrap mechanism for obtaining remote object references. In addition, a **RMISecurityManager** is provided which defines the security

policy for stubs in distributed applications.

There are a number of subpackages related to RMI. The *java.rmi.dgc* package provides classes for distributed garbage collection of inactive objects. The *java.rmi.registry* package provides interfaces and classes for the registry that exists on every node in a distributed system (**Registry, RegistryHandler, LocateRegistry**). A registry is used to map names to remote object references. The *java.rmi.server* package contains the rest of the interfaces and classes that implement RMI. These include interfaces for calls to remote objects (**RemoteCall**) and for remote object references (**RemoteRef, ServerRef**). Classes are defined for remote objects, servers, and stubs (**RemoteObject, RemoteServer, RemoteStub, UnicastRemoteObject**) as well as object identity (**ObjectID, UID**), method descriptions (**Operation**), and logging of errors (**LogStream**).

java.sql

The *java.sql* package is better known as JDBC (Java DataBase Connectivity). This package allows database access from Java and is based on the X/Open SQL CLI. A **DriverManager** class is provided which is responsible for loading the correct **Driver** implementation for a given database and will create a *Connection* to a particular database. A **Driver** implementation for ODBC databases is provided as part of the JDK, and other database vendors provide drivers for their particular database implementations. Interfaces are provided for executing SQL statements against a database (**CallableStatement, PreparedStatement, Statement**), and for returning the results of the database queries (**ResultSet**). Classes are also provided for the SQL data types (**Date, Time, Timestamp, Types**). In addition, support is provided to query the underlying database and database driver to find out what features are supported (**DatabaseMetaData, DriverPropertyInfo, ResultSetMetaData**). The overall result is a package that allows Java code to access data in different databases from different vendors using a common programming interface.

Java Development Environments

The minimum requirement is to get the Sun Java Development Kit (JDK), which provides a command line Java compiler, a rudimentary debugger, and a base implementation of the Java Virtual Machine. However, most developers are

accustomed to using integrated development environments which feature GUI builders, source code debugging, project management for source code control, and source code browsers and editors. Most also include JIT compilers for better performance. There are several companies which sell integrated development environments for Java. These include Symantec, with its set of Café products, Sun with its Java Workshop tools, IBM with its VisualAge for Java, and Microsoft, with its J++ development tool. The beauty of Java and its portable virtual machine interface is that it doesn't really matter which vendor or tool you choose to develop your Java applications and agents. As long as you use the classes which are part of the standard language and utilities, it will be able to run on any other platform. A cautionary note is that Microsoft has introduced proprietary elements into its Java tools and implementation. Be careful that you do not rely on these features, or you will be limited to Microsoft Windows platforms.

Notes from a C++ Programmer

As an experienced C++ programmer, from everything that I had read about Java, I understood that programming in Java would be no big deal. After several days of coding, I can confirm that Java is a comfortable and very productive programming environment. To give a little better perspective on where I am coming from, let me briefly describe my C++ experience. I have designed and written several development tools in C++ using the Star Division, Inc. *StarView* GUI class libraries (for OS/2, Windows 3.1, and AIX) as well as the IBM OpenClass (also known as ICLUI) class libraries. I have also used the standard template-based collection class libraries which are now standard in the C++ world. So, here are some Java novice comments from an experienced C++ programmer:

Wow! Writing new code in Java reminds me of earlier days using the Turbo Pascal products. I am using the Symantec Café development environment which is nice, but not outstanding in any way. What impresses me most is the base power of the Java language and utility classes. It took some getting used to for me not to sketch out my classes in header files and then create method stubs in a cpp file. In Java you must define the methods inside the class definition (unless you are using interfaces, which is something we'll get to later). Another problem I hit was that I wanted to make all of my helper classes public, but Java won't let me. They all have to be in separate .java files. Since I was just prototyping some of the AI code, I wanted to just use one program

file. I did this by making all of the helper classes nonpublic. Once I write more code, I will pull out the common objects into their own classes.

What impresses me about Java? The Vector class seems to be an extremely usable List class which can hold objects of any type and which can be used as a queue. I especially like the Enumeration interface which Vector (and other classes) support. With this you can do a *while()* loop with *list.hasMoreElements()* and *list.nextElement()*. I can't tell you how many hours it took to get my first C++ template-based class to compile and link. This was a breeze.

Another thing I had to do was parse some text. In C I would use *strtok()* function, which is not pretty to work with. In IBM OpenClass I would use an IString with a *word()* interface. In Java, I used a StringTokenizer which supports the Enumeration interface. Parsing a string into tokens was as easy as a *while() hasMoreElements()* and *nextElement()* loop. Cool.

I had some problems with the Java "everything is a reference" paradigm for objects. When I code a class in C++, I can specify that objects are contained or referenced. C++ knows enough to call the default constructor for data members which are contained (not pointers to instances). But since Java has no notion of pointers, all data members are implicitly "by reference." This means that unless you do a *new classXYZ()* in your constructor, the first time you try to reference any data member objects, you will get a null pointer exception.

Another not-so-obvious "feature" of Java is the immutable wrapper classes for the primitive data types. You can code things as *int, float*, and *double* just like in C or C++. Java kindly provides **Integer, Float**, and **Double** classes so you can turn these values into full-fledged Java objects. The only problem is that they are immutable (they can't be changed once they are created). I tried to pass an Integer into a method and use it as an output variable. But that didn't work because I couldn't change it. A setValue() method on the wrapper class sure would be nice.

One last comment on Java versus C++ as a development language: I don't miss the long edit/compile/link cycles at all. What a pleasure to be able to make a quick change and immediately compile and run the changed code. Not as nice as Smalltalk, but a tremendous advance over C++ productivity.

Using Java for Intelligent Agents

In the previous sections of this chapter, we have described the major elements of the Java programming language. In this section, we talk about specific features of Java which support intelligent agent applications. These features will be explored in more detail in the remainder of the book.

Notes from a Smalltalk Programmer

As an experienced Smalltalk programmer, I wasn't expecting much from Java. To me, the Smalltalk programming environment is the most productive I have ever experienced. I know that Java is supposed to be hot, but I really couldn't see it outshining Smalltalk in my heart. My Smalltalk programming experience started in 1987, when I was working on my Master's degree at Lehigh University in Bethlehem, PA. I had written most of my thesis on various types of neural networks and I wanted to implement a neural network development and test environment. My thesis advisor, Ed Kay, was teaching a Smalltalk class and was being funded by Tektronix (we had 3 or 4 Tek Smalltalk-80 workstations). So, that September I started simultaneously learning Smalltalk and OO programming techniques. It was quite an experience. Later on, we used Digitalk Smalltalk/V to implement the GUI on the Neural Network Utility Version 2 product, which was released in mid-1991. This was one of the first IBM products shipped which was written in Smalltalk. More recently, I have used IBM VisualAge for Smalltalk to create a prototype interface to the NNU V3 APIs. This reinforced my perception that Smalltalk is a superior programming environment to C++ and also convinced me that visual programming could be extremely productive. Earlier, I said that I wasn't expecting much from Java, when compared to Smalltalk. This is because Java relies on the C++ syntax, which is even worse than C. But I know that Java is object-based (ignoring those int and float types for the moment) and that it has garbage collection instead of those wonderful new/delete methods in C++. Once you learn the Smalltalk order of precedence (unary, binary, message) for expression evaluation, it is the most natural way to think about OO code, so I couldn't image what Java could offer.

Well, after using Java I can say that it has some nice things. The first of which

is that the code is comprehensible for C and C++ programmers (sadly, Smalltalk takes some getting used to). I also found the class hierarchy to be less complex and more straightforward to use than the traditional Smalltalk classes.

While Java might look more like C++, it thinks (feels and acts) more like Smalltalk. I love being able to make a Vector or Stack of objects, and being free to put any type of object into it. Of course you have to know what type of object it is when you access it, but like Smalltalk, Java lets you ask the object "what type it is." This seemingly simple function adds lots of power to the language.

Autonomy

For a software program to be autonomous, it must be a separate process or thread. Java applications are separate processes and as such can be long-running and autonomous. A Java application can communicate with other programs using sockets. In an application, an agent can be a separate thread of control. Java supports threaded applications and provides support for autonomy using both techniques.

In the Introduction, we described intelligent agents as autonomous programs or processes. As such, they are always waiting, ready to respond to a user request or a change in the environment. One question that comes to mind is "How does the agent know when something changes?" In our model, as with many others, the agent is informed by sending it an event. From an object-oriented design perspective, an event is nothing more than a method call or message, with information passed along on the method call which defines what happened or what action we want the agent to perform, as well as data required to process the event.

In Java there is an event-processing mechanism which is used in the Abstract Windowing Toolkit (AWT) to pass user-defined events such as mouse movements and menu selections to the underlying window components. Depending on the type of agent we are building, we may need to process these low-level events, such as when a GUI control gets focus, or a window is resized or moved, or the user presses a key. The *awt.event* package also supports higher-level semantic events such as user actions.

Java also supports another higher-level event interface called the

Observable/Observer framework. In this approach, interested objects or Observers register themselves with the **Observable object**. Whenever a method is called on the **Observable** or model object that causes a significant state change or something equivalent to a high-level semantic event, then all of the registered **Observer** objects are notified.

Previous | Table of Contents | Next

Intelligence

The intelligence in intelligent agents can range from hardcoded procedural or object-oriented logic to sophisticated reasoning and learning capabilities. While Prolog and Lisp are the two languages usually associated with artificial intelligence programming, in recent years, much of the commercial AI work has been coded in C and C++. As a general-purpose, object-oriented programming language, Java provides all of the base function needed to support these behaviors.

There are two major aspects to AI applications, knowledge representation and algorithms which manipulate those representations. All knowledge representations are based on the use of slots or attributes which hold information regarding some entity, and links or references to other entities. Java objects can be used to encode this data and behavior as well as the relationships between objects. Standard AI knowledge representation such as frames, semantic nets, and if-then rules can all be easily and naturally implemented using Java.

Mobility

There are several different aspects to mobility in the context of intelligent agents and intelligent applications. Java's portable bytecodes and JAR files allow groups of compiled Java classes to be sent over a network and then executed on the target machine. Java applets provide a mechanism for running Java code remotely via a Web browser. Other environments, such as the IBM aglets, allow Java processes to be started, suspended, and moved.

One of the prime requirements for mobile programs is the ability to save the state of the running process, ship it off, and then resume where the process left off, only now it is running on a different system. Computer science researchers have explored this topic in great detail in relation to load balancing on distributed computer systems such as networks of workstations. Having homogeneous machines was a crucial part of making this work. Once again, the Java Virtual Machine comes to the rescue. By providing a standard computing environment for a Java process to run in, the JVM provides a homogeneous virtual machine that allows Java agents to move between heterogeneous

hardware systems (from a PC to a Sun workstation, to an AS/400 system) without losing a beat.

Other aspects of Java also enable mobility. The Delegation Event Model described earlier allows dynamic registration of **EventSources** and **EventListeners**. Thus a mobile Java agent could "plug in" to an already-running server environment when it arrives, and then "unplug" itself, when it is time to move on. The *java.net* package provides network communications capability which allows mobile agents or a mobile agent infrastructure to talk to other servers and send Java code and process state over sockets.

Summary

In this chapter we described the major features of the Java programming language. We also compared it to C++ and Smalltalk and evaluated it as a language for implementing intelligent agent applications. The major points include the following:

- Java is a portable, architecture-neutral language, because it is compiled into *bytecodes* which are then interpreted and run by the *Java Virtual Machine*. Any system that supports the Java Virtual Machine can run any pure Java program.
- *Just-In-Time compilers* are used to convert bytecodes into machine language to improve performance. *Static compilers* are also available to turn Java source code directly into platform-specific executables.
- Java does not provide pointers or operators for manipulation of pointers. All memory allocation is controlled by the programmer using the *new* operator. Object storage (memory) is recovered by an automatic garbage collection function.
- Java *applets* are small Java programs which can be downloaded from a server to a client and run by a Web browser as part of an HTML page. For security reasons, applets are restricted from accessing local resources. However, Java supports standalone *applications* which are equivalent to standard C or C++ programs, and which have unrestricted access to system resources.
- Java is an object-oriented programming language with C++ syntax. A Java *class* contains data *members* and *methods* that define the state and behavior of the *instances* or *objects* of that class. In addition to objects, Java supports elementary data types for performance.

- The Java *Native Method Interface* allows programs running the Java Virtual Machine to call application program interfaces written in C and C++. It also allows programs written in other languages to start up the Java VM and run Java programs.
- Java uses the *Unicode* 16-bit character set for encoding text. The Unicode format supports most of the major languages in the world, including Japanese, Korean, and Chinese.
- The Java language is extended through a set of *packages*, which contain supporting classes for a wide variety of functions, including network communications (*java.net*), security (*java.security*), graphical user interfaces (*java.awt*), remote method calls (*java.rmi*), and database access (*java.sql*).
- *JavaBeans* is the Java software component architecture. Beans can be used in a visual builder environment to construct applications out of existing software components, including ActiveX components.
- Java provides all of the functionality required to design and implement intelligent agents. Its general-purpose, object-oriented language capabilities allow knowledge to be represented and reasoning and learning algorithms to be implemented with ease. Java's portable bytecodes allow agents to be packaged as applets or as mobile Java programs.

Exercises

1. If you haven't already done so, get started learning Java. The Web site at http://java.sun.com has an excellent on-line tutorial on Java including descriptions of the JDK 1.1 enhancements.
2. Write a hello world application using any Java development tool. What makes a Java class an application? Pass your name into the application so it displays "Hello <<your; name>>!"
3. Write a hello world applet. Run it using the applet viewer or your favorite Web browser. Pass your name into the applet so it displays "Hello <<your; name>>!" Can you pass in any type of parameter value?

Chapter 2
Problem Solving Using Search

In this chapter we explore the first major thrust of artificial intelligence research, problem solving using search. We discuss how problems can be represented as states and then solved using simple brute-force search techniques or quite sophisticated heuristic search methods. A Java applet is developed which implements four basic search algorithms.

Defining the Problem

The focus of much AI research has been on solving problems. While some critics have argued that AI has focused on unrealistically simple or toy problems in the past, that is certainly not the case today. In hindsight, it is not clear that the "toy problem" criticisms were justified at all. Much of the point of the AI research was to understand "how" to solve the problem, not just to get a solution. So seemingly simple problems—puzzles, games, stacking blocks—were the focus of AI programs. And one of the first areas of work, general problem-solving methods, highlighted a major barrier to artificial intelligence software. How do you represent a problem so that the computer can solve it? Even before that, how do you define the problem with enough precision so that you can figure out how to represent it?

While "knowing what business you are in" is one of the elementary business maxims, for artificial intelligence it is "knowing what problem you are trying to solve." For some people, solving the problem successfully is the only goal. Why use a computer unless it can help you solve business problems (and make money)? For others, the challenge is to reproduce human problem-solving techniques or, at least, gain a better understanding of how people solve complex problems. Today, very few people would claim that search-based methods demonstrate how our brain solves problems, but these methods have proven extremely useful and have a place in any discussion of practical AI techniques.

The first step in any problem-solving exercise is to clearly and succinctly define

what it is we are trying to do. Do we want to find the best possible route for a trip or just one that will get us there? Can we wait several hours or days for an answer, or do we need the best answer we can compute in ten seconds? Someone once said that in AI the most important part of problem solving is "representation, representation, representation." In this chapter, we are interested in how to represent our problem so that we can solve it using search techniques, and which search techniques to use. In the next section, we explore one of the primary problem representations, the state-space approach.

State Space

Suppose that the problem we want to solve deals with playing and winning a game. This game could be a simple one such as tic-tac-toe or a more complex one such as checkers, chess, or backgammon. In any case, the key to approaching this problem with a computer is to develop a mapping from the game-space world of pieces and the geometric pattern on a board, to a data structure which captures the essence of the current game-state. For tic-tac-toe, we could use an array with nine elements, or we could define a 3 by 3 matrix with 1s and 0s to denote the Xs and Os of each player.

We start with an *initial state*, which in the case of tic-tac-toe could be a 3 by 3 matrix filled with spaces (or empty markers). For any state of our game board, we have a set of *operators* which can be used to modify the current state, thereby creating a new state. In our case, this would be a player marking an empty space with either an X or an O. The combination of the initial state and the set of operators make up the *state space* of the problem. The sequence of states produced by the valid application of operators from the initial state is called the *path* in the state space. Now that we have the means to go from our initial state to additional valid game states, we need to be able to detect when we have reached our *goal* state. In tic-tac-toe a goal state is when any row, column, or diagonal consists of all Xs or Os. In this simple example, we can check the tic-tac-toe board to see if either player has won by explicitly testing for our goal condition. In more complicated problems, defining the *goal test* may be a substantial problem in itself.

In many search problems, we are not only interested in reaching a goal state, we would like to reach it with the lowest possible cost (or the maximum profit). Thus, we can compute a cost as we apply operators and transition from state to state. This *path cost* or cost function is usually denoted by *g*. Given a problem

which can be represented by a set of states and operators and then solved using a search algorithm, we can compare the quality of the solution by measuring the path cost. In the case of tic-tac-toe we have a limited search space. However, for many real-world problems the search space can grow very large, so we need algorithms to deal with that.

Effective search algorithms must do two things: cause motion or traversal of the state space, and do so in a controlled or systematic manner. Random search may work in some problems, but in general, we need to search in an organized, methodical way. If we have a systematic search strategy that does not use information about the problem to help direct the search, it is called *brute-force*, *uninformed*, or *blind* search. The only difference between the different brute-force search techniques is the order in which nodes are expanded. But even slight changes in the order can have a significant impact on the behavior of the algorithm. Search algorithms which use information about the problem, such as the cost or distance to the goal state, are called *heuristic*, *informed*, or *directed* search. The primary advantage of heuristic search algorithms is that we can make better choices concerning which node to expand next. This substantially improves the efficiency of the search algorithms.

There are several aspects of the performance of a search algorithm which are important to recognize (Russell and Norvig 1995). An algorithm is *optimal* if it will find the best solution from among several possible solutions. A strategy is *complete* if it guarantees that it will find a solution if one exists. The complexity of the algorithm, in terms of *time complexity* (how long it takes to find a solution) and *space complexity* (how much memory it requires), is a major practical consideration. Having an optimal search algorithm that runs forever has little practical value. Nor does having a complete algorithm that is a memory hog, if it runs out of memory just before it finds the solution.

Search Strategies

In this section we examine two basic search techniques used to solve problems in AI, breadth-first search and depth-first search. Later we will explore enhancements to these algorithms, but first we need to make sure we understand how these basic approaches work. We will work through several examples, using the map shown in Figure 2.1.

First, we need to define the problem we are trying to solve. In this case, it is

quite simple. Given a starting point at any one of the cities on the map, can we find any other city on the map as long as there is a path from the start city to the end or goal city? At this time, we are not trying to find the shortest path or anything like that. We simply want to find whether the goal city is on the map.

Okay, now that we have defined the problem, the next step is to decide how to represent the problem as a state-space. A map like the one in Figure 2.1 can be naturally represented by a graph data structure, where the cities names are the nodes, and the major roadways between cities are the links or edges of the graph. So, from a programming perspective, our problem is to traverse a graph data structure in a systematic way until we either find the goal city or exhaust all possibilities. Hopefully having the entire state-space shown on a map will make understanding the operations of the search algorithms easier. In more complex problems, all we have is the single start state and a set of operators which are used to generate more and more new states. The search algorithms work the same way, but conceptually, we are growing or expanding the graph, instead of having it specified at the start.

Figure 2.1 Map of midwestern U.S. cities.

Breadth-First Search

The breadth-first search algorithm searches a state-space by constructing a hierarchical tree structure consisting of a set of nodes and links. The algorithm defines a way to move through the tree structure, examining the values at nodes in a controlled and systematic way, so that we can find a node which offers a solution to the problem we have represented using the tree-structure.

The algorithm follows:

1. Create a queue and add the first Node to it.
2. Loop :

    - If the queue is empty, quit.

- Remove the first Node from the queue.
- If the Node contains the goal state, then exit with the Node as the solution.
- For each child of the current Node: Add the new state to the back of the queue.

Previous | Table of Contents | Next

The breadth-first algorithm spreads out in a uniform manner from the start node. From the start, it looks at each node one edge away. Then it moves out from those nodes to all nodes two edges away from the start. This continues until either the goal node is found or the entire tree is searched. Breadth-first search is complete; it will find a solution if one exists. But it is neither optimal in the general case (it won't find the best solution, just the first one that matches the goal state), nor does it have good time or space complexity (it grows exponentially in time and memory consumption).

Let's walk through an example to see how breadth-first search could find a city on our map. Our search begins in **Rochester**, and we want to know if we can get to **Wausau** from there. The **Rochester** node is placed on the queue in step 1. Next we enter our search loop at step 2. We remove Rochester, the first node from the queue. **Rochester** does not contain our goal state (**Wausau**) so we expand it by taking each child node in **Rochester**, and adding them to the back of the queue. So we add **[Sioux Falls, Minneapolis, LaCrosse, and Dubuque]** to our search queue. Now we are back at the top of our loop. We remove the first node from the queue (**Sioux Falls**) and test it to see if it is our goal state. It is not, so we expand it, adding Fargo and Rochester to the end of our queue, which now contains **[Minneapolis, LaCrosse, Dubuque, Fargo, and Rochester]**. We remove **Minneapolis**, the goal test fails, and we expand that node, adding **St.Cloud, Wausau, Duluth, LaCrosse**, and **Rochester** to the search queue, now holding **[LaCrosse, Dubuque, Fargo, Rochester, St.Cloud, Wausau, Duluth, LaCrosse, and Rochester]**. We test **LaCrosse** and then expand it, adding **Minneapolis, GreenBay, Madison, Dubuque**, and **Rochester** to the list, which has now grown to **[Dubuque, Fargo, Rochester, St.Cloud, Wausau, Duluth, LaCrosse, Rochester, Minneapolis, GreenBay, Madison, Dubuque, and Rochester]**. We remove **Dubuque** and add **Rochester, LaCrosse**, and **Rockford** to the search queue.

At this point, we have tested every node which is one level in the tree away from the start node (**Rochester**). Our search queue contains the following nodes: **[Fargo, Rochester, St.Cloud, Wausau, Duluth, LaCrosse, Rochester, Minneapolis, GreenBay, Madison, Dubuque, Rochester, Rochester, LaCrosse, and Rockford]**. We remove **Fargo**, which is two levels away from **Rochester**, and add **Grand Forks, St. Cloud**, and **Sioux Falls**. Then we test and expand **Rochester** (Rochester to Minneapolis to Rochester is two levels away

from our start). Next is **St. Cloud**; again we expand that node. Finally, we get to **Wausau**; our goal test succeeds and we declare success. Our search order was Rochester, Sioux Falls, Minneapolis, LaCrosse, Dubuque, Fargo, Rochester, St. Cloud, and Wausau.

Note that this trace could have been greatly simplified by keeping track of nodes which had been tested and expanded. This would have cut down on our time and space complexity, and still been a complete algorithm because we would have searched every node before we stopped. However, in more realistic problems where each node is expanded using a list of operators and the states are more complex than just strings, it is not so easy to determine that two states are identical. This explosion of nodes and states is not unusual for a breadth-first search problem.

Depth-First Search

Depth-first search is another way to systematically traverse a tree structure to find a goal or solution node. Instead of completely searching each level of the tree before going deeper, the depth-first algorithm follows a single branch of the tree down as many levels as possible until we either reach a solution or a dead end. The algorithm follows:

1. Create a queue and add the first SearchNode to it.
2. Loop :

    - If the queue is empty, quit.
    - Remove the first SearchNode from the queue.
    - If the SearchNode contains the goal state, then exit with the SearchNode as the solution.
    - For each child of the current SearchNode: Add the new state to the front of the queue.

Notice that this algorithm is identical to the breadth-first search with the exception of step 2d. The depth-first algorithm searches from the start or root node all the way down to a leaf node. If it does not find the goal node, it backtracks up the tree and searches down the next untested path until it reaches the next leaf. If you imagine a large tree, the depth-first algorithm may spend a large amount of time searching the paths on the lower left when the answer is really in the lower right. But since depth-first search is a brute-force method, it

will blindly follow this search pattern until it comes across a node containing the goal state, or it searches the entire tree. Depth-first search has lower memory requirements than breadth-first search, but it is neither complete nor optimal.

As we did above, let's walk through a simple example of how depth-first search would work if we started in **Rochester** and wanted to see if we could get to **Wausau**. Starting with **Rochester**, we test and expand it, placing **Sioux Falls**, then **Minneapolis**, then **LaCrosse**, then **Dubuque** at the front of the search queue **[Dubuque, LaCrosse, Minneapolis, Sioux Falls]**. We remove **Dubuque** and test it; it fails, so we expand it adding **Rochester** to the front, then **LaCrosse**, then **Rockford**. Our search queue now looks like **[Rockford, LaCrosse, Rochester, LaCrosse, Minneapolis, Sioux Falls**]. We remove **Rockford**, and add **Dubuque, Madison**, and **Chicago** to the front of the queue in that order, yielding **[Chicago, Madison, Dubuque, LaCrosse, Rochester, LaCrosse, Minneapolis, Sioux Falls]**. We test **Chicago**, and place **Rockford**, and **Milwaukee** on the queue. We take **Milwaukee** from the front and add **Chicago, Madison**, and **Green Bay** to the search queue. It is now **[Green Bay, Madison, Chicago, Rockford, Chicago, Madison, Dubuque, LaCrosse, Rochester, LaCrosse, Minneapolis, Sioux Falls]**. We remove **Green Bay** and add **Milwaukee, LaCrosse**, and **Wausau** to the queue in that order. Finally, **Wausau** is at the front of the queue and our goal test succeeds and our search ends. Our search order was Rochester, Dubuque, Rockford, Chicago, Milwaukee, Green Bay, and Wausau.

In this example, we again did not prevent tested nodes from being added to the search queue. As a result, we had duplicate nodes on the queue. In the depth-first case, this could have been disastrous. We could have easily had a cycle or loop where we tested one city, then a second, then the first again, ad infinitum. In the next section, we show our Java implementation of these search algorithms, starting with a class for the node and one for the search graph. We include tests to avoid this duplication of tests and nodes on the search queue.

The SearchNode Class

These problems are defined using a hierarchical tree or graph structure, comprised of a set of nodes and links. Our search algorithms work on a node structure which we defined as a Java class **SearchNode**. The **SearchNode** constructors and data members are shown below:

```
public class SearchNode extends Object {
  String label ;     // symbolic name
  Object state ;     // defines the state-space
  Object oper;       // operator used to generate this node
  Vector links;      // links to other nodes
  int depth ;        // depth in a tree from start node
  boolean expanded ; // indicates if node has been expanded
  boolean tested ;   // indicates if node was ever tested
  float cost=0 ;     // cost to get to this node

  static TextArea textArea1 ; // used for trace output only
  public static final int FRONT = 0 ;
  public static final int BACK = 1 ;
  public static final int INSERT = 2;

  SearchNode(String Label, Object State) {
    label = Label ; state = State ; depth = 0 ;
    links = new Vector() ; oper = null ;
    expanded = false ; tested = false ;
}
```

The first data member in our **SearchNode** class is the symbolic name or label of the node. Next is the state, which can be any **Object**. The *oper* is the definition of the operation which created the state of the object. This would be used in problems such as games where the state of a parent node would be expanded into multiple child nodes and states based on the application of the set of operators.

The *links* member is a **Vector** containing references to all other **SearchNode** objects to which this node is linked. **SearchNode** objects can be connected to form any graph including tree structures.

The *depth* member is an integer which defines the distance from the start or root node in any search. The two boolean flags, *expanded* and *tested*, are used by the search algorithms to avoid getting into infinite loops. The first, *expanded*, is set whenever the node is expanded during a search. The second, *tested*, is used whenever the state of the node is tested during a search. Depending on the search algorithm this flag may or may not be used. The *cost* member can be used to represent the current accumulated cost or any other cost measure associated with the search problem. The static member *textArea1* is used to display trace information in our example applet.

The constructor takes two parameters, the name or label, and the initial object state. The constructor initializes the *links* and *oper* members and sets the boolean flags to false.

Once we have created an instance of a **SearchNode**, we must specify the links to other nodes. The *addLink()* and *addLinks()* methods are provided for this purpose. Methods are provided for adding one, two, three, or four links to other **SearchNode** objects as well as a **Vector** of **SearchNode** objects when needed for very large graphs with high connectivity.

```
public void addLinks(SearchNode Node) {
              links.addElement(Node);
}

public void addLinks(SearchNode n1, SearchNode n2) {
              links.addElement(n1) ;
              links.addElement(n2) ;
}

public void addLinks(SearchNode n1, SearchNode n2,
                          SearchNode n3) {
              links.addElement(n1) ;
              links.addElement(n2) ;
              links.addElement(n3) ;
}

public void addLinks(SearchNode n1, SearchNode n2,
```

```
                        SearchNode n3, SearchNode n4) {
              links.addElement(n1) ; links.addElement(n2) ;
              links.addElement(n3) ; links.addElement(n4) ;
}

public void addLinks(Vector Nodes) {
             for (int i=0 ; i < Nodes.size() ; i++) {
                links.addElement(Nodes.elementAt(i)) ;
             }
}
```

We provide a set of methods for testing whether the node is a leaf node in a tree structure (i.e., it has no children) and methods to set the *depth, oper, expanded,* and *tested* data members.

```
public boolean leaf() { return (links.size() == 0) ; }
public void setDepth(int Depth) { depth = Depth; }
public void setOperator(Object Oper) { oper = Oper; }
public void setExpanded() { expanded = true; }
public void setExpanded(boolean state) { expanded = state; }
public void setTested(boolean state) { tested = state ; }
```

The *reset()* method is used to reset the node depth and the boolean flags before starting a search.

```
// initialize the node for another search
  public void reset() {
      depth = 0 ;
      expanded = false ;
      tested = false ;
  }
```

The *setDisplay()* method is used in our example applet to register a **TextArea** component for displaying trace information during a search.

```
static public void setDisplay(TextArea textArea) {
      textArea1 = textArea;
}
```

The *trace()* method writes out an indented string indicating the depth of the node in the search tree along with its label and state. As written, it assumes that the state is a string.

```
// write a trace statement -- indent to indicate depth
public void trace() {
     String indent = new String() ;
```

```
     for (int i=0 ; i < depth ; i++) indent += “  ” ;
     textArea1.appendText(indent + “Searching ” +
                          depth + “: ” + label +
                         “ with state = ” + state + “\n”) ;
}
```

The most complicated method in our **SearchNode** class is the *expand()* method, which is used by the various search algorithms to build up a search tree from the initial search graph. The parameters are a queue (a **Vector** instance) and a parameter specifying where the child nodes should be placed on the queue. The options are FRONT, BACK, or INSERT, based on the current cost in the **SearchNode**.

First we mark the node as expanded. Then we loop over all of the nodes to which the node has links. For each node, if it has not been *tested* (actually, this means placed on the queue, because the node states are tested only when they are removed from the front of the queue) we mark it as *tested*, set its *depth* in the search tree, and then place it on the *queue* in the specified position. For FRONT, we add at position 0. For BACK we simply *addElement()*, which places it at the end of the **Vector**.

For the INSERT case, we have a loop where we compare the cost of the node we are trying to insert, *nextNode*, to the cost of the nodes on the queue. If *nextCost* is lower, we place it on the queue. There are two cases where the *nextNode* will not be inserted in this loop: when the *queue* is empty, and when the *nextCost* is greater than the cost of all of the nodes already on the queue. We use a boolean flag, *inserted*, to handle these cases.

```
// expand the node and add to queue at specified position
// position 0=front, 1=back, 2=based on node cost
public void expand(Vector queue, int position) {
    setExpanded() ;
    for (int j = 0; j < links.size(); j++) {
       SearchNode nextNode = (SearchNode)links.elementAt(j) ;
       if (!nextNode.tested) {
         nextNode.setTested(true) ;
         nextNode.setDepth(depth+1) ;
         switch (position) {
          case FRONT: queue.insertElementAt(nextNode,0);
                  break ;
          case BACK: queue.addElement(nextNode);
                  break ;
          case INSERT:
           boolean inserted = false ;
```

```
            float nextCost = nextNode.cost ;
            for (int k=0 ; k < queue.size() ; k++) {
              // find where to insert this node
              if (nextCost < ((SearchNode)queue.elementAt(k)).cost){
                queue.insertElementAt(nextNode, k);
                inserted = true ;
                break ;     // exit the for loop
              }
            }
            // couldn't find place to insert, just add to end
            if (!inserted) queue.addElement(nextNode) ;
            break;
          }
        }
      }
}
```

Now that we have defined our **SearchNode** class, we can talk about our **SearchGraph** class, which contains the set of **SearchNode** objects which define our problem states. **SearchGraph** is a relatively simple class. It extends the **java.util.Hashtable** class by adding a name and a set of methods for handling operations on a collection of **SearchNode** objects. The constructor takes a single **String** parameter for the **SearchGraph** name. The default **Hashtable** behavior is extended by two methods. The *reset()* method uses the enumeration of the elements in the Hashtable to iterate over the **SearchNode** objects and reset each one in turn. The *put()* method takes a **SearchNode** object as a single argument, and then calls the Hashtable *put()* method using the *label* as the key and the **SearchNode** object as the value.

```
// SearchGraph is a container for a set of SearchNodes
// SearchNodes are stored in a Hashtable so they can be
//     retrieved by name

class SearchGraph extends Hashtable {
  String name ;

  SearchGraph(String Name) {
    name = Name ;
  }

  // reset each SearhNode in the graph
  // clear expanded and tested flags, set depth=0
  void reset() {
    Enumeration enum = this.elements() ;
    while (enum.hasMoreElements()) {
        SearchNode nextNode = (SearchNode)enum.nextElement();
```

```
      nextNode.reset() ;
  }
}

// add node to Hashtable, using node label as key
void put(SearchNode node) {
   put(node.label, node) ;
}
```

Previous | Table of Contents | Next

Search Applet

In this section, we develop a Java applet to illustrate the behavior of four search algorithms. The user interface of our Applet is comprised of a single dialog shown in Figure 2.2. A user can select from one of four search techniques: depth-first, breadth-first, iterated-deepening, and best-first. We have already presented the breadth-first and depth-first algorithms in some detail. We will discuss all four algorithms and their implementations in the following sections. The user can select the start and goal state in our test **SearchGraph** along with the type. Pressing the **Star**t button will invoke the corresponding search algorithm with the specified start node and goal states passed as arguments. As the search algorithms progress, trace information is displayed in the **TextArea** at the top of the dialog. Pressing the **Clear** button will clear this area between runs, if desired.

Figure 2.2 The search applet dialog.

A single static method *testGraph()* is defined in our **SearchApplet** which creates a sample **SearchGraph** for our examples of the different search algorithms. This graph defines the set of midwestern U.S. cities as shown in Figure 2.1. First we instantiate the **SearchGraph** object and then one **SearchNode** for each city. After each **SearchNode** object is created, it is added to the **SearchGraph** by using the *put()* method inherited from **Hashtable**. Note that the node name and the state (also the city's name) are identical. This is for illustration purposes only. There is no reason why the node *label* and the *state* have to be the same. In fact, the second parameter on the **SearchNode** constructor can be any Java **Object**. So we could have an array or matrix for tic-tac-toe, or some other arbitrarily complex state-space representation, depending on the problem we are trying to solve.

After the **SearchNode** objects are created, we then define the connectivity between nodes using the *addLinks()* method. Next, we set the *cost* of each node

as the distance from each city to Rochester, Minnesota. We use this in the best-first search example discussed later in this section. This cost value is set to a static value for this example. In most search problems, the cost would be computed as the search progresses. This is another simplification used to make the explanation more clear.

```
// build a test graph and then call the specified search routine
// this graph is a set of cites in the mid-west United States
public static SearchGraph testGraph() {

  SearchGraph graph = new SearchGraph(“test”) ;
  // first build the example tree
  SearchNode roch = new SearchNode(“Rochester”,“Rochester”);
  graph.put(roch) ;
  SearchNode sfalls = new SearchNode(“Sioux Falls”,“Sioux Falls”);
  graph.put(sfalls) ;
  SearchNode mpls = new SearchNode(“Minneapolis”,“Minneapolis”) ;
  graph.put(mpls) ;
  SearchNode lacrosse = new SearchNode(“LaCrosse”,“LaCrosse”) ;
  graph.put(lacrosse) ;
  SearchNode fargo = new SearchNode(“Fargo”,“Fargo”) ;
  graph.put(fargo) ;
  SearchNode stcloud = new SearchNode(“St.Cloud”,“St.Cloud”) ;
  graph.put(stcloud) ;
  SearchNode duluth = new SearchNode(“Duluth”,“Duluth”) ;
  graph.put(duluth) ;
  SearchNode wausau = new SearchNode(“Wausau”,“Wausau”) ;
  graph.put(wausau) ;
  SearchNode gforks = new SearchNode(“Grand Forks”,“Grand Forks”);
  graph.put(gforks) ;
  SearchNode bemidji = new SearchNode(“Bemidji”,“Bemidji”) ;
  graph.put(bemidji) ;
  SearchNode ifalls = new SearchNode(“International Falls”,“Internat
  graph.put(ifalls) ;
  SearchNode gbay = new SearchNode(“Green Bay”,“Green Bay”) ;
  graph.put(gbay) ;
  SearchNode madison = new SearchNode(“Madison”,“Madison”) ;
  graph.put(madison) ;
  SearchNode dubuque = new SearchNode(“Dubuque”,“Dubuque”) ;
  graph.put(dubuque) ;
  SearchNode rockford = new SearchNode(“Rockford”,“Rockford”) ;
  graph.put(rockford) ;
  SearchNode chicago = new SearchNode(“Chicago”,“Chicago”) ;
  graph.put(chicago) ;
  SearchNode milwaukee = new SearchNode(“Milwaukee”,“Milwaukee”) ;
  graph.put(milwaukee) ;

  roch.addLinks(mpls, lacrosse, sfalls, dubuque) ;
```

```
  mpls.addLinks(duluth, stcloud, wausau);
  mpls.addLinks(lacrosse, roch ) ;
  lacrosse.addLinks(madison, dubuque, roch);
  lacrosse.addLinks(mpls,gbay) ;
  sfalls.addLinks(fargo, roch) ;
  fargo.addLinks(sfalls, gforks, stcloud) ;
  gforks.addLinks(bemidji, fargo, ifalls) ;
  bemidji.addLinks(gforks, ifalls, stcloud, duluth) ;
  ifalls.addLinks(bemidji, duluth, gforks) ;
  duluth.addLinks(ifalls,mpls, bemidji) ;
  stcloud.addLinks(bemidji, mpls, fargo) ;
  dubuque.addLinks(lacrosse, rockford) ;
  rockford.addLinks(dubuque, madison, chicago) ;
  chicago.addLinks(rockford, milwaukee) ;
  milwaukee.addLinks(gbay, chicago) ;
  gbay.addLinks(wausau, milwaukee, lacrosse) ;
  wausau.addLinks(mpls, gbay) ;

  // use as costs for best first search example
  // straight line distances from cities to Rochester
     roch.cost = 0 ;      // goal
     sfalls.cost = 232 ;
     mpls.cost = 90 ;
     lacrosse.cost = 70 ;
     dubuque.cost = 140 ;
     madison.cost = 170 ;
     milwaukee.cost = 230 ;
     rockford.cost = 210 ;
     chicago.cost = 280 ;
     stcloud.cost = 140 ;
     duluth.cost = 180 ;
     bemidji.cost = 260 ;
     wausau.cost = 200 ;
     gbay.cost = 220;
     fargo.cost = 280;
     gforks.cost = 340;

   return graph ;
}
```

We construct our **SearchApplet** using Symantec's Visual Café development environment. The visual builder allows us to create a generic applet stub and then drag and drop the awt controls onto the pane. Visual Café automatically generates the Java code for creating the controls and adding them to the applet. This code is not presented here, because any Java development tool could have been used to construct this dialog. After the generated Java code, we insert our code to instantiate our test **SearchGraph**, fill the two **Choice** (listbox) controls

with the names of the cities on the map, set the depth-first radio button on, register the **TextArea** control for the trace information, select Rochester as our default start location, and wait for user input.

```
    // set up for search test applet
     graph = testGraph();              // build the test graph
     Enumeration enum = graph.keys() ; // get city names
     while(enum.hasMoreElements()) {
         String name = (String)enum.nextElement();
         choice1.addItem(name);   // fill start cities
         choice2.addItem(name);   // fill goal cities
     } // endwhile
     radioButton1.setState(true) ;      // default to depth-first
     SearchNode.setDisplay(textArea1) ; // used for trace display
     choice1.select("Rochester") ;
}
```

When the **Start** button is pressed, an **awt. Event** is generated and the following code is invoked. First we retrieve the *start* and *goal* **Strings** from the **Choice** controls. Next we clear all of the **SearchNode** objects in the graph by calling *reset()*. Then we check the state of the radio buttons using *getState()* and call the selected search algorithm. Only one radiobutton can be selected at any time.

```
void button1_Clicked(Event event) {
    int method = 0 ;
    SearchNode answer = null ;
   SearchNode startNode ;
    String start = choice1.getSelectedItem() ;
   startNode = (SearchNode)graph.get(start) ;
    String goal = choice2.getSelectedItem() ;
    graph.reset() ;         // reset all nodes for another search
    if (radioButton1.getState() == true) {
        textArea1.appendText("\n\nDepth-First Search for " +
                             goal + ":\n\n");
      answer = graph.depthFirstSearch(startNode,goal) ;
    } // endif
    if (radioButton2.getState() == true){
      textArea1.appendText("\n\nBreadth-First Search for " +
                           goal + ":\n\n");
      answer = graph.breadthFirstSearch(startNode, goal) ;
    } // endif
    if (radioButton3.getState() == true) {
      textArea1.appendText("\n\nIterated-Deepening Search for "
                            + goal + ":\n\n");
      answer = graph.iterDeepSearch(startNode, goal) ; //
    } // endif
    if (radioButton4.getState() == true) {
```

```
      textArea1.appendText("\n\nBest-First Search for " + goal
                           + ":\n\n");
      choice2.select("Rochester") ; // goal must be Rochester
      answer = graph.bestFirstSearch(startNode, "Rochester") ;
    } // endif

    if (answer == null) {
      textArea1.appendText("Could not find answer!\n");
    } else {
      textArea1.appendText("Answer found in node " +
                           answer.label);
    }
  }

void button2_Clicked(Event event) { // stop
    // for later use --- when we have large search problems
  }
```

Previous | Table of Contents | Next

When the Clear button is pressed, we clear out the **TextArea**.

```
void button3_Clicked(Event event) {  // clear
      // Clear the text for TextArea
      textArea1.setText("");
}
```

Breadth-First Search The Java implementation of the breadth-first search algorithm is shown below. The *breadthFirstSearch()* method is part of the **SearchGraph** class. The parameters are the initial SearchNode and the goal state, which in our **SearchApplet** is object label. First we instantiate our queue of nodes to search, and add the *initialNode* to the queue and mark it as tested.

In a *while()* loop, we first check to see if there are any more nodes to expand. If so, we remove the first node, call *trace()* to print out a message, then test the node to see if it matches our goal state. Although in our example applet we are using Strings to represent our goal state, this code should work for any state **Object** provided it implements the *equals()* method. If the goal test succeeds, we return with the node which matched the goal state. If the goal test fails and we haven't already expanded this node, we expand it by placing all of the nodes it has links to on the back of the queue. When the queue is empty we exit the *while()* loop.

```
// do a breadth-first search on graph
public SearchNode breadthFirstSearch(SearchNode initialNode,
                                     Object goalState)
{
    Vector queue = new Vector() ;
    queue.addElement(initialNode) ;
    initialNode.setTested(true) ;  // test each node once
    while (queue.size()> 0) {
      SearchNode testNode = (SearchNode)queue.firstElement() ;
      queue.removeElementAt(0) ;
      testNode.trace() ;
      if (testNode.state.equals(goalState)) return testNode;
      if (!testNode.expanded) {
         testNode.expand(queue,SearchNode.BACK) ;
      }
    }
    return null ;
}
```

Breadth-first search performs a complete search of the state space. If there are multiple solutions, it will find a solution with the shortest number of steps because each node of length **n** is examined before the next layer of nodes is explored. In Figure 2.3 we show the output of a breadth-first search run against our *testGraph* with Chicago as the start node and Rochester as the destination. Note that the exact search order is affected by how we define the links in our *testGraph*.

Figure 2.3 Breadth-first search algorithm example.

Depth-First Search The Java code for our depth-first search algorithm is similar to our breadth-first code, except the child nodes are added to the front of the queue instead of the back. This small change makes a big difference in the search behavior of the algorithm. Depth-first search requires less memory than breadth-first search but it can also spend lots of time exploring paths which lead to dead ends.

The *depthFirstSearch()* parameters are the starting **SearchNode** and the goal state. First we instantiate our *queue* of nodes to search, and add the *initialNode* to the queue and mark it as tested.

In a *while()* loop, we first check to see if there are any more nodes to expand. If so, we remove the first node, call *trace()* to print out a message, then test the node to see if it matches our goal state. Although in our example applet we are using **Strings** to represent our goal state, this code should work for any state **Object** provided it implements the *equals()* method. If the goal test succeeds, we return with the node which matched the goal state. If the goal test fails and we haven't already expanded this node, we expand it by placing all of the nodes it has links to on the front of the *queue*. When the *queue* is empty we exit the *while()* loop.

```
// do a depth first search on graph
public SearchNode depthFirstSearch(SearchNode initialNode,
                                   Object goalState)
  {
    Vector queue = new Vector() ;
```

```
    queue.addElement(initialNode) ;
    initialNode.setTested(true) ;  // test each node once

    while (queue.size()> 0) {
      SearchNode testNode = (SearchNode)queue.firstElement() ;
      queue.removeElementAt(0) ;
      testNode.trace() ;           // display trace information
      if (testNode.state.equals(goalState)) return testNode;

      if (!testNode.expanded) {
        testNode.expand(queue,SearchNode.FRONT);
      }
    }
    return null ;
}
```

Figure 2.4 Depth-first search algorithm example.

Figure 2.4 shows depth-first search results using our search applet with Chicago as the start node and Rochester as the destination.

Improving Depth-First Search

One easy way to get the best characteristics of the depth-first search algorithm along with the advantages of the breadth-first search is to use a technique called iterative-deepening search. In this approach, we first add a slight modification to the depth-first search algorithm, by adding a parameter called *maxDepth*, to limit our search to a maximum depth of the tree. Then we add a control loop where we continually deepen our depth-first search until we find the solution.

This algorithm, like standard breadth-first search, is a complete search and will find an optimal solution, but it has much lower memory requirements, like the depth-first algorithm. Although we are retracing ground when we increase our depth of search, this approach is still more efficient than pure breadth-first or pure unlimited depth-first search for large search spaces (Russell and Norvig 1995).

Our Java implementation uses two methods. The *iterDeepSearch()* method

performs the outer loop, repeatedly calling *depthLimitedSearch()* with the *maxDepth* increasing by one for each successive call. The *depthLimitedSearch()* method is essentially the same as our standard depth-first search algorithm, with the test against *maxDepth* to short-circuit expansion of **SearchNode** once they get too deep.

```
// this is a slightly modified depth-first search algorithm
// that stops searching at a pre-defined depth
public SearchNode depthLimitedSearch(SearchNode initialNode,
                                     Object goalState,
                                     int maxDepth)
{
    Vector queue = new Vector() ;
    queue.addElement(initialNode) ;
    initialNode.setTested(true) ;  // only test each node once

    while (queue.size()> 0) {
      SearchNode testNode = (SearchNode)queue.firstElement() ;
      queue.removeElementAt(0) ;
      testNode.trace() ;
      if (testNode.state.equals(goalState)) return testNode ;

      // limit the depth of search to maxDepth
      if (testNode.depth < maxDepth) {
        if (!testNode.expanded) {
            testNode.expand(queue,SearchNode.FRONT) ;
        }
      }
    }
    return null ;
}

// use depth-first search to find goal
public SearchNode iterDeepSearch(SearchNode startNode,
                                 Object goalState) {
   int maxDepth = 10 ;                 // arbitrary limit
   for (int j=0 ; j < maxDepth ; j++) {
     reset() ;
     SearchNode answer = depthLimitedSearch(startNode,goalState,j);
     if (answer != null) return answer;
   }
   return null ; // failed to find solution in maxDepth
}
```

In Figure 2.5 we show a trace of a search from Chicago to Rochester using the iterated-deepening algorithm. Notice how the algorithm recovers ground as it goes to each level.

Heuristic Search

The Traveling Salesman Problem (TSP), where a salesman makes a complete tour of the cities on his route, visiting each city exactly once, while traveling the shortest possible distance, is an example of a problem which has a combinatorial explosion. As such, it cannot be solved using breadth-first or depth-first search for problems of any realistic size. TSP belongs to a class of problems known as NP-hard or NP-complete. Unfortunately, there are many problems which have this form and which are essentially intractable (they can't be solved). In these cases, finding the best possible answer is not computationally feasible, and so we have to settle for a good answer. In this section we discuss several heuristic search methods which attempt to provide a practical means for approaching these kinds of search problems.

Figure 2.5 Iterated-deepening search algorithm example.

Heuristic search methods are characterized by this sense that we have limited time and space in which to find an answer to complex problems and so we are willing to accept a good solution. As such, we apply heuristics or rules of thumb as we are searching the tree to try to determine the likelihood that following one path or another is more likely to lead to a solution. Note this is in stark contrast to brute-force methods which chug along merrily regardless of whether a solution is anywhere in sight.

Heuristic search methods use objective functions called (surprise!) heuristic functions to try to gauge the value of a particular node in the search tree and to estimate the value of following down any of the paths from the node. In the next sections we describe four types of heuristic search algorithms.

Generate and Test The generate and test algorithm is the most basic heuristic search function. The steps are:

1. Generate a possible solution, either a new state or a path through the problem space.
2. Test to see if the new state or path is a solution by comparing it to a set of goal states.
3. If a solution has been found, return success; else return to step 1.

This is a depth-first search procedure which performs an exhaustive search of the state space. If a solution is possible, the generate and test algorithm will find it. However, it may take an extremely long time. For small problems, generate and test can be an effective algorithm, but for large problems, the undirected search strategy leads to lengthy run times and is impractical. The major weakness of generate and test is that we get no feedback on which direction to search. We can greatly improve this algorithm by providing feedback through the use of heuristic functions.

Hill climbing is an improved generate-and-test algorithm, where feedback from the tests are used to help direct the generation (and evaluation) of new candidate states. When a node state is evaluated by the goal test function, a measure or estimate of the distance to the goal state is also computed. One problem with hill climbing search in general is that the algorithm can get caught in local minima or maxima. Because we are always going in the direction of least cost, we can follow a path up to a locally good solution, while missing the globally excellent solution available just a few nodes away. Once at the top of the locally best solution, moving to any other node would lead to a node with lower goodness. Another possibility is that a plateau or flat spot exists in the problem space. Once the search algorithm gets up to this area all moves would have the same goodness and so progress would be halted.

To avoid getting trapped in suboptimal states, variations on the hill climbing strategy have been proposed. One is to inject noise into the evaluation function, with the initial noise level high and slowly decreasing over time. This technique, called simulated annealing, allows the search algorithm to go in directions which are not “best” but allow more complete exploration of the search space. Simulated annealing is analogous to annealing of metals, whereby they are heated and then gradually cooled. Thus a temperature parameter is used in simulated annealing, where a high temperature allows more searching, but as the temperature cools, the search algorithm reverts to the more standard hill climbing behavior (Kirkpatrick, Gelatt, and Vecchi 1983).

In the next section we describe best-first search, which is a more formal specification of a greedy, hill climbing type search algorithm.

Best-First Search Best-first search is a systematic control strategy, combining the strengths of breadth-first and depth-first search into one algorithm. The main difference between best-first search and the brute-force search techniques is that we make use of an evaluation or heuristic function to order the **SearchNode** objects on the queue. In this way, we choose the SearchNode that appears to be best, before any others, regardless of their position in the tree or graph. Our implementation of Best-first search uses the current value of the cost data member to order the **SearchNode** objects on the search queue. In our *testGraph()* method above, we set the cost of the nodes to be equal to the approximate straight-line distance from each city to Rochester, Minnesota.

The *bestFirstSearch()* parameters are the initial **SearchNode** and the goal state. To start, we instantiate our *queue* of nodes to search, and add the **initialNode** to the queue and mark it as tested.

In a *while()* loop, we first check to see if there are any more nodes to expand. If so, we remove the first node, call *trace()* to print out a message, then test the node to see if it matches our goal state. Although in our example applet we are using **Strings** to represent our goal state, this code should work for any state **Object** provided it implements the *equals()* method. If the goal test succeeds, we return with the node which matched the goal state. If the goal test fails and we haven't already expanded this node, we expand it by placing all of the nodes it has links to on the front of the *queue*. When the *queue* is empty we exit the *while()* loop.

```
// use best-first search algorithm to find the goal
// default implementation based on SearchNode cost
public SearchNode bestFirstSearch(SearchNode initialNode,
                                  Object goalState)
{
    Vector queue = new Vector() ;
    queue.addElement(initialNode) ;
    initialNode.setTested(true) ;  // only test each node once

    while (queue.size()> 0) {
      SearchNode testNode = (SearchNode)queue.firstElement() ;
      queue.removeElementAt(0) ;
      testNode.trace() ;
      if (testNode.state.equals(goalState)) return testNode ;
```

```
      // now, heuristically add nodes to queue
      // insert the child nodes according to cost
       if (!testNode.expanded) {
            testNode.expand(queue,SearchNode.INSERT) ;
       }
     }
     return null ;
}
```

Figure 2.6 shows an example of the best-first search algorithm applied to the problem of finding the best route from Chicago to Rochester, MN.

Greedy Search Greedy search is a best-first strategy where we try to minimize the estimated cost to reach the goal (certainly an intuitive approach!). Since we are greedy, we always expand the node that is estimated to be closest to the goal state. Unfortunately, the exact cost of reaching the goal state usually can't be computed, but we can estimate it by using a cost estimate or heuristic function *h()*. When we are examining node **n**, then *h*(**n**) gives us the estimated cost of the cheapest path from n's state to the goal state. Of course, the better an estimate *h()* gives, the better and faster we will find a solution to our problem. Greedy search has similar behavior to depth-first search. Its advantages are delivered via the use of a quality heuristic function to direct the search.

A* Search One of the most famous search algorithms used in AI is the A* search algorithm, which combines the greedy search algorithm for efficiency with the uniform-cost search for optimality and completeness. In A* the evaluation function is computed by adding the two heuristic measures; the *h*(**n**) cost estimate of traversing from **n** to the goal state, and *g*(**n**) which is the known path cost from the start node to **n** into a function called *f*(**n**).

Figure 2.6 Best-first search algorithm example.

Again, there is nothing really new here from a search algorithm point of view; all we are doing is using better information (heuristics) to evaluate and order the nodes on our search queue. We know how much it cost to get where we are (node **n**) and we can guesstimate how much it will cost to reach the goal from **n**. Thus we are bringing all of the information about the problem we can to bear on directing our search. This combination of strategies turns out to provide A* with both completeness and optimality.

Constraint Satisfaction Another approach to problem solving using search is called constraint satisfaction. All problems have some constraints which define what the acceptable solutions are. For example, if our problem is to load a delivery truck with packages, a constraint may be that the truck holds only 2000 pounds. This constraint could help us substantially reduce our search space by ignoring search paths which contain a set of items which weigh more than 2000 pounds. Constraint satisfaction search uses a set of constraints to define the space of acceptable solutions. Rather than a simple goal test, the search is complete when we have a set of bindings of values to variables, a state where the minimum set of constraints holds.

The role of constraints is to bind variables to values or limit the range of values which they may take on, thus reducing the number of combinations we need to explore. First, an initial set of constraints are applied and propagated through the problem, subject to the dependencies which exist. For example, if we set a limit of 2000 pounds for a particular delivery truck, that may immediately remove some items from consideration. Also, once we assign an object to the truck which weighs 800 pounds, we can infer that now the limit for additional items is 1200 pounds. If a solution is not found by propagating the constraints, then search is required. This may involve backtracking or undoing an assignment or variable binding.

Means-Ends Analysis Means-ends analysis is a process for problem solving which is based on detecting differences between states and then trying to reduce those differences. First used in the General Problem Solver (Newell and Simon 1963), means-ends analysis uses both forward and backward reasoning and a recursive algorithm to systematically minimize the differences between the initial and goal states.

Like any search algorithm, means-ends analysis has a set of states along with a set of operators which transform that state. However, the operators or rules for transformation do not completely specify the before and after states. The left side or antecedent of the rule only contains the subset of conditions, the *preconditions*, which must be true in order for the operator to be applied. Likewise the right-hand side of the rule identifies only those parts of the state which the operator changes. So each operator has a before and after list of states and state changes which are only a subset of the whole problem space. A simplified version of the means-ends analysis is described here (Rich and Knight 1991).

Means-Ends Analysis (Current-State, Goal-State)

1. Compare the current-state to the goal-state. If states are identical then return success.
2. Select the most important difference and reduce it by performing the following steps until success or failure :

a. Select an operator that is applicable to the current difference. If there are no operators which can be applied, then return failure.
b. Attempt to apply the operator to the current state by generating two temporary states, one where the operator's preconditions are true (prestate), and one that would be the result if the operator were applied to the current state (poststate).
c. Divide the problem into two parts, a FIRST part, from the current-state to the pre-state, and a LAST part, from the poststate to the goal state. Call means-ends analysis to solve both pieces. If both are true, then return success, with the solution consisting of the FIRST part, the selected operator, and the LAST part.

Means-ends analysis is a powerful heuristic search algorithm. It has been applied in many artificial intelligence applications, especially in planning systems.

Summary

In this chapter we presented the major search algorithms used in artificial intelligence applications. The major points include:

- The *state space* approach to problem representation requires a mapping

from the real-world problem to a data structure (the state), an *initial state*, a set of *operators* for changing the state, and defining a *goal test* to determine when we have reached a goal state. A *cost* may also be associated with a generated state.

- Effective search algorithms must cause systematic motion through the state space. *Brute-force search* algorithms blindly search the state space, while *heuristic search* algorithms use feedback or information about the problem to direct the search.
- A search algorithm is *optimal* if it is guaranteed to find the best solution from a set of possible solutions. An algorithm is *complete* if it will always find a solution if one exists. The *time complexity* defines how fast an algorithm performs and scales. The *space complexity* describes how much memory the algorithm requires to perform the search.
- *Breadth-first search* is a complete algorithm with exponential time and space complexity. It examines each state one step away from the initial state, then each one two steps away, and so on.
- *Depth-first search* has lower memory requirements than breadth-first search, but it is neither complete (it can get stuck in loops) nor optimal. In depth-first search, a single path is chosen and followed until a solution is found, or a dead end (a leaf node) is reached. The algorithm then backs up and continues down another path.
- *Iterated-deepening search* uses a modified version of depth-first search to limit the depth of search. It does this with a control loop which increases the depth on each iteration. The approach combines the best of breadth-first and best-first search. It is complete and optimal, and it has much lower memory requirements than unlimited depth-first search.
- *Heuristic search* algorithms use information about the problem to help direct the path through the search space. This information is used to select which nodes to expand. *Best-first* search always expands the node that looks to be closest to the solution. *A* search* uses the known cost combined with an estimate of the distance from the state to the goal to choose a node to expand. A* is complete and optimal, and has lower memory requirements similar to depth-first search.
- *Constraint satisfaction* search uses information about valid states to help limit or constrain the range of the search. *Means-ends analysis* solves problems by detecting differences between states and then trying to reduce those differences.

Exercises

1. Open your Java-enabled Web browser or applet viewer on the *SearchApplet.html* file. While referring to Figure 2.1, select two cities and run the depth-first, breadth-first, and iterated-deepening search algorithms. Which algorithm finds the solution in the least time? Why? What if you selected two other cities?

2. In this chapter we developed a version of the best-first search algorithm, where the estimated cost from the current node to the goal state was the straight-line distance to the goal. The A* search algorithm combines this estimated cost with the current known cost from the start node to the current node. Extend the best-first implementation to perform A* search. How does it compare to best-first?

3. What is the major drawback to the search-based problem solving approach? How do heuristics help overcome this weakness? How does constraint satisfaction search overcome this weakness?

Previous | Table of Contents | Next

Chapter 3
Knowledge Representation

In this chapter we explore some of the techniques used to represent domain knowledge in artificial intelligence programs. We start with a discussion of several kinds of knowledge and the different demands that people and computers place on knowledge representations. We describe procedural and declarative knowledge representations and follow with an introduction to propositional and predicate logic. Next we discuss frames and semantic nets, two related artificial intelligence techniques for representing concepts and their relationships. The Knowledge Interchange Format is presented as an emerging industry standard for knowledge representation. Finally, we discuss the knowledge acquisition process, where expert domain knowledge is turned into a knowledge base for problem solving.

From Knowledge to Knowledge Representation

What is knowledge? Is knowledge the same thing as facts? *Webster's Dictionary* (Merriam-Webster 1988) defines knowledge as "the fact or condition of knowing something with familiarity gained through experience or association." People gain knowledge through experience—they see, hear, touch, feel, and taste the world around them. We can associate something we see with something we hear, thereby gaining new knowledge about the world. An alternate definition for knowledge is "the fact or condition of being aware of something." How do we make a computer aware of something? Suppose we know that the sun is hot, balls are round, and the sky is blue. These facts are knowledge about the world. How *do* we store this knowledge in our brain? How *could* we store this knowledge in a computer? This problem, called knowledge representation, is one of the first, most fundamental issues that researchers in artificial intelligence had to face. And the answer they found was *symbols*.

While psychologists and neuroscientists are still searching for the answer to how people store knowledge in their brains (we know it has something to do with the synapses), in the field of artificial intelligence, programmers use symbols to

represent and manipulate knowledge in computers. What is a symbol? A symbol is a number or character string that represents an object or idea. Strings and numbers are used because computers are very good at processing them. This is called the internal representation of the knowledge. However, people are most comfortable using a natural language like English to represent knowledge. Thus, for practical reasons, we need mappings from facts to an internal computer representation and also to a form that people can understand.

Though natural language is perhaps the most easily understood knowledge representation for people, it is certainly not the best for computers, because natural language is inherently ambiguous. That is, two people can read the same statement and disagree as to what it means. This ambiguity is exactly why the languages of formal mathematics and logic were developed during the past two millennia. It should be no surprise that artificial intelligence was first applied to formal languages and mathematical theorem proving, and that one of the first knowledge representations was formal logic. For those mathematicians who were the first computer programmers, logic was a “natural” language to use.

There are many different kinds of knowledge we may want to represent: simple facts or complex relationships, mathematical formulas or rules for natural language syntax, associations between related concepts, inheritance hierarchies between classes of objects. As we will show, each type of knowledge places special requirements on both human comprehension and computer manipulation. Knowledge representation is not a one-size-fits-all proposition. Consequently, choosing a knowledge representation for any particular application involves tradeoffs between the needs of people and computers. In addition to being easy to use, a good knowledge representation also must be easily modified and extended, either by changing the knowledge manually or through automatic machine learning techniques. Let’s look at some common kinds of knowledge and the most popular approaches for storing that knowledge in computers.

Procedural Representation

Perhaps the most common technique for representing knowledge in computers is *procedural* knowledge. Procedural code not only encodes facts (constants or bound variables) but also defines the sequence of operations for using and manipulating those facts. Thus, program code is a perfectly natural way of encoding procedural knowledge. Whether data structures or objects are used to model the problem, the program is essentially one big knowledge representation.

Programs written in scripting languages such as Visual Basic, JavaScript, and LotusScript are examples of a procedural knowledge representation. The knowledge of how to process data is encoded in the control structures and sequence of the program statements. This “hardcoded” logic is typically not considered to be part of AI per se, but few real AI programs exist which do not contain some amount of procedural control code.

In procedural code, the knowledge and the manipulation of that knowledge are inextricably linked. This weakness is overcome by the most popular knowledge representation approach, called *declarative*. In declarative knowledge representation, a user simply states facts, rules, and relationships. These facts, rules, and relationships stand by themselves and represent pure knowledge. Most of the knowledge representation techniques studied in artificial intelligence and discussed in the remainder of this chapter are declarative. However, declarative knowledge needs to be processed by some procedural code, so we never get too far from the need for explicit sequential instructions for the computer to follow. Still, the separation of knowledge from the algorithm used to manipulate or reason with that knowledge provides advantages over procedural code. Because the knowledge is explicitly represented, it can be more easily modified. Also, separating the control logic and reasoning algorithms from the knowledge allows us to write optimized and reusable inferencing procedures.

Relational Representation

Another way to represent information is in relational form, such as that used in relational database systems. Relational databases provide a powerful and flexible mechanism for storing knowledge, which is why they have almost completely taken over the business of storing information in commercial business systems. Knowledge is represented by tuples or records of information about an item, with each tuple containing a set of fields or columns defining specific attributes and values of that item. By storing a collection of information in a table, we can use relational calculus to manipulate the data, based on the relations defined, and query the information stored in the table. Structured Query Language (SQL) is the most popular language for manipulating relational data.

While relational database tables are flexible, they are not good at representing complex relationships between concepts or objects in the real world. This is where network and hierarchical database systems, such as IBM's IMS, are strong. Having links or pointers between related groups of data allows both hierarchical and complex network graphs to be built. The AI techniques of semantic nets and frames, discussed in more detail later in this chapter, use a similar approach for representing knowledge.

Hierarchical Representation

Another type of knowledge is inheritable knowledge, which centers on relationships and shared attributes between kinds or classes of objects. Hierarchical knowledge is best used to represent "isa" relationships, where a general or abstract type (for example, ball) is linked to more specific types (rubber, golf, baseball, football) which inherit the basic properties of the general type. The strength of object inheritance allows for compact representation of knowledge and allows reasoning algorithms to process at different levels of abstraction or granularity. We could reason about sports and the common attributes of balls at one level, or we could delve into the details of a particular sport and its respective type of ball. The use of categories or types gives structure to the world by grouping similar objects together. Using categories or clusters simplifies reasoning by limiting the number of distinct things we have to deal with. A taxonomy or hierarchy of objects or concepts is a useful way to organize collections of categories, because it allows us to reduce complexity and think at higher levels of abstraction where possible.

Using objects to model the world and to represent knowledge is becoming increasingly popular. In addition to relational and network databases, object databases that store object data and methods are now being deployed. Besides mapping naturally onto the real world, objects can also be used to model abstract ideas and their relationships. Object-oriented programming languages such as Smalltalk, C++, and Java provide a natural framework for representing knowledge as objects, and for reasoning about and manipulating those objects. Inheritance and class hierarchies are fundamental concepts on which object programs are built, and basically come "for free" with object-oriented knowledge representations.

Capturing knowledge about objects in the real world and nonphysical measurements such as time are often required in AI problem solving. Knowing what to expect based on the elapsed time from one event to another is often the hallmark of intelligent behavior. Knowing that a friend just threw a snowball at your head would be useful knowledge so that you could brace for possible impact. Knowing which friend threw the snowball could help you determine the likelihood of getting hit at all. But this knowledge would only apply for a short time (2–5 seconds after the throwing event). Time concepts such as *before*, *after*, and *during* are crucial to common-sense reasoning and planning. When we are trying to solve a problem, we often pretend that "time stands still" while we are doing our computation. However, in many problems we must explicitly deal with changes, due to the passing of time, or movement of objects in the world. Special forms of logic, called temporal logic, have been developed to deal with representing and reasoning about time.

Although there are as many different ways to represent knowledge as there are types of knowledge, only a handful of knowledge representations are widely used in artificial intelligent applications. In the remainder of this chapter, we explore formal logic, frames, and semantic nets, while in the next chapter we explore rule-based knowledge representations. We start our discussion with predicate logic.

Predicate Logic

The use of formal logic as a primary knowledge representation harkens back to the beginnings of artificial intelligence research. Mathematical deduction, based on logic, was a well-known method of generating new knowledge from existing knowledge. Early AI researchers, therefore, used what they knew. While a detailed discussion of formal logic is outside the scope of this book, we will give a brief introduction to this topic.

Formal logic is a language with its own *syntax*, which defines how to make sentences, and corresponding *semantics*, which describe the meaning of the sentences. The most basic form of logic representation is called boolean or propositional logic, where each proposition or fact is represented by a symbol that evaluates to either **true** or **false**. Sentences can be constructed using proposition symbols (**P**, **Q**, **R**, ...) and boolean connectives, such as conjunction (**and**), disjunction (**or**), implication (**P implies Q**), and equivalence (**A is equivalent to B**). Using this simple syntax, we can write implications or rules,

such as **(P and Q) implies R** or, more programmatically **if P and Q then R**. In the preceding rule, **P and Q** is called the premise or *antecedent*, and **R** is the conclusion or consequent. In addition to joining propositions with connectives, a proposition can be negated (**not**) so that if **P** is **true**, then **not P** is **false**. Common rules of inference can be used to reason and infer facts from propositional logic sentences. One of the most familiar is called *Modus Ponens*, where given a rule, **A implies B**, and knowledge that the antecedent sentence **A** is true, we can infer that the sentence **B** is also true.

Boolean logic is used as the basis for designing digital computers and is quite suited and powerful for circuit design. However, it quickly runs out of steam as a knowledge representation language. So, *predicate logic*, which allows *predicates* on objects to define attributes and relations between objects, has become the preferred logic for knowledge representation in artificial intelligence systems. Using objects, attributes, and relations, we can represent almost any type of knowledge. In addition, predicate logic introduces the concept of *quantifiers*, which allow us to refer to sets of objects. The two quantifiers are *existential* (there exists some object that has the specified attribute) and *universal* (all objects of this type have this attribute). A statement such as "**Minnesota is cold in the winter**." could be represented in predicate logic in several ways. We could use a conjunction of functions (say that fast three times), where functions are relations with a single parameter, as in **place(Minnesota) and temperature(cold) and season(winter)**. Or we could use a single relation, such as **cold(Minnesota, winter)**. Or we could even say **winter(Minnesota, cold)** to represent the same statement.

Previous | Table of Contents | Next

From these examples, you can see that predicate logic does not give any guidance as to what predicates we should use. It also does not explicitly say how time or events should be represented. That is not to say that we can't represent these types of knowledge in predicate logic. It can be done. The main point is to understand that even with the syntax and semantics defined, we still have a lot of decisions to make in how the knowledge is represented by predicates.

But having a good knowledge representation solves only half of the problem, because we also need to manipulate the knowledge to generate new facts and prove or refute assertions about the knowledge. Unfortunately, predicate logic does not provide a sure-fire way to derive new information. However, it still can be used to process knowledge in a useful way. Two techniques, called *resolution* and *unification* are used to process predicate statements to prove whether a particular statement is true or not, based on the other known facts. Together these algorithms form the basis for Prolog (**Pro**gramming in **log**ic). Prolog and Lisp are the two programming languages traditionally used for artificial intelligence applications (Clocksin and Mellish 1981). While we will not go into details here, a basic understanding of the capabilities and limitations of predicate logic as a basis for reasoning is necessary. Our discussion will focus on two basic mechanisms, resolution and unification.

Resolution

Resolution is an algorithm for proving facts true or false by virtue of contradiction (Robinson 1965). If we want to prove a theorem **X** is true, we have to show that the negation of **X** is not true. For example, suppose that we know the following two facts:

1. **not** feathers(Tweety) **or** bird(Tweety)
2. feathers(Tweety)

Sentence 1 states that either Tweety does not have feathers or else Tweety is a bird. Sentence 2 states that Tweety has feathers. To prove that Tweety is a bird, we first add an assumption that is the negation of that predicate, giving sentence 3:

1. **not** feathers(Tweety) **or** bird(Tweety)

2. feathers(Tweety)
3. **not** bird(Tweety)

In sentences 1 and 2, **not feathers(Tweety)** and **feathers(Tweety)** cancel each other out. Resolving sentences 1 and 2 produces the resolvant, sentence 4, which is added to our fact set:

1. **not** feathers(Tweety**) or** bird(Tweety)
2. feathers(Tweety)
3. **not** bird(Tweety)
4. bird(Tweety)

It is clear that sentences 3 and 4 cannot both be true, either Tweety is a bird or it is not. Thus, we have a contradiction. We have just proved that our first assumption, **not bird(Tweety)**, is false, and the alternative, **bird(Tweety)**, must be true (Winston 1993). If the clauses to be resolved are selected in systematic ways, then resolution is guaranteed to find a contradiction if one exists, although it may take a long time to find.

Unification

Unification is a technique for taking two sentences in predicate logic and finding a substitution that makes them look the same. This is a requirement for proving theorems using resolution, as discussed previously. If two predicates are identical, then they match, by definition. If one or both contains variables, then appropriate substitutions must be found using unification as follows:

- A variable can be replaced by a constant.
- A variable can be replaced by another variable.
- A variable can be replaced with a predicate, as long as the predicate does not contain that variable.

Given the following set of predicates, let's explore how they can be unified:

1. hates(*X* , *Y*)
2. hates(George, broccoli)
3. hates(Alex, spinach)

We could unify sentence 2 with 1 by binding George to variable *X*, and broccoli

to variable *Y*. Similarly, we could bind Alex to *X* and spinach to *Y*. Note that if the predicate names were different, we could not unify these predicates. If we introduce a few more predicates, we can explore more complex unifications:

4. hates(*X*, vegetable(*Y*))
5. hates(George, vegetable(broccoli))
6. hates(*Z*, broccoli)

We could unify sentence 6 with sentence 1 by replacing variable *X* with variable *Z* and variable *Y* with the constant broccoli. Sentences 4 and 5 could be unified with George bound to *X*, and broccoli to variable *Y*.

A generalized version of the unification algorithm, called *match*, is used in Prolog. *Facts* are represented in Prolog by clauses, which look like standard predicates, and declare things which are unconditionally true. *Rules* are clauses where the conclusion may be true, provided that all of the clauses in the condition part are true. Prolog provides a built-in inferencing procedure, based on resolution, for processing rules and answering questions posed as goal clauses (Bratko 1986).

While predicates can be used to represent and reason with rules, all rule systems do not use predicate logic as their knowledge representation language. Early rule-based systems were developed using Prolog and Lisp, but most commercial implementations are now written in C and C++. In Chapter 4, we will describe a rule-based inferencing system developed using Java.

Frames

A frame is a collection of attributes which defines the state of an object and its relationship to other frames (objects). But a frame is much more than just a record or data structure containing data. In AI, frames are called slot-and-filler data representations. The slots are the data values, and the fillers are attached procedures which are called *before*, *during* (to compute the value of), or *after* the slot's value is changed. Frames are often linked into a hierarchy to represent has-part and isa relationships.

If a sequence of actions is applied to a frame, then some of the attributes change while most remain the same. When sequences of operations are required, as in a search problem, a problem known as the *frame problem* arises. The problem is that if we copy the complete frame (state of the object) for each step in the sequence, then we may quickly use up the computer memory, as we duplicate the same unchanged knowledge over and over. The frame problem deals with the problem that when an action occurs, it is not always obvious which attributes in the frame should change. The solution is to only specify the parts of the state which must match for a condition to be true, and to only change those slots or attributes that change as a result of an operator.

Figure 3.1 shows an example of a frame representation for a subset of the domain of vehicles. Notice that each frame has a set of slots, and the frames are linked together to show their relationships. For example, the automobile frame has three slots. It is a subset of vehicles as indicated by the isa link. Our definition says that an automobile must have doors, a motor, and four wheels. A sports car is a subset or type of automobile. It has two doors and is small. Notice that sports car inherits several attributes from automobile. Finally, Corvette is an instance of a sports car, and each instance has a unique license number.

Figure 3.1 A frame example.

To anyone familiar with object-oriented programming, a frame sounds very much like an object, whose data members are the slots, and whose methods are the attached procedures or daemons. In some sense, any Java program is a frame-based mechanism for knowledge representation. Object-oriented programs also make use of inheritance for isa relationships, and containment or references for has-part relationships. In fact, an instance of a Java object with data members to hold the state and methods to test and change the object state provides a solution to the frame problem described above.

Semantic Nets

Semantic nets are used to define the meaning of a concept by its relationships to other concepts. A graph data structure is used, with nodes used to hold concepts, and links with natural language labels used to show the relationships. A portion of a semantic net representation of the vehicle domain is shown in Figure 3.2.

Again, the standard relationships such as isa, has-part, and instance should be familiar to readers with object-oriented design experience. Much of the object modeling work was anticipated by the semantic net research done in the 1960s (Quillian 1968). Once the semantic net is constructed, a technique called *spreading activation* is used to see how two concepts or nodes are related.

A modern implementation of a semantic net is the Knowledge Utility (KnU) developed by IBM. KnU is the technology behind the Aqui' Web site (http://www.ibm.aqui.com). More than just a semantic net, KnU builds an attributed net, where each user has his or her own preferences mapped onto the underlying semantic net. This allows personalized views of the content and concepts encoded in the semantic net. For example, one person may be at the Java concept node and have strong links to gourmet coffee providers, while another may have strong links to Java development tool providers.

While not obvious at first glance, both frames and semantic nets are very closely related to predicate logic. Russell and Norvig (1995) provide an algorithm for transforming both representations into first-order logic. The major difference between these knowledge representations is their syntax. Some people prefer formal logic, while others can relate more easily to graphical representations.

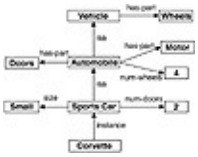


Figure 3.2 A semantic net example.

Representing Uncertainty

In almost any real-world application, a reasoning system will not have all of the relevant information it needs to solve a problem *a priori*. We have uncertainty. In most cases, there will be some information available, but the rest will have to be

inferred or collected as the inferencing proceeds. Fortunately, we have a statistical theory which works well under conditions of uncertainty, called Bayes' rule or Bayes' theorem.

The probability of something can range from a probability of 0.0 (no chance) to a probability of 1.0 (certainty). Statisticians differentiate between two kinds of probabilities. First, there are unconditional (or prior) probabilities which represent the chance that something will happen. For example, we can look in a weather almanac and see that, on average, it has rained 10 days in March in Minnesota over the last hundred years. So the probability that it will rain on any given day in March is roughly 33 percent. This is the prior or unconditional probability. However, suppose we know that a big storm is blowing in from South Dakota, and it will reach Minnesota tomorrow. Given that knowledge, we may say there is an 80 percent chance of rain. Did the long-term probability change? No; but we have evidence that tomorrow could be a rainy day, so we make use of that evidence to update our forecast. Statisticians call this type of probability estimate a conditional probability, expressed as P(*H* | E), that is read as the probability of hypothesis *H* given that we have observed evidence *E*.

Bayes' theorem says that we can compute the conditional probability that event *Y* will occur given that event *X* already occurred, providing we know the prior probabilities that *X* and *Y* could happen, and the conditional probability that *X* will occur when we know that *Y* has already occurred.

$$P(Y \mid X) = P(X \mid Y) P(Y) / P(X)$$

A Bayesian network (also called a belief network, causal network, or probabilistic network) is a data structure (a directed acyclic graph) which is used to represent dependence between variables. Each variable has a corresponding node with a conditional probability table defining the relationships between its parent nodes. The primary use of Bayesian networks is to use probability theory to reason with uncertainty.

Knowledge Interchange Format

So far, we have discussed predicate logic, frames, and semantic nets, and have shown how they can represent knowledge. Because proponents of the various techniques could not agree on one knowledge representation format for their AI applications, a clear need was identified for a common language to bridge the gaps.

The Knowledge Interchange Format (KIF) is a language that was expressly designed for the interchange of knowledge between agents (Gensereth and Fikes 1992). Based on predicate logic, KIF is a flexible knowledge representation language that supports the definition of objects, functions, relations, rules, and metaknowledge (knowledge about knowledge). In the past few years, KIF has emerged as the preferred language in efforts to have a standard knowledge representation format for use between a variety of intelligent agents.

The KIF language syntax is reminiscent of Lisp, which is not surprising given its predicate logic basis. Unlike Lisp however, KIF is not meant to be a programming language. Nor is it meant to be used as an internal knowledge representation. KIF was explicitly designed to provide a common format for exchanging information. KIF is powerful and expressive enough to support the requirements of a wide variety of AI programs. Regardless of the internal knowledge representations, as long as every program can read and write KIF, the knowledge is portable and reusable in many different contexts. KIF is formally defined and is the result of several years of effort by the Defense Advanced Research Projects Agency (DARPA) Knowledge Sharing Environment workgroup. The KIF syntax can be split into three major groups: variables, operators, and constants.

KIF supports two types of variables, individual variables which begin with the ? character, and sequence variables which begin with an @ character. The four types of operators include term operators, rule operators, sentence operators, and definition operators. If a token is not a variable or an operator, then it must be a constant. KIF provides distinctions for several different types of constants. All numbers, characters, and strings are basic constants in KIF. Object constants denote objects, function constants denote functions over objects, relation

constants denote relations, and logical constants express boolean conditions about the world which must be either true or false.

The language supports four types of expressions: terms, sentences, rules, and definitions. Terms denote objects, sentences represent facts, rules represent legal steps of inferencing, and definitions are used to define constants. Sentences are made up of constants, terms, and other sentences. KIF defines both forward (premise followed by consequent) and reverse (consequent followed by premise) rules.

A form in KIF is either a sentence, a rule, or a definition. And finally, a KIF knowledge base is a finite set of forms. The order of the forms in a knowledge base is not important.

An example rule in KIF is :

```
(=> (EventName “AGENT:STARTING”)
    (SetIdentifiedIntervalAlarm “NETSCAPE” 20 “minutes”))
```

which translates to “If we receive an AGENT:STARTING event, then start a named interval alarm (a timer) identified by the string “NETSCAPE” to go off every 20 minutes.” The KIF rule:

```
(GetStockPrice ?price) (IntegerCompare ?price “>” 150)
```

means “get the stock price and place it in variable *?price*, then test to see if the price is greater than 150.”

A complex rule can be almost unreadable (no offense intended to Lisp programmers) :

```
(=> (AND (AND (EventName “StockAdapter:StockPriceEvent”)
  (GetStockEventCount ?count)) (IntegerCompare ?count “=” 2))
  (AND (AND ( AND (TurnOffIdentifiedIntervalAlarm “IBM”)
  (StockAdapterShow “Turned off the alarm.” “ ” “ ”))
  (SetIdentifiedIntervalAlarm “HSY” 20 “minutes”))
  (StockAdapterShow “Started an alarm for Hershey.” “ “ “ “)))
```

This means “If this is a StockPriceEvent and this is the second StockPriceEvent event sent from the StockAdapter, then turn off the IBM stock watch timer, display a message, start a timer for Hershey foods, and display another message.” Keep in mind that KIF is not a programming language. By using KIF

as a knowledge representation, we could use multiple rule editors with nice (but very different) graphical environments for authoring rules. As long as they all write out the rules in KIF format, they would be interchangeable. Likewise, if one agent could write out its knowledge in KIF format and another one could read it, they could share their knowledge, even though their internal representations may be totally incompatible. Unfortunately, there is more to agent interaction than just having a common interchange format for knowledge. We discuss some of those issues in Chapter 6.

This brings us to the end of our discussion of knowledge representation formats. We can use procedural code, predicate logic, if-then rules, semantic nets, or frames to represent facts, rules, relationships, and complex transformations. These various kinds of knowledge representations can encode knowledge about almost any problem domain. When combined with a reasoning system or inference engine, a knowledge base becomes a part of a knowledge-based system, the last topic in this chapter.

Building a Knowledge Base

Now that we have described many types of knowledge we want to represent, we need to group it all together in one place, in our knowledge base. The knowledge base is the central repository of information containing the facts we know about objects and their relationships. It was only after several years of AI research that it became clear that knowledge was the key to building successful AI systems. All of the work on search algorithms and general problem-solving methods showed that without deep knowledge of the problem domain, any realistic problem soon got out of the limits of standard search techniques.

The process of mapping the set of knowledge in a particular problem domain and converting it into a knowledge base is called *knowledge engineering* or *knowledge acquisition*. These terms grew out of the development of early expert systems, where several unique roles were identified. First, there is the *domain expert*, who, through years of experience, has gathered the knowledge about how things work and relate to one another, and how to solve problems in his or her specialty. A *knowledge engineer* is a person who can take that domain knowledge and represent it in a form for use by the reasoning system. As an intermediary between the human expert and the expert system, the knowledge engineer must have good people skills as well as good technical skills. A combination of questionnaires, interviews, and first-hand observations are used to give the knowledge engineer the deep understanding required to transform the expert's knowledge into facts and rules for the knowledge base.

This process is called knowledge acquisition and, while essential, it soon became clear that this was a difficult and costly process. Sometimes the experts weren't so keen on having their expertise captured and turned into a computer application that could put them out of a job. Sometimes the experts couldn't really explain "how" they came up with those incredible insights into problems. Sometimes, the problem was that the experts were, well, experts, and so their time was valuable. They needed to solve the problems, not spend hours talking to a knowledge engineer. A third word was soon associated with knowledge acquisition, and that word was *bottleneck*. Several major expert system projects failed in the mid- to late 1980s, and the *knowledge acquisition bottleneck* was identified as the major culprit.

At the same time, artificial intelligence researchers became interested in machine learning techniques. Using historical data from databases or examples generated by experts, neural networks could be trained to perform classification and prediction tasks without going through the expensive knowledge acquisition process. These expert networks, as they are called, performed as well as the painstakingly crafted rule-based systems in many cases. This spurred a renewed interest in adaptive systems which continues today.

Neural networks are more than a solution to the knowledge acquisition problem, they offer an alternative to symbol processing. Neural networks do not manipulate symbols, or anything that can be easily related to symbols. As a data

or knowledge representation, neural networks are essentially a “black box.” The topology and values of the interconnection weights define the neural network and the knowledge it encodes. This representation doesn’t easily lend itself to examination or understanding at the symbolic level. Only the input fields and the output fields are identifiable. The internal states and processing are the result of an adaptive “learning” or “training” process where data is presented to the neural network and the connection weights are automatically adjusted via a learning algorithm.

However, even though neural networks may not be easily converted to a symbolic form, they most definitely are a knowledge base, because they encode the knowledge implicit in the training data. We look at neural networks and learning in more detail in Chapter 5.

Summary

In this chapter we talked about knowledge and knowledge representation techniques commonly used in artificial intelligence programs. The main points are:

- There are many types of *knowledge* including facts and relationships, which may be organized by categories, associations, and hierarchies. The process of taking knowledge and putting it into a computer so we can use it to solve problems is called *knowledge representation*.
- A good knowledge representation must be easily understood by people. It should be unambiguous and easily manipulated by computers and reasoning algorithms.
- There are two primary types of knowledge representations, *procedural* and *declarative*. An example of a procedural knowledge representation is program code. Most knowledge representations used in artificial intelligence are declarative.
- *Propositional* and *predicate logic* can be used to represent objects, functions, and relationships. *Resolution* and *unification* are techniques used to prove theorems and infer new facts using logic.
- *Frames* are comprised of *slots* for data values, and *fillers*, procedures for computing values and enforcing constraints between the slots. Each frame represents a concept and can be linked to other frames to form hierarchical and inheritance relationships.
- *Semantic nets* are graphs used to define relationships between concepts.

Natural language labels on links show the relationships between the nodes.

- *Bayes' rule* is the basis for Bayesian or *causal networks*, which represent probabilistic relationships between variables and allow reasoning under conditions of *uncertainty*.
- *Knowledge Interchange Format* (*KIF*) is a language designed to allow intelligent agents to exchange knowledge. KIF is based on predicate logic, and supports object definitions, functions over objects, relationships between objects, and rules.
- A *knowledge base* is a collection of knowledge related to a specific problem domain. A *knowledge engineer* translates an expert's knowledge into a computer knowledge representation.
- *Machine learning* can be used to overcome the *knowledge acquisition bottleneck. Neural networks* can encode knowledge in their connection weights, but as a knowledge representation they are a "black box."

Exercises

1. Write a set of rules using propositional logic that represent this knowledge:
Intelligent agents have three major attributes: agency, intelligence, and mobility. Agency measures agent autonomy. Intelligence indicates agent reasoning or learning capabilities. Mobility means the agent can move across the network.

2. Rewrite the set of rules in exercise 3.1 using predicate logic.

3. Create a semantic net for the domain of pizza pies or ice cream (your choice).

Chapter 4
Reasoning Systems

In this chapter, we focus on how the various knowledge representations introduced in Chapter 3 can be used for reasoning. We implement a Java applet that uses both forward- and backward-chaining algorithms to process if-then rules. Fuzzy logic and rule processing are also described. Finally, we explore several artificial intelligence techniques used for planning, including goal stack planning, nonlinear planning, and hierarchical planning.

Reasoning with Rules

If-then rules have become the most popular form of declarative knowledge representation used in artificial intelligence applications. There are several reasons for this. Knowledge represented as if-then rules is easily understandable. Most people are comfortable reading rules, in contrast to knowledge represented in predicate logic. Each rule can be viewed as a standalone piece of knowledge or unit of information in a knowledge base. New knowledge can be easily added, and existing knowledge can be changed simply by creating or modifying individual rules.

Rules are easily manipulated by reasoning systems. Forward chaining can be used to produce new facts (hence the term “production” rules), and backward chaining can deduce whether statements are true or not. Rule-based systems were one of the first large-scale commercial successes of artificial intelligence research. An *expert system* or *knowledge-based system* is the common term used to describe a rule-based processing system. It consists of three major elements, a *knowledge base* (the set of if-then rules and known facts), a *working memory* or database of derived facts and data, and an *inference engine*, which contains the reasoning logic used to process the rules and data.

Before we get into the details of reasoning with rules, let’s look at a simple rule:

```
if num_wheels = 4 and motor = yes then vehicleType = automobile
```

has two *antecedent* clauses joined by a conjunction (num_wheels = 4 and motor = yes) and has a single *consequent* clause (vehicleType = automobile). A rule states a relationship between clauses (assertions or facts) and, depending on the situation, can be used to generate new information or prove the truth of an assertion. For example, if we know that a vehicle has four wheels and a motor, then, using the above rule, we can conclude that the vehicleType is an automobile and add that fact to our knowledge base. On the other hand, if we are trying to prove that the vehicleType is an automobile, we need to find out whether the vehicle has four wheels and a motor. In the first case, we are forward chaining, using facts and rules to derive new facts. In the second case, we are backward chaining, trying to prove an assertion in the consequence of a rule by showing that the antecedent clauses are true.

A rule whose antecedent clauses are all true is said to be *triggered* or *ready to fire*. We *fire* a triggered rule by asserting the consequent clause and adding it as a fact to our working memory. At any time, a rule base may contain several rules that are ready to fire. It is up to the control strategy of the inference engine to decide which one gets fired. We will discuss this point in more detail later in this chapter.

Most rule-based systems allow rules to have names or labels such as **Rule1:** or **Automobile:** to easily identify rules for editing or for tracing during inferencing. Some systems allow disjunctions (**or**) between antecedent clauses. This is a short-hand that reduces the size of a rule base. For example:

> Rule 1: if num_wheels = 2 then vehicleType = cycle
> Rule 2: if num_wheels = 3 then vehicleType = cycle
> Rule 3: if (num_wheels = 2 or num_wheels = 3) then vehicleType = cycle

Rule 3 could replace Rule 1 and Rule 2 in the rule base. Most rule systems also allow boolean condition operators such as <, >, and !=, in addition to equality. We could rewrite Rule 3 without using disjunctions:

> Rule 4: if (num_wheels > 1) and (num_wheels < 4) then vehicleType = cycle

Another common enhancement to rule syntax is the addition of a confidence or certainty factor. If we only write rules which applied 100 percent of the time with 100 percent confidence, we would have a very small knowledge base. In

general, we write rules that apply most of the time. These *heuristics* or “rules of thumb” are often sufficient to produce reasonable behavior from a rule-based system. However, in some cases, we must deal with uncertainty in the data, as well as our uncertainty concerning the applicability of the rule. For example, we could write a rule for weather prediction:

> Rule 5: if (weather_forecast = rain) and (weather_probability > 80%) then (chance_of_rain = high) with CF: 90.

That says if the weather forecast says rain is likely with a probability above 80 percent, then we are 90 percent certain that it will rain. If we have little confidence in our local weather prediction service, we may lower the certainty factor of the rule to 50 percent.

Many rule-based systems allow functions to be called from the antecedent clauses. These functions are called *sensors*, because they go out of the inference engine and test some condition in the environment. Sensors usually return boolean values, but they may also return data or facts to the working memory. When functions are allowed in the consequent, they are called *effectors*, which greatly expand the capability of the rule-based system. An effector turns a rule from a fact-generating mechanism into an action-generating device. These action rules allow intelligent agents to do things for us. For example, an intelligent agent that processes e-mail might contain the following rule:

> Rule 6: if sensor(mailArrived) then effector(processMail)

where mailArrived and processMail are defined as functions which interface with the e-mail system, and sensor() and effector() are methods provided by the inferencing system for invoking those functions.

In small numbers, rules can adequately represent many types of domain knowledge. However, as the number of rules grows, the intuitive aspects of if-then rules are diminished, and they lose their effectiveness from a readability perspective. Commercial rule-base systems often allow grouping or partitioning of rules so that they can be treated as logical blocks of knowledge in an attempt to overcome this weakness.

Another common problem in rule-based systems is that as more complete information comes in, or as things change in the outside world, rules which may have been true before become false. The consequence is that we may have to

“take back” some of the “facts” which were generated by the rules. For example, if we see that the grass is wet, we may fire a rule that concludes it is raining. However, we may then get information that the sprinkler system is on. This may cause us to retract our assertion that it is raining or at least lower our confidence or certainty in making that conclusion. This problem of dealing with changes and retracting facts or assertions is called nonmonotonic reasoning. Keeping a rule-base consistent by managing dependencies between inferred facts requires a truth maintenance system. Most reasoning systems, such as predicate logic, are monotonic, that is, they add information but do not retract information from the knowledge base.

In the next two sections we explore forward and backward reasoning with rules. In both sections, we will be using a simple rule base as an example. This rule base should be familiar. We described some of the relations in Chapter 3 when we discussed frames and semantic nets. We call it the Vehicles Rule Base (Figure 4.1). It has only nine rules and seven variables, with one intermediate variable. Seven of the rules define the kind of vehicle, and two are used to determine if the vehicle is a cycle or an automobile. The brevity and clarity of this small rule-base will help us focus on the issues related to rule-based inferencing, and hopefully not get distracted by the details of a more complex problem domain.

Our vehicles domain can identify three types of cycles (one with a motor and two without) and seven types of automobiles. We differentiate cycles as having less than four wheels, and automobiles as having exactly four wheels and a motor. The various types of automobiles are identified by their relative size and the number of doors they have.

Forward Chaining

Forward chaining is a data-driven reasoning process where a set of rules is used to derive new facts from an initial set of data. It does not use the resolution algorithm used in predicate logic. The forward-chaining algorithm generates new data by the simple and straightforward application or firing of the rules. As an inferencing procedure, forward chaining is very fast. Early expert system applications of forward chaining include R1/XCON, an expert system used to build configurations for Digital Equipment Corporation's VAX computer systems (McDermott 1982), and PROSPECTOR, a system used to predict the location of mineral deposits from prospecting data (Duda, et al. 1977). Forward chaining is also used in real-time monitoring and diagnostic systems where quick identification and response to problems are required.

Several commercial tools were developed primarily to do forward-chaining reasoning. The OPS5 language, and the later enhanced OPS83, were developed at Carnegie-Mellon University. OPS5 was used to implement the R1 system. IBM developed KnowledgeTool and TIRS (The Integrated Reasoning System), which were primarily forward-chaining tools. In addition to providing the reasoning capabilities, these commercial systems also provided several different control strategies and the ability to mix in procedural program code, effectively giving sensor and effector capabilities.

Figure 4.1 The vehicles rule base.

Later on, we are going to implement a forward-chaining system, but first, let's look at the reasoning process in more detail. As mentioned before, any expert system requires three basic elements, a knowledge base of rules and facts, a working memory for storing data during inferencing, and an inference engine. The following steps are part of the forward-chaining cycle:

1. Load the rule base into the inference engine, and any facts from the knowledge base into the working memory.
2. Add any additional initial data into the working memory.
3. *Match* the rules against the data in working memory and determine which rules are triggered, meaning that all of their antecedent clauses are true. This set of triggered rules is called the *conflict set*.
4. Use the *conflict resolution* procedure to select a single rule from the conflict set.
5. *Fire* the selected rule by evaluating the consequent clause(s); either update the working memory if it is a fact-generating rule, or call the effector procedure, if it is an action rule. This is referred to as the *act* step.
6. Repeat steps 3, 4, and 5 until the conflict set is empty.

During the match phase of forward chaining the inference system compares the known facts or working memory against the antecedent clauses in the rules to determine which rule or rules could fire. In a knowledge base with many facts and rules, the match phase can take an enormous amount of processing time. Thus, we would like to only test those rules whose antecedent clauses refer to facts which have been updated by the prior rule's firing. The Rete algorithm, developed for the OPS5 language, builds a network data structure to manage the dependencies between the data, condition tests, and rules, and minimizes the number of tests required for each match operation (Forgy 1982). While the Rete algorithm is the Cadillac of match algorithms, many forward-chaining systems use methods that are less efficient but easier to implement.

Once we have completed the match phase and produced the conflict set, we move to the conflict resolution step. Conflict resolution is perhaps the most important step in terms of the behavior of the forward-chaining inferencing system. When the conflict set is empty or contains only a single rule, the problem is trivial. However, in many cases, there will be more than one rule that is triggered. Which rule do we select to fire? Several alternatives are available:

- Select the first rule in the conflict set. This is certainly simple, and for some domains it works.
- Select the rule with the highest specificity or number of antecedent clauses. The idea here is to select the rule that is the most specific (has the most test conditions on the if part of the rule) before we fire more general rules that have fewer antecedent clauses.
- Select the rule that refers to the data which has changed most recently.

This method requires that changes to the working memory are time-stamped or somehow tagged to show when they were last modified.

- If the rule has fired on the previous cycle, do not add it to the conflict set. This rule is sometimes extended to limit rules so they can only fire once.
- In cases where there is a tie, select a rule randomly from this subset of the original conflict set.

In addition to match and conflict resolution, forward-chaining systems often utilize one of several control strategies to help guide the inferencing process. Note that this practice is a diversion from the pure declarative approach of if-then rules. However, to build practical working forward-chaining expert systems these control strategies are required. This is a case of real-world pragmatism versus academic idealism.

A common control strategy in rule-based systems is to assign priorities to rules which can be used to aid in the selection process. If we have a process that proceeds in three phases, we can assign priority 1 to the set of rules that contribute to the first phase, priority 2 to the rules in the second phase, and likewise for the third subset of rules. The advantage of using priorities is that it greatly reduces the number of rules that have to be searched and tested in the match phase.

Another approach that can be used to achieve the same results, even if the inferencing system doesn't formally support priorities, is to use guard clauses. For example, we could add a clause priority = 1 to the antecedents of all rules in group one, priority = 2 in group two and so on. While less efficient than if the inferencing system supports priorities, it does produce the desired behavior.

A Forward-Chaining Example

In this section, we take a look at a simple example of forward chaining in our vehicle domain. To start, we load our vehicle rule base into the inference engine and define a set of initial values for variables in the working memory:

```
num_wheels=4
motor=yes
num_doors = 3
size=medium
```

Next, we do a match phase, where we examine the antecedent clauses of each

rule to determine which ones can be triggered. We have no value for *vehicleType*, so the first seven rules are not triggered. The last two rules require values for *num_wheels* and motor, so they are candidates. The *num_wheels < 4* clause in Cycle: rule is false, so that is not triggered, but *num_wheels = 4* and *motor = yes* are both true and so the Automobile rule is triggered. Our conflict set from our first match cycle contains a single rule, the Automobile: rule.

```
Automobile: IF num_wheels=4
 AND motor=yes
 THEN vehicleType=automobile
```

Conflict resolution is easy: We select the single rule and fire it in the act cycle. Firing the Automobile: rule causes us to bind the value “automobile” to the *vehicleType* variable, and add it to our working memory:

```
num_wheels=4
motor=yes
num_doors = 3
size=medium
vehicleType=automobile
```

Now we are ready for our next inferencing cycle. We do a match against the rules to determine which ones could be fired. Now that *vehicleType* has a value, the first seven rules are candidates. However, the first three rules require that *vehicleTyp*e = cycle, which is false. This leaves the next four rules: SportsCar, Sedan, MiniVan, and SUV. Only a single rule has all of its antecedent clauses satisfied, the MiniVan: rule with *num_doors = 3* and *size = medium*. Once again, our conflict set has only a single rule in it:

```
MiniVan: IF vehicleType=automobile
 AND size=medium
 AND num_doors=3
 THEN vehicle=MiniVan
```

We fire the MiniVan: rule and add the new information that *vehicle = MiniVan* to the working memory:

```
num_wheels=4
motor=yes
num_doors = 3
size=medium
vehicleType=automobile
vehicle=MiniVan
```

We do yet another match phase and find that only one rule is triggered again, the MiniVan: rule. However, because it has already fired, we do not add it to the conflict set. Our conflict set is now empty, so we halt our forward-chaining inferencing. In this example, we started with four facts and computed two new facts, determining that the *vehicle* is a MiniVan.

Previous | Table of Contents | Next

Backward Chaining

Backward chaining is often called goal-directed inferencing, because a particular consequence or goal clause is evaluated first, and then we go backward through the rules. Unlike forward chaining, which uses rules to produce new information, backward chaining uses rules to answer questions about whether a goal clause is true or not. Backward chaining is more focused than forward chaining, because it only processes rules that are relevant to the question. It is similar to how resolution is used in predicate logic. However, it does not use contradiction. It simply traverses the rule base trying to prove that clauses are true in a systematic manner.

Backward chaining is used for advisory expert systems, where users ask questions and get asked leading questions to find an answer. A famous early expert system, Mycin, used backward chaining to perform diagnoses of bacterial infections in medical patients (Shortliffe 1976).

One advantage of backward chaining is that, because the inferencing is directed, information can be requested from the user when it is needed. Some reasoning systems also provide a trace capability which allows the user to ask the inference engine why it is asking for some piece of information, or why it came to some conclusion.

We are going to implement a backward-chaining system later in this chapter. Now, let’s look at the steps that are part of the backward-chaining cycle:

1. Load the rule base into the inference engine, and any facts from the knowledge base into the working memory.
2. Add any additional initial data into the working memory.
3. Specify a goal variable for the inference engine to find.
4. Find the set of rules which refer to the goal variable in a consequent clause. That is, find all rules which set the value of the goal variable when they fire. Put each rule on the goal stack.
5. If the goal stack is empty, halt.
6. Take the top rule off the goal stack.
7. Try to prove the rule is true by testing all antecedent clauses to see if they are true. We test each antecedent clause in turn: (A) If the clause is

true, go on to the next antecedent clause. (B) If the clause is false, then pop the rule off the goal stack; go to step 5. (C) If the truth value is unknown because the antecedent variable is unknown, go to step 4, with the antecedent variable as the new goal variable. (D) If all antecedent clauses are true, fire the rule, setting the consequent variable to the consequent value, pop the rule off the goal stack, and go to 5.

Let's look more closely at how the backward-chaining algorithm works using a rule from our vehicle rule base as an example. Suppose we want to find out whether the *vehicle* we have is a MiniVan. This is the rule that must be satisfied:

```
MiniVan: IF vehicleType=automobile
 AND size=medium
 AND num_doors=3
THEN vehicle=MiniVan
```

We start with an empty working memory. No facts are known about the vehicle attributes. The first thing we would do is check working memory to see if *vehicle = MiniVan* is already true. If not, then all of the antecedent clauses of the MiniVan: rule must be true to safely conclude that the *vehicle* is a MiniVan. Consequently, we must try to prove each antecedent clause in turn. The first thing we do is test if *vehicleType = automobile* is true. The *vehicleType* variable is has no value, so we look for a rule that has *vehicleType = automobile* in its consequent clause, and find the Automobile: rule below:

```
Automobile: IF num_wheels=4
 AND motor=yes
 THEN vehicleType=automobile
```

This is an example of backward chaining. We start with the MiniVan: rule, and, in the course of proving that true, we have chained to another rule, the Automobile: rule. These rules are linked by the *vehicleType = automobile* clause that they both share. Focused now on the Automobile: rule, we need to know if *num_wheels = 4*. We look in the working memory and see that *num_wheels* has no value. We look for a rule that has *num_wheels = 4* as a consequent. There are none. Now, we could either give up, or we could ask the user to provide an answer. We ask the user, who says there are four wheels on the vehicle. The first antecedent clause is true, so we move onto the second clause, *motor = yes*. We check the working memory, and *motor* has no value. We again search the rule base for a rule with *motor = yes* in the consequent and can find none. We ask the user to provide a value. The user answers that the vehicle has a motor, so *motor*

= *yes* is true. Both of the antecedent clauses are now true. We have proved that the Automobile: rule is true, and we can set *vehicleType = automobile*. Our working memory now contains the following facts:

```
num_wheels=4
motor=yes
vehicleType=automobile
```

Going back to our original rule, we now know that the first antecedent clause is true. We next need to find values for *size* and *num_doors*. Using the same process described above, we end up asking the user for these values. The user indicates that *size = medium* and *num_doors = 3*. All the antecedent clauses have been satisfied, so we can conclude that the *vehicle* is a MiniVan. Our final working memory contains:

```
num_wheels=4
motor=yes
vehicleType=automobile
size=medium
num_doors=3
vehicle=MiniVan
```

While our little example worked out, in many cases the rule we are trying to prove is not true. We may need to search other paths through alternate rules in order to answer the user's question. The backward-chaining algorithm performs what amounts to a depth-first search through the rule base while trying to prove a goal clause.

The Java Rule Applet

In this section, we present an applet which implements the two major types of reasoning algorithms used with rule-based systems: forward and backward chaining. The Rule applet features three text panes for displaying the rule base, the variables and their current values, and a trace of the inferencing process. Figure 4.2 shows the Rule applet.

Figure 4.2 The Rule applet dialog.

At the top of the dialog, users can select one of the three sample rule bases provided, the vehicle rule base already discussed, a bugs rule base, and a plants rule base. The rules of the selected rule base are displayed in the **java.awt. TextArea** control at the top left. To the right are two **java.awt Choice** controls (drop-down list boxes) that allow the user to specify the variable and a corresponding value for the selected variable. Note that both the variable and value must be selected in order for the variable to be changed. Whenever a variable is changed, its new value is logged in the variable **TextArea** control which is immediately below the **Choice** controls.

Near the bottom of the dialog, two radio button controls allow the user to specify whether forward or backward chaining should be used. If backward chaining is specified, the name of the goal variable must be entered in the Goal **java.awt TextField** to the right of the radio buttons.

There are three pushbuttons at the very bottom of the dialog. The leftmost one is **Find Goal**, which causes an inferencing cycle to be performed. This cycle will use the current variable-value settings. After a **Find Goal** is performed, you can press the **Reset** button on the bottom right. This will set all of the rule-base variables to null values. When the middle button, **Run Demo,** is clicked, depending on the rule base and chaining method selected, the variables will be set to preset values, and an automatic **Find Goal** is performed.

Our Java implementation includes a **Rule** class, a **RuleVariable** class, and a **RuleBase** class, as well as several support classes such as **Clause**. We start our discussion with the **Rule** class.

Rules

The **Rule** class is used to define a single rule and also contains methods that support the inferencing process. Each **Rule** has a name data member, a reference to the owning **RuleBase** object (described later), an array of antecedent **Clauses**, and a single consequent **Clause**. The **Rule**'s truth value is stored in the **Boolean** *truth*. Note, this is a **Boolean** object, not an elementary boolean variable. This allows us to use a null value to indicate that the rule's truth cannot be determined (because one of the variables referenced in a clause is also null or undefined). The *fired* boolean member indicates whether this rule has been fired or not.

There are several **Rule** constructors, each requiring a reference to the **RuleBase**

instance, the **Rule** name, one or more antecedent or left-hand-side (LHS) clauses, and the single consequent or right-hand-side (RHS) clause. Each constructor allocates the correct number of entries in the *antecedents* array, and also registers itself with the **Clause** objects as it adds them to its data members. The **Rule** *truth* is initialized to null, meaning undefined or unknown, and the **Rule** registers itself with the owning **RuleBase**.

```
public class Rule {
  RuleBase rb ;
  String name ;
  Clause antecedents[] ;  // allow up to 4 antecedents for now
  Clause consequent ;     //only 1 consequent clause allowed
  Boolean truth;       // states = (null=unknown, true, or false)
  boolean fired=false;
  Rule(RuleBase Rb, String Name, Clause lhs, Clause rhs) {
    rb = Rb ;
    name = Name ;
    antecedents = new Clause[1] ;
    antecedents[0] = lhs ;
    lhs.addRuleRef(this) ;
    consequent = rhs ;
    rhs.addRuleRef(this) ;
    rhs.isConsequent() ;
    rb.ruleList.addElement(this) ;  // add self to rule list
    truth = null ;
  }

 Rule(RuleBase Rb, String Name, Clause lhs1, Clause lhs2,
                                Clause rhs) {
    rb = Rb ;
    name = Name ;
    antecedents = new Clause[2] ;
    antecedents[0] = lhs1 ;
    lhs1.addRuleRef(this) ;
    antecedents[1] = lhs2 ;
    lhs2.addRuleRef(this) ;
    consequent = rhs ;
    rhs.addRuleRef(this) ;
    rhs.isConsequent() ;
    rb.ruleList.addElement(this) ;  // add self to rule list
    truth = null ;
  }
 Rule(RuleBase Rb, String Name, Clause lhs1, Clause lhs2,
                               Clause lhs3, Clause rhs) {
    rb = Rb ;
    name = Name ;
    antecedents = new Clause[3] ;
    antecedents[0] = lhs1 ;
```

```
    lhs1.addRuleRef(this) ;
    antecedents[1] = lhs2 ;
    lhs2.addRuleRef(this) ;
    antecedents[2] = lhs3 ;
    lhs3.addRuleRef(this) ;
    consequent = rhs ;
    rhs.addRuleRef(this) ;
    rhs.isConsequent() ;
    rb.ruleList.addElement(this) ;  // add self to rule list
    truth = null ;
  }
 Rule(RuleBase Rb, String Name, Clause lhs1, Clause lhs2,
                                Clause lhs3, Clause lhs4,
                                Clause rhs) {
    rb = Rb ;
    name = Name ;
    antecedents = new Clause[4] ;
    antecedents[0] = lhs1 ;
    lhs1.addRuleRef(this) ;
    antecedents[1] = lhs2 ;
    lhs2.addRuleRef(this) ;
    antecedents[2] = lhs3 ;
    lhs3.addRuleRef(this) ;
    antecedents[3] = lhs4 ;
    lhs4.addRuleRef(this) ;
    consequent = rhs ;
    rhs.addRuleRef(this) ;
    rhs.isConsequent() ;
    rb.ruleList.addElement(this) ;  // add self to rule list
    truth = null ;
  }

  long numAntecedents() { return antecedents.length; }
  . . .
}
```

Previous | Table of Contents | Next

Clauses

Clauses are used both in the antecedent and consequent parts of a **Rule**. A **Clause** is usually made up of a **RuleVariable** on the left-hand side, a **Condition**, which tests equality, greater than, or less than, and the right-hand side, which in our implementation is a **String** (symbolic or numeric) value. For example the rule:

```
Automobile: IF num_wheels=4
 AND motor=yes
 THEN vehicleType=automobile
```

contains three clauses. The first antecedent clause is made up of the **RuleVariable** "num_wheels", the **Condition** "=", and the **String** "4". The other clauses are similarly composed. A **Clause** also contains a vector of the **Rules** which contain this **Clause**, a consequent boolean which indicates whether the clause appears in the antecedent or the consequent of the rule, and a truth **Boolean** which indicates whether the clause is true, false, or unknown (null).

The **Clause** constructor takes a **RuleVariable** for the left-hand side, a **Condition** object, and a **String** for the right-hand side. The **Clause** registers itself with the **RuleVariable** so that whenever the variable's value is changed, the **Clause** can be automatically retested. The *consequent* **Boolean** is set to false, initially, because most clauses are antecedent clauses.

The **Clause** class contains four methods. The *addRuleRef()* method is used by the **Rule** constructor to register the **Rule** with this **Clause**. The *check()* method performs a test of the clause. If the clause is used as a consequent clause, then testing its truth value makes no sense; we return a null value. If the variable on the left-hand side is unbound, we also return null, because a truth value cannot be determined. If the variable is bound, we use the switch statement to test the specified logical condition and return the resulting truth value. The *isConsequent()* returns the consequent **Boolean** and the *getRule()* method returns a reference to the owning **Rule** instance.

```
public class Clause {
    Vector ruleRefs ;
    RuleVariable lhs ;
```

```
    String rhs ;
    Condition  cond ;
    Boolean consequent ;  // true or false
    Boolean truth ;   // states = null(unknown), true or false
    Clause(RuleVariable Lhs, Condition Cond, String Rhs)
    {
     lhs = Lhs ; cond = Cond ; rhs = Rhs ;
     lhs.addClauseRef(this) ;
     ruleRefs = new Vector() ;
     truth = null ;
     consequent = new Boolean(false) ;
    }

    void addRuleRef(Rule ref) { ruleRefs.addElement(ref) ; }

    Boolean check() {
        if (consequent.booleanValue() == true) return null ;
        if (lhs.value == null) {
            return truth = null ;  // var value is undefined
        } else {

        switch(cond.index) {
       case 1: truth = new Boolean(lhs.value.equals(rhs)) ;
         break ;
       case 2: truth = new Boolean(lhs.value.compareTo(rhs) > 0) ;
         break ;
       case 3: truth = new Boolean(lhs.value.compareTo(rhs) < 0) ;
         break ;
       case 4: truth = new Boolean(lhs.value.compareTo(rhs) != 0) ;
         break ;
       }

       return truth ;
        }
    }

    void isConsequent() { consequent = new Boolean(true); }

    Rule getRule() { if (consequent.booleanValue() == true)
        return (Rule)ruleRefs.firstElement() ;
        else return null ;}
};
```

The **Condition** class is a helper class to **Clause**. It takes a **String** representation of a conditional test and converts that into a code for use in the *switch* statement in the *Clause.check()* method.

```
public class Condition {
```

```
    int index ;
    String symbol ;
    Condition(String Symbol) {
        symbol = Symbol ;
        if (Symbol.equals(“=”)) index = 1 ;
        else if (Symbol.equals(“>”)) index = 2 ;
        else if (Symbol.equals(“<”)) index = 3 ;
        else if (Symbol.equals(“!=”)) index = 4 ;
        else index = -1 ;
    }
    String asString() {
        String temp = new String() ;
        switch (index) {
         case 1: temp = “=” ;
         break;
         case 2: temp = “>” ;
         break ;
         case 3: temp = “<” ;
         break;
         case 4: temp = “!=” ;
         break;
        }
     return temp ;
    }
}
```

Variables

We define a base class for variables which support the function we need for rule processing and for learning in the next chapter. The **Variable** class has a *name* member to identify the variable, and a **String** *value* member (which could be an **Object** in a more general-purpose application). The *column* is used to specify the position of the variable in a data file. There is a default constructor, as well as one where the name is specified. Two accessor methods are provided to set the *value* and get the *value* of the **Variable**. The *labels* member is used to hold discrete symbols for categorical variables. The *setLabels()* method is used to define the valid symbolic values for categorical variables. The *getLabel()* method returns the symbolic value for the specified index and the inverse method *getIndex()* returns the index given a symbolic value. An example of how these methods are used is shown in the section on the Vehicle Rule Base implementation where the **Variables** are defined.

```
public abstract class Variable {
```

```
String name ;
String value ;
int column ;

public Variable() {} ;
public Variable(String Name) {name = Name; value = null; }
void   setValue(String val) { value = val ; }
String getValue() { return value; }

// used by categorical only
Vector labels ;
void setLabels(String Labels) {
    labels = new Vector() ;
    StringTokenizer tok = new StringTokenizer(Labels,“ ”) ;
    while (tok.hasMoreTokens()) {
         labels.addElement(new String(tok.nextToken())) ;
    }
}
// return the label with the specified index
String getLabel(int index) {
    return (String)labels.elementAt(index);
}

// return a string containing all labels
String getLabels() {
    String labelList = new String();
    Enumeration enum = labels.elements() ;
    while(enum.hasMoreElements()) {
         labelList += enum.nextElement() + “ ” ;
    }
    return labelList ;
}

// given a label, return its index
int getIndex(String label) {
    int i = 0, index = 0  ;
    Enumeration enum = labels.elements() ;
    while(enum.hasMoreElements()) {
        if (label.equals(enum.nextElement()))
        { index = i ; break ; }
        i++;
     }
     return i;
}
```

Rule Variables

For rule processing, we subclass our **Variable** class and add some rule-specific

behavior. It provides the support necessary for variables used in inferencing. The constructor takes the *name* of the variable as the only parameter. **RuleVariables** inherit the discrete symbolic behavior of the base **Variable** class. A new data member is the **Vector** *clauseRefs*, which holds references to all **Clauses** which refer to this variable. Instances of **Clause** register themselves by calling the *addClauseRef()* method. There are several methods which are overridden as well as some new ones added for rule processing. The *setValue()* method not only sets the *value* of the variable, it also calls the *updateClauses()* method, which iterates through every **Clause** which refers to this **RuleVariable** and retests its *truth* value via its *check()* method.

The *promptString* holds the text that is displayed when the user is prompted to provide a value for this variable. The *ruleName* holds the name of the rule which set this **RuleVariables** value; when the rule fires it calls the *setRuleName()* method accordingly. The *askUser()* method calls the *waitForAnswer()* method on the **RuleApplet**. This opens a dialog box which prompts the user to provide a value for the variable during inferencing. This is only used by the backward-chaining algorithm.

```
public class RuleVariable extends Variable {

public RuleVariable(String Name) {
        super(Name);
        clauseRefs = new Vector();
        }

void setValue(String val) { value = val;
                            updateClauses(); }

// prompt a user to provide a value for a variable during inferencin
String askUser() {
  String answer = RuleApplet.waitForAnswer(name, getLabels()) ;
  RuleBase.appendText("\n  !!! Looking for " + name +
                      ". User entered: " + answer) ;
  setValue(answer) ; // need to set value from textField here
  return value ;
}

Vector clauseRefs ;   // clauses which refer to this var
void addClauseRef(Clause ref) { clauseRefs.addElement(ref) ; }

void updateClauses() {
   Enumeration enum = clauseRefs.elements() ;
   while(enum.hasMoreElements()) {
     ((Clause)enum.nextElement()).check() ; // retest the clause
```

```
   }
}
String promptString ;  // used to prompt user for value
String ruleName ;      // if value is inferred, null = user provided
void   setRuleName(String rname) { ruleName = rname; }

 ...

};
```

Previous | Table of Contents | Next

Rule Base

The **RuleBase** class defines a set of **RuleVariables** and **Rules**, along with the high-level methods for forward and backward chaining. The **RuleBase** has a *name*, a *variableList* which contains all of the **RuleVariables** referenced by the **Rules**, and the *ruleList*, which contains all of the **Rules**. The *forwardChain()* and *backwardChain()* methods as well as other **RuleBase** data members that are used by the inferencing algorithms are described in their respective sections later in this chapter.

```
public class RuleBase {

 String name ;
 Hashtable variableList ;    // all variables in the rulebase
 Clause clauseVarList[];
 Vector ruleList ;           // list of all rules
 Vector conclusionVarList ;  // queue of variables
 Rule rulePtr ;              // working pointer to current rule
 Clause clausePtr ;          // working pointer to current clause
 Stack goalClauseStack;      // for goals (cons clauses) and subgoal

 static TextArea textArea1 ;
 public void setDisplay(TextArea txtArea) { textArea1 = txtArea; }

 RuleBase(String Name) { name = Name; }
 public static void appendText(String text) {
                         textArea1.appendText(text); }

 // for trace purposes - display all variables and their value
 public void displayVariables(TextArea textArea) {

   Enumeration enum = variableList.elements() ;
   while(enum.hasMoreElements()) {
     RuleVariable temp = (RuleVariable)enum.nextElement() ;
     textArea.appendText("\n" + temp.name + " value = "
                         + temp.value) ;
   }
 }

// for trace purposes - display all rules in text format
 public void displayRules(TextArea textArea) {
   textArea.appendText("\n" + name + " Rule Base: " + "\n");
   Enumeration enum = ruleList.elements() ;
```

```
    while(enum.hasMoreElements()) {
      Rule temp = (Rule)enum.nextElement() ;
      temp.display(textArea) ;
    }
  }

  // for trace purposes - display all rules in the conflict set
  public void displayConflictSet(Vector ruleSet) {
    textArea1.appendText("\n" + " -- Rules in conflict set:\n");
    Enumeration enum = ruleSet.elements() ;
    while(enum.hasMoreElements()) {
      Rule temp = (Rule)enum.nextElement() ;
      textArea1.appendText(temp.name + "(" + temp.numAntecedents()+ "
    }
  }

  // reset the rule base for another round of inferencing
  // by setting all variable values to null
  public void reset() {
    textArea1.appendText("\n --- Setting all " + name + " variables t
    Enumeration enum = variableList.elements() ;
    while(enum.hasMoreElements()) {
      RuleVariable temp = (RuleVariable)enum.nextElement() ;
      temp.setValue(null) ;
    }
  }

...

}
```

Forward-Chaining Implementation

Our forward-chaining implementation uses methods in both the **RuleBase** class and the **Rule** class. The *forwardChain()* method in the **RuleBase** class contains the main control logic for forward chaining. The method first allocates the *conflictRuleSet* vector. The *match()* method is called with a boolean true parameter to force an initial test of all rules in the rule base. This returns with the initial *conflictRuleSet*, a **Vector** of the rules which are triggered and could be fired. We then enter a *while()* loop, which runs until we have an empty *conflictRuleSet*. Inside the loop, we first call the *selectRule()* method, passing the *conflictRuleSet* as a parameter. The *selectRule()* method performs the conflict resolution strategy and returns with a single rule to fire. We call the *Rule.fire()* method to perform the consequent clause assignment, and then retest all **Clauses** and **Rules** which refer to the updated **Variable**. While not a Rete

implementation, this approach limits the amount of clause testing that needs to be performed. With the updated variableList, we call *match()* again, but this time we pass in a boolean false parameter value. This tells *match()* to only look at the rule truth values, not to test each rule.

```
public void forwardChain() {
    Vector conflictRuleSet = new Vector() ;

    // first test all rules, based on initial data
    conflictRuleSet = match(true); // see which rules can fire

    while(conflictRuleSet.size() > 0) {

      Rule selected = selectRule(conflictRuleSet); // select the “be
      selected.fire() ; // fire the rule
                        // do the consequent action/assignment
                        // update all clauses and rules

      conflictRuleSet = match(false); // see which rules can fire
    }
 }
```

Now let’s look at the individual methods in more detail. The *RuleBase.match()* method takes a single boolean parameter. It walks through the *ruleList*. If the *test* parameter is true, it calls the *Rule.check()* method to test all rule antecedent clauses and set the rule’s truth value. If *test* is false, *match()* simply looks at the current rule’s truth value. If the rule is true, and it hasn’t already been fired, it will add it to the *matchList* **Vector**. If not, we just continue on to the next **Rule** on the *ruleList*. For tracing purposes, we display the conflict set.

```
// used for forward chaining only
// determine which rules can fire, return a Vector
public Vector match(boolean test) {
   Vector matchList = new Vector() ;
   Enumeration enum = ruleList.elements() ;
   while (enum.hasMoreElements()) {
     Rule testRule = (Rule)enum.nextElement() ;
     if (test) testRule.check() ; // test the rule antecedents
     if (testRule.truth == null) continue ;
     // fire the rule only once for now
     if ((testRule.truth.booleanValue() == true) &&
         (testRule.fired == false)) matchList.addElement(testRule);
   }
   displayConflictSet(matchList) ;
   return matchList ;
}
```

The *RuleBase.selectRule()* method takes a **Vector** of rules, the conflict set, as an input parameter. Our implementation is fairly simple. We use specificity, that is, the number of antecedent clauses, as our primary method for selecting a rule to fire. If two or more rules have the same number of antecedent clauses, we select the first rule we encounter. We start by taking the first rule off the list and designate it as our *bestRule*, also taking the number of antecedents as the current best or *max* value. In a *while()* loop, we walk through the rest of the conflict set. If we encounter a rule that has more antecedent clauses than our previous *max*, we set that rule as our current *bestRule* and corresponding *max* value. After looking at all of the rules in the conflict set, we return the final *bestRule* to be fired.

```
// used for forward chaining only
// select a rule to fire based on specificity
public Rule selectRule(Vector ruleSet) {
   Enumeration enum = ruleSet.elements() ;
   long numClauses ;
   Rule nextRule ;

   Rule bestRule = (Rule)enum.nextElement() ;
   long max = bestRule.numAntecedents() ;
   while (enum.hasMoreElements()) {
       nextRule = (Rule)enum.nextElement() ;
       if ((numClauses = nextRule.numAntecedents()) > max) {
           max = numClauses ;
           bestRule = nextRule ;
       }
   }
   return bestRule ;
}
```

The *Rule.check()* method is used during forward chaining to test the antecedent clauses of the rule. If any of the clauses has an undefined truth value, then a null value is returned by *check()*. If any of the clauses is false, then the rule's truth value is set to false and a false value is returned. If all of the antecedent clauses are true, then the rule's truth value is set to true, and a true value is returned. Note that only conjunctions (**and**) are supported, not disjunctions (**or**) between clauses.

```
// used by forward chaining only !
  Boolean check() {  // if antecedent is true and rule has not fired
   RuleBase.appendText("\nTesting rule " + name ) ;
   for (int i=0 ; i < antecedents.length ; i++ ) {
    if (antecedents[i].truth == null) return null ;
```

```
  if (antecedents[i].truth.booleanValue() == true) {
      continue ;
  } else {
      return truth = new Boolean(false) ; //don't fire this rule
  }
 } // endfor
 return truth = new Boolean(true) ;  // could fire this rule
}
```

Previous | Table of Contents | Next

The *Rule.fire()* method is used during forward chaining when a rule with a true truth value is selected to be fired. The fired boolean flag is set to show that the rule has fired. The **RuleVariable** on the left-hand side of the consequent clause is set to the value of the right-hand side. The *checkRules()* method is then called to retest those rules which refer to the consequent variable.

```
// used by forward chaining only !
// fire this rule -- perform the consequent clause
// if a variable is changes, update all clauses where
//  it is references, and then all rules which contain
//  those clauses
void fire() {
  RuleBase.appendText("\nFiring rule " + name ) ;
  truth = new Boolean(true) ;
  fired = true ;
  // set the variable value and update clauses
  consequent.lhs.setValue(consequent.rhs) ;
  // now retest any rules whose clauses just changed
  checkRules(consequent.lhs.clauseRefs) ;
}
```

The *Rule.checkRules()* method retests every clause which refers to the **RuleVariable** which was changed by the firing of the rule. Because the **RuleVariable** now has a value, all of the antecedent clauses which referred to it and had undefined or null truth values can now be set to either true or false. This means that **Rules** which referred to those clauses may now evaluate as either true or false.

```
// used by forward chaining only !
// a variable value was found, so retest all clauses
// that reference that variable, and then all rules which
// references those clauses
public static void checkRules(Vector clauseRefs) {
 Enumeration enum = clauseRefs.elements();
while(enum.hasMoreElements()) {
  Clause temp = (Clause)enum.nextElement();
  Enumeration enum2 = temp.ruleRefs.elements() ;
  while(enum2.hasMoreElements()) {
      ((Rule)enum2.nextElement()).check() ; // retest the rule
  }
}
```

The *Rule.display()* method writes the rule out in a text format in the upper

TextArea of the **RuleApplet**. This allows the user to examine the **RuleBase** to follow a chain of inference.

```
// display the rule in text format
void display(TextArea textArea) {
  textArea.appendText(name +“: IF ”) ;
  for(int i=0 ; i < antecedents.length ; i++) {
    Clause nextClause = antecedents[i] ;
    textArea.appendText(nextClause.lhs.name +
                        nextClause.cond.asString() +
                        nextClause.rhs + “ ”) ;
    if ((i+1) < antecedents.length)
        textArea.appendText(“\n     AND ”) ;
  }
  textArea.appendText(“\n     THEN ”) ;
     textArea.appendText(consequent.lhs.name +
                        consequent.cond.asString() +
                        consequent.rhs + “\n”) ;
}
```

Backward-Chaining Implementation

The *RuleBase.backwardChain()* method takes a single parameter, a **String** which is the name of the goal variable. This variable name is used to retrieve the goal’s **RuleVariable** instance. All clauses which refer to the goal variable are enumerated and a *while()* loop is used to process each **Clause** object. If it is not a consequent clause, it is ignored and we continue through the loop to examine the next *goalClause*. If it is a consequent clause, we push it onto our *goalClauseStack*. We then get a reference to the **Rule** that contains this clause as its consequent. We call *Rule.backChain()* on that rule to see if it is true or not. *Rule.backChain()* will make recursive calls to *RuleBase.backwardChain()*, if necessary, to follow a chain of inferences through the rule base in order to find out whether the original *goalClause* is true or false. *Rule.backChain()* will return the **Rule**’s truth value.

- If the rule’s truth value is null, we could not determine whether the current *goalClause* is true or not. Either the rule base is incomplete, or the user provided an invalid value when prompted to provide one.
- If the rule was proven true, we fire the rule by setting the current goal variable to the value on the right-hand side of the *goalClause*; we add a reference to the variable to tell it what rule produced its value; we pop the clause off the *goalClauseStack* and display a success message. If the

goalClauseStack is empty, we are done backward chaining, so we display a victory message and break out of the loop.

- If the rule was false, we pop the *goalClause* from the *goalClauseStack*, display a failure message, and continue through the *while()* loop to process the next *goalClause*.

```
 // for all consequent clauses which refer to this goalVar
 // try to find goalVar value via a rule being true
 //     if rule is true then pop, assign value, re-eval rule
 //     if rule is false then pop, continue
 //     if rule is null then we couldnt find a value
 //
public void backwardChain(String goalVarName)
{
 RuleVariable goalVar = (RuleVariable)variableList.get(goalVarName);
 Enumeration goalClauses = goalVar.clauseRefs.elements() ;

 while (goalClauses.hasMoreElements()) {
    Clause goalClause = (Clause)goalClauses.nextElement() ;
    if (goalClause.consequent.booleanValue() == false) continue ;

    goalClauseStack.push(goalClause) ;

    Rule goalRule = goalClause.getRule();
    Boolean ruleTruth = goalRule.backChain(); // find rule truth
    if (ruleTruth == null) {
       textArea1.appendText("\nRule " + goalRule.name +
                            " is null, can't determine truth value.");
     } else if (ruleTruth.booleanValue() == true) {
      // rule is OK, assign consequent value to variable
      goalVar.setValue(goalClause.rhs) ;
      goalVar.setRuleName(goalRule.name) ;
      goalClauseStack.pop() ;  // clear item from subgoal stack
      textArea1.appendText("\nRule " + goalRule.name +
                  " is true, setting " + goalVar.name + ": = "
                  + goalVar.value);

      if (goalClauseStack.empty() == true) {
        textArea1.appendText("\n +++ Found Solution for goal: "
                             + goalVar.name);
        break ; // for now, only find first solution, then stop
      }
    } else {
      goalClauseStack.pop() ; // clear item from subgoal stack
      textArea1.appendText("\nRule " + goalRule.name +
                           " is false, can't set " + goalVar.name);
    }
  }
```

```
  if (goalVar.value == null) {
    textArea1.appendText("\n +++ Could Not Find Solution for goal: "
                         + goalVar.name);
  }
}
```

The *Rule.backChain()* method will try to prove a rule either true or false by recursively calling *RuleBase.backwardChain()* until the truth value can be determined. The method consists of a *for()* loop, where each antecedent clause is evaluated in turn. If the variable in an antecedent clause is undefined, then *RuleBase.backwardChain()* is called to determine its value. If a value cannot be inferred, the user is prompted for a value using the *RuleVariable.askUser()* method. Once the user provides a value, the clause is tested using the *Clause.check()* method. If the clause is true, we continue through the loop to evaluate the next clause. If it was false, we exit, reporting that the rule's truth value is false because one of the antecedent clauses is false. If we get through the entire loop, then all of the antecedent clauses are true, so we set and return true as the **Rule**'s truth value.

```
// determine if a rule is true or false
// by recursively trying to prove its antecedent clauses are true
// if any are false, the rule is false
Boolean backChain()
{
 RuleBase.appendText("\nEvaluating rule " + name) ;
 for (int i=0; i < antecedents.length; i++) { // test each clause
   if (antecedents[i].truth == null)
        rb.backwardChain(antecedents[i].lhs.name);
   if (antecedents[i].truth == null) { // couldn't prove t or f
       antecedents[i].lhs.askUser() ;  // so ask user for help
       truth = antecedents[i].check() ; // redundant?
   }
   if (antecedents[i].truth.booleanValue() == true) {
       continue ;           // test the next antecedent (if any)
   } else {
       return truth = new Boolean(false) ; // exit, one is false
   }
 }
 return truth = new Boolean(true) ; // all antecedents are true
}
```

Rule Applet Implementation

Our Rule applet uses the **RuleBase**, **Rule**, and **RuleVariable** classes and provides a graphical interface for experimentation. The main applet dialog was designed using Symantec Visual CafŽ but this code could have been implemented manually or with some other Java GUI builder tool. The **RuleApplet** class is a subclass of the **Applet** class.

The first six methods process user actions on the three **Choice** controls—the rule base selector, the variable selector, and the value selector—and the three buttons —the **Find**, **Run Demo**, and **Reset** buttons respectively.

The *init()* method contains a section of code generated by the Visual CafŽ visual builder to initialize the applet awt controls. At the bottom of the *init()* method, we instantiate a frame to be the parent of the **RuleVarDialog**, initialize the rule base **Choice** control, and then instantiate and initialize the three example **RuleBase** objects. The *handleEvent()* method routes the user actions to the appropriate event-handling methods. Note that this is Java 1.0.2 event-handling style. The Java 1.1 event model is quite different, but Symantec did not have support for Java 1.1 when we wrote this code. Next, the awt controls are allocated by CafŽ generated code, and we provide static member variables to hold our frame and **RuleBase** instances. The *currentRuleBase* member always points to the selected rule base in the *choice1* control.

```
/*  RuleApplet class

   Constructing Intelligent Agents with Java
   (C) Joseph P.Bigus and Jennifer Bigus 1997

*/

import java.awt.*;
import java.applet.*;
import java.util.* ;

public class RuleApplet extends Applet {

    // user selected a rule base
    void choice1_Clicked() {
       String rbName = choice1.getSelectedItem() ;
```

```
   if (rbName.equals("Vehicles")) currentRuleBase = vehicles ;
   if (rbName.equals("Bugs")) currentRuleBase = bugs ;
   if (rbName.equals("Plants")) currentRuleBase = plants ;

   currentRuleBase.reset() ;  // reset the rule base
   Enumeration vars = currentRuleBase.variableList.elements() ;
   while (vars.hasMoreElements()) {
     choice2.addItem(((RuleVariable)vars.nextElement()).name) ;
   }
   currentRuleBase.displayVariables(textArea3) ;

 }

// user selected a variable
void choice2_Clicked(Event event) {
   String varName = choice2.getSelectedItem() ;
   choice3.removeAll() ;

   RuleVariable rvar =
   (RuleVariable)currentRuleBase.variableList.get(varName);
   Enumeration labels = rvar.labels.elements();
   while (labels.hasMoreElements()) {
      choice3.addItem(((String)labels.nextElement())) ;
   }
 }
// user selected a value for a variable
void choice3_Clicked(Event event) {
   String varName = choice2.getSelectedItem() ;
   String varValue = choice3.getSelectedItem() ;

   RuleVariable rvar =
      (RuleVariable)currentRuleBase.variableList.get(varName);
   rvar.setValue(varValue) ;
   textArea3.appendText("\n"+ rvar.name + " set to "+ varValue)
 }

// user pressed Find button -- do an infernece cycle
 void button1_Clicked(Event event) {
   String goal = textField1.getText() ;

   textArea2.appendText("\n --- Starting Inferencing Cycle --- \
   currentRuleBase.displayVariables(textArea2) ;
   if (radioButton1.getState() == true)
       currentRuleBase.forwardChain();
   if (radioButton2.getState() == true)
       currentRuleBase.backwardChain(goal);
   currentRuleBase.displayVariables(textArea2) ;
   textArea2.appendText("\n --- Ending Inferencing Cycle --- \n"
}
```

```
// user pressed Demo button -- do inference with pre-set values
void button2_Clicked(Event event) {
   String rbName = choice1.getSelectedItem() ;
   if (rbName.equals("Vehicles")) {
     if (radioButton1.getState() == true)
         demoVehiclesFC(vehicles);
     if (radioButton2.getState() == true)
         demoVehiclesBC(vehicles);
  } else if (rbName.equals("Bugs")) {
     if (radioButton1.getState() == true) demoBugsFC(bugs);
     if (radioButton2.getState() == true) demoBugsBC(bugs);
  } else {
     if (radioButton1.getState() == true) demoPlantsFC(plants);
     if (radioButton2.getState() == true) demoPlantsBC(plants);
  }
 }

// User press the Reset button
void button3_Clicked(Event event) {

       //{{CONNECTION
       // Clear the text for TextArea
       textArea1.setText("");
       textArea2.setText("");
       textArea3.setText("");
       //}}

    currentRuleBase.reset() ;
    currentRuleBase.displayRules(textArea1);
    currentRuleBase.displayVariables(textArea3) ;
 }

 public void init() {
    super.init();

       // Note, this code is generated by Visual Cafe'
       //{{INIT_CONTROLS
       setLayout(null);
       addNotify();
       resize(624,527);
       button1 = new java.awt.Button("Find Goal");
       button1.reshape(36,468,108,30);
       add(button1);
       button2 = new java.awt.Button("Run Demo");
       button2.reshape(228,468,108,30);
       add(button2);
       button3 = new java.awt.Button("Reset");
       button3.reshape(444,468,108,30);
```

```
add(button3);
textArea1 = new java.awt.TextArea();
textArea1.reshape(12,48,312,144);
add(textArea1);
textArea2 = new java.awt.TextArea();
textArea2.reshape(12,216,600,168);
add(textArea2);
label2 = new java.awt.Label("Trace Log");
label2.reshape(24,192,168,24);
add(label2);
label1 = new java.awt.Label("Rule Base");
label1.reshape(24,12,96,24);
add(label1);
choice1 = new java.awt.Choice();
add(choice1);
choice1.reshape(132,12,192,24);
Group1 = new CheckboxGroup();
radioButton1 = new java.awt.Checkbox("Forward Chain",
                                     Group1, false);
radioButton1.reshape(36,396,156,21);
add(radioButton1);
choice3 = new java.awt.Choice();
add(choice3);
choice3.reshape(480,36,135,24);
label5 = new java.awt.Label("Value");
label5.reshape(480,12,95,24);
add(label5);
choice2 = new java.awt.Choice();
add(choice2);
choice2.reshape(336,36,137,24);
textArea3 = new java.awt.TextArea();
textArea3.reshape(336,72,276,122);
add(textArea3);
label4 = new java.awt.Label("Variable");
label4.reshape(336,12,109,24);
add(label4);
radioButton2 = new java.awt.Checkbox("Backward Chain",
                                    Group1, false);
radioButton2.reshape(36,420,156,24);
add(radioButton2);
textField1 = new java.awt.TextField();
textField1.reshape(324,420,142,27);
add(textField1);
label3 = new java.awt.Label("Goal");
label3.reshape(324,384,80,30);
add(label3);
//}}

// initialize the rule applet
```

```
        frame = new Frame(“Ask User”) ;
        frame.resize(50,50) ;
        frame.setLocation(100,100) ;
        choice1.addItem(“Vehicles”) ;
        choice1.addItem(“Bugs”) ;
        choice1.addItem(“Plants”) ;

        vehicles = new RuleBase(“Vehicles Rule Base”) ;
        vehicles.setDisplay(textArea2) ;
        initVehiclesRuleBase(vehicles) ;
        currentRuleBase = vehicles ;

        bugs = new RuleBase(“Bugs Rule Base”) ;
        bugs.setDisplay(textArea2) ;
        initBugsRuleBase(bugs) ;

        plants = new RuleBase(“Plants Rule Base”) ;
        plants.setDisplay(textArea2) ;
        initPlantsRuleBase(plants) ;

    // initialize textAreas and list controls
    currentRuleBase.displayRules(textArea1) ;
    currentRuleBase.displayVariables(textArea3) ;
    radioButton1.setState(true) ;
    choice1_Clicked() ; // fill variable list
}

 // Note: this is Java 1.0.2 event model
 public boolean handleEvent(Event event) {
      if (event.target == button1 &&
          event.id == Event.ACTION_EVENT) {
           button1_Clicked(event);
           return true;
      }if (event.target == button2 &&
           event.id == Event.ACTION_EVENT) {
           button2_Clicked(event);
           return true;
      }if (event.target == button3 &&
           event.id == Event.ACTION_EVENT) {
           button3_Clicked(event);
           return true;
      }
      if (event.target == dlg &&
           event.id == Event.ACTION_EVENT) {
           return dlg.handleEvent(event);
      }
     if (event.target == choice1 &&
          event.id == Event.ACTION_EVENT) {
           choice1_Clicked();
```

```
                return true;
            }
            if (event.target == choice2 &&
               event.id == Event.ACTION_EVENT) {
                  choice2_Clicked(event);
                  return true;
            }
              if (event.target == choice3 &&
                   event.id == Event.ACTION_EVENT) {
                  choice3_Clicked(event);
                  return true;
            }
             return super.handleEvent(event);
        }

        // Note this code is generated by Visual Cafe'
        //{{DECLARE_CONTROLS
        java.awt.Button button1;
        java.awt.Button button2;
        java.awt.Button button3;
        java.awt.TextArea textArea1;
        java.awt.TextArea textArea2;
        java.awt.Label label2;
        java.awt.Label label1;
        java.awt.Choice choice1;
        java.awt.Checkbox radioButton1;
        CheckboxGroup Group1;
        java.awt.Choice choice3;
        java.awt.Label label5;
        java.awt.Choice choice2;
        java.awt.TextArea textArea3;
        java.awt.Label label4;
        java.awt.Checkbox radioButton2;
        java.awt.TextField textField1;
        java.awt.Label label3;
        //}}

        static Frame frame ;
        static RuleVarDialog dlg ;
        static RuleBase bugs ;
        static RuleBase plants ;
        static RuleBase vehicles ;
        static RuleBase currentRuleBase ;
....
// Rule base definitions

}
```

The *RuleApplet.waitForAnswer()* method is called by the **RuleVariable** method, ask*User()*, during backward chaining in cases where a value for the variable could not be determined through inferencing. An instance of the **RuleVarDialog** is created as a modal dialog (Figure 4.3). The *prompt* **String**, along with a list of the valid values, and a single **TextField** are displayed in a small dialog. When the user enters a value for the variable and clicks on the **Set** pushbutton, the variable value is returned and the dialog is closed.

```
// display dialog to get user value for a variable
static public String waitForAnswer(String prompt, String labels) {

     // position dialog over parent dialog
    Point p = frame.getLocation() ;
    dlg = new RuleVarDialog(frame, true) ;
     dlg.label1.setText("   " + prompt + "\n  (" + labels + ")");
    dlg.setLocation(400, 250) ;
    dlg.show() ;
    String ans = dlg.getText() ;
    return ans ;
}
```

Figure 4.3 The RuleVar dialog.

The Vehicles Rule Base Implementation The vehicles rule base is defined using the *RuleApplet.initVehiclesRuleBase()* method. The **RuleBase** instance is passed in as the only argument. The method first instantiates the *goalClauseStack* and *variableList* members. Each **RuleVariable** is then defined by creating an instance with the variable name as the argument on the constructor. The valid values are set using the *setLabels()* method. The prompt text, used in the **RuleVarDialog** during backward chaining, is set using the *setPromptText()* method. Each **RuleVariable** is also added directly to the **RuleBase** *variableList*.

After all the variables are defined, a few condition instances are created for equals, not equals, and less than. The *ruleList* member is instantiated, and each **Rule** in the **RuleBase** is instantiated. The **Rule** constructors take a **RuleBase**

reference, the rule *name* **String**, and two or more clauses. The last clause is the consequent clause and all prior clauses are antecedent clauses.

```
// initialize the Vehicles rule base
 public void initVehiclesRuleBase(RuleBase rb) {
  rb.goalClauseStack = new Stack() ; // goals and subgoals

  rb.variableList = new Hashtable() ;
  RuleVariable vehicle = new RuleVariable(“vehicle”) ;
  vehicle.setLabels(“Bicycle Tricycle MotorCycle Sports_Car Sedan Mi
Sports_Utility_Vehicle”) ;
  vehicle.setPromptText(“What kind of vehicle is it?”);

  rb.variableList.put(vehicle.name,vehicle) ;
  RuleVariable vehicleType = new RuleVariable(“vehicleType”) ;
  vehicleType.setLabels(“cycle automobile”) ;
  vehicleType.setPromptText(“What type of vehicle is it?”) ;
  rb.variableList.put(vehicleType.name, vehicleType) ;

  RuleVariable size = new RuleVariable(“size”) ;
  size.setLabels(“small medium large”) ;
  size.setPromptText(“What size is the vehicle?”) ;
  rb.variableList.put(size.name,size) ;

  RuleVariable motor = new RuleVariable(“motor”) ;
  motor.setLabels(“yes no”) ;
  motor.setPromptText(“Does the vehicle have a motor?”) ;
  rb.variableList.put(motor.name,motor) ;

  RuleVariable num_wheels = new RuleVariable(“num_wheels”) ;
  num_wheels.setLabels(“2 3 4”) ;
  num_wheels.setPromptText(“How many wheels does it have?”);
  rb.variableList.put(num_wheels.name,num_wheels) ;

  RuleVariable num_doors = new RuleVariable(“num_doors”) ;
  num_doors.setLabels(“2 3 4”) ;
  num_doors.setPromptText(“How many doors does it have?”) ;
  rb.variableList.put(num_doors.name,num_doors) ;

  // Note: at this point all variables values are NULL

 Condition cEquals = new Condition(“=”) ;
 Condition cNotEquals = new Condition(“!=”) ;
 Condition cLessThan = new Condition(“<”) ;

 // define rules
 rb.ruleList = new Vector() ;
 Rule Bicycle = new Rule(rb, “bicycle”,
      new Clause(vehicleType,cEquals, “cycle”)  ,
```

```
     new Clause(num_wheels,cEquals, “2”),
     new Clause(motor, cEquals, “no”),
     new Clause(vehicle, cEquals, “Bicycle”)) ;

Rule Tricycle = new Rule(rb, “tricycle”,
     new Clause(vehicleType,cEquals, “cycle”)  ,
     new Clause(num_wheels,cEquals, “3”),
     new Clause(motor, cEquals, “no”),
     new Clause(vehicle, cEquals, “Tricycle”)) ;

Rule Motorcycle = new Rule(rb, “motorcycle”,
     new Clause(vehicleType,cEquals, “cycle”)  ,
     new Clause(num_wheels,cEquals, “2”),
     new Clause(motor,cEquals, “yes”),
     new Clause(vehicle,cEquals, “Motorcycle”)) ;

Rule SportsCar = new Rule(rb, “sportsCar”,
     new Clause(vehicleType,cEquals, “automobile”)  ,
     new Clause(size,cEquals, “small”),
     new Clause(num_doors,cEquals, “2”),
     new Clause(vehicle,cEquals, “Sports_Car”)) ;

Rule Sedan = new Rule(rb, “sedan”,
     new Clause(vehicleType,cEquals, “automobile”)  ,
     new Clause(size,cEquals, “medium”),
     new Clause(num_doors,cEquals, “4”),
     new Clause(vehicle,cEquals, “Sedan”)) ;

Rule MiniVan = new Rule(rb, “miniVan”,
     new Clause(vehicleType,cEquals, “automobile”)  ,
     new Clause(size,cEquals, “medium”),
     new Clause(num_doors,cEquals, “3”),
     new Clause(vehicle,cEquals, “MiniVan”)) ;

Rule SUV = new Rule(rb, “SUV”,
     new Clause(vehicleType,cEquals, “automobile”)  ,
     new Clause(size,cEquals, “large”),
     new Clause(num_doors,cEquals, “4”),
     new Clause(vehicle,cEquals, “Sports_Utility_Vehicle”)) ;

Rule Cycle = new Rule(rb, “Cycle”,
     new Clause(num_wheels,cLessThan, “4”)  ,
     new Clause(vehicleType,cEquals, “cycle”)) ;

Rule Automobile = new Rule(rb, “Automobile”,
     new Clause(num_wheels,cEquals, “4”)  ,
     new Clause(motor,cEquals, “yes”),
     new Clause(vehicleType,cEquals, “automobile”)) ;

}
```

The *RuleApplet.demoVehiclesFC()* method is called when the vehicles rule base and the forward chaining radio button are selected and the **Run Demo** pushbutton is clicked. The *vehicle* and *vehicleType* variables are set to null, and the other variables are set to indicate that the *vehicle* is a MiniVan. The variable value settings are displayed in the trace panel, the *RuleBase.forwardChain()* method is called, and the resulting variable values are displayed in the trace panel.

```
public void demoVehiclesFC(RuleBase rb) {

 textArea2.appendText("\n --- Starting Demo ForwardChain ---\n ") ;
  // should be a Mini-Van
 ((RuleVariable)rb.variableList.get("vehicle")).setValue(null) ;
 ((RuleVariable)rb.variableList.get("vehicleType")).setValue(null) ;
 ((RuleVariable)rb.variableList.get("size")).setValue("medium") ;
 ((RuleVariable)rb.variableList.get("num_wheels")).setValue("4") ;
 ((RuleVariable)rb.variableList.get("num_doors")).setValue("3") ;
 ((RuleVariable)rb.variableList.get("motor")).setValue("yes") ;
  rb.displayVariables(textArea2) ;
  rb.forwardChain() ;  // chain until quiescence...
  textArea2.appendText("\n --- Stopping Demo ForwardChain! ---\n") ;

  rb.displayVariables(textArea2);
 }
```

The *RuleApplet.demoVehiclesBC()* method is called when the vehicles rule base and the backward chaining radio button are selected and the **Run Demo** pushbutton is clicked. The *vehicle* and *vehicleType* variables are set to null, and the other variables are set to indicate that the *vehicle* is a MiniVan. The variable value settings are displayed in the trace panel, the *RuleBase.backwardChain()* method is called, and the resulting variable values are displayed in the trace panel.

```
public void demoVehiclesBC(RuleBase rb) {
 textArea2.appendText("\n --- Starting Demo BackwardChain ---\n ") ;
  // should be a minivan
 ((RuleVariable)rb.variableList.get("vehicle")).setValue(null) ;
 ((RuleVariable)rb.variableList.get("vehicleType")).setValue(null) ;
 ((RuleVariable)rb.variableList.get("size")).setValue("medium") ;
 ((RuleVariable)rb.variableList.get("num_wheels")).setValue("4") ;
 ((RuleVariable)rb.variableList.get("num_doors")).setValue("3") ;
 ((RuleVariable)rb.variableList.get("motor")).setValue("yes") ;
  rb.displayVariables(textArea2) ;
  rb.backwardChain("vehicle") ;  // chain until quiescence...
  textArea2.appendText("\n --- Stopping Demo BackwardChain! ---\n ")
  rb.displayVariables(textArea2) ;
}
```

Implementation of the other two rule bases provided with the **Rule Applet**, the bugs and plants rule bases, is listed in Appendix A.

That brings to a close our discussion of the Rule applet. All of the code described in this chapter is included on the CD-Rom. The applet and the underlying **RuleBase**, **Rule**, and **RuleVariable** classes provide basic functionality. Some obvious enhancements come to mind. These include support for **Objects** on both the left-hand and right-hand side of **Clauses**. In the **Rule** class, it would be nice to provide support for sensors and effectors (which could be useful in Part 2 of this book). Support for multiple instances of variables and time-stamping of data would greatly extend the forward-chaining capabilities. Adding rule priorities would also be useful for larger rule bases. Adding enhanced tracing and how and why support for the backward-chaining algorithm would be a nice touch. Although there is much more we could do, it is important to recognize that even this level of functionality is capable of producing useful behavior in applications, or as we will see, in intelligent agents.

In the remainder of this chapter, we change our focus from implementation to a more high-level discussion of fuzzy rule systems and then planning. Fuzzy rule systems are used to perform a kind of forward chaining, but the underlying logic system is based on fuzzy logic, not boolean logic. Planning is an interesting topic, because for anything other than simple tasks, our intelligent agents will need to build plans and execute them in order to perform useful work for us.

Fuzzy Rule Systems

In the rich history of rule-based reasoning in AI, the inference engines almost without exception were based on Boolean or binary logic. However, in the same way that neural networks have enriched the AI landscape by providing an alternative to symbol processing techniques, fuzzy logic has provided an alternative to Boolean logic–based systems (Bigus 1996).

Unlike Boolean logic, which has only two states, true or false, fuzzy logic deals with truth values which range continuously from 0 to 1. Thus something could be *half true* 0.5 or *very likely true* 0.9 or *probably not true* 0.1. The use of fuzzy logic in reasoning systems impacts not only the inference engine but the knowledge representation itself (Zadeh 1994). For, instead of making arbitrary distinctions between variables and states, as is required with Boolean logic systems, fuzzy logic allows one to express knowledge in a rule format that is close to a natural language expression. For example, we could say:

if temperature is hot and humidity is sticky then fan_speed is high

The difference between this fuzzy rule and the Boolean-logic rules we used in our forward- and backward-chaining examples is that the clauses “temperature is hot” and “humidity is sticky” are not strictly true or false. Clauses in fuzzy rules are real-valued functions called membership functions that map the fuzzy set “hot” onto the domain of the fuzzy variable “temperature” and produce a truth-value that ranges from 0.0 to 1.0 (a continuous output value, much like neural networks).

Reasoning with fuzzy rule systems is a forward-chaining procedure. The initial numeric data values are *fuzzified*, that is, turned into fuzzy values using the membership functions. Instead of a match and conflict resolution phase where we select a triggered rule to fire, in fuzzy systems, all rules are evaluated, because all fuzzy rules can be true to some degree (ranging from 0.0 to 1.0). The

antecedent clause truth values are combined using fuzzy logic operators (a fuzzy conjunction or **and** operation takes the minimum value of the two fuzzy clauses). Next, the fuzzy sets specified in the consequent clauses of all rules are combined, using the rule truth values as scaling factors. The result is a single fuzzy set, which is then *defuzzified* to return a crisp output value.

Planning

Planning is one of the most complex problems which AI has attempted to solve. However, it is one of the most useful in that most nontrivial problems require some ordered sequence of operations and a variety of techniques. Planning involves several aspects of problem solving. First is the decomposition of a big problem into smaller, more easily solved subproblems. Second is taking account of constraints on the order of solution of the subproblems so that we do not undo work we did in the previous step. Last is the challenge of keeping track of the state of the world as we progress toward a solution (Russell and Norvig 1995).

An important point to remember is that planning is not search. We are not simply trying to traverse a graph in search of a single goal node or to find a path to a goal node. In planning, we compute several steps of a problem-solving approach without executing them. Also, it is sometimes not possible to start at the beginning and proceed linearly to a solution. There may be discontinuities or breaks in our plan which we cannot overcome. For example, how do we get from A to B when there is no path between those points? In order to plan, we must have an internal model of the world so that we can propose operations and predict what the outcome will be, and whether it will bring us closer to the goal or not.

Similar to any search algorithm, a planning system must be able to select the best action or operation to perform for a given situation. It must be able to model or predict what the consequence of taking that action will be. It must detect when it has reached the goal, or realize when it has reached a dead end and needs to try a new plan of attack.

Perhaps the biggest difference between a planning problem and a simple search problem is that the state space is so large that you cannot carry the complete state along with you at each decision node. The sheer size of the space makes us have to only keep track of the things that explicitly change due to some action we take. This problem is called the *frame problem* in artificial intelligence literature.

There are three major approaches to planning that we will discuss briefly. First is goal stack planning, which is a linear problem-solving technique, next is a nonlinear method, and finally we'll look at a hierarchical planning algorithm.

Goal stack planning was an early attempt at solving problems that could be broken up into smaller problems, and where there were constraints or interactions between steps in the plan. The research problem focused on a blocks world, where the goal was simply to arrange the blocks in a specified order. While seemingly trivial (any two-year-old can do the task), this toy problem illustrated essential requirements for successful planning strategies.

Given an initial configuration of blocks stacked on each other, or lying on a table, we specify a desired goal configuration. We have a defined set of operators for moving blocks, either to pick them up, move them, or stack them on other blocks. Our plan is to develop the sequence of operators which goes from our initial state to our goal state. Goal stack planning, as the name implies, uses a stack to hold subgoals or intermediate goals we solve as subproblems. A major problem with goal stack planning is that we can expend a lot of effort solving parts of the problem while making other parts more difficult, if not impossible, to solve. The major weakness is that goal stack planning tries to solve the problem and subproblems linearly, in order, and does not take into account how the sequence of actions can impact progress toward the ultimate goal.

Several nonlinear techniques were developed in order to avoid the problems with linear planning techniques such as goal stack planning. Nonlinear planning approaches can account for the interactions between operators and subgoals. They construct a plan where two or more subgoals can proceed simultaneously. Famous nonlinear AI planning systems include NOAH (Sacerdoti 1974) and MOLGEN (Stefik 1981). A technique called constraint posting allows nonlinear plans to be constructed incrementally, with a set of operators with incompletely specified order and variable bindings. Rather than apply a single operator and

then update the space as in state-space planning or search, constraint posting allows multiple operators and variable bindings to be specified in a single node.

Means-ends analysis is used to select operations along the way. Each operator has a set of preconditions which must hold in order for the operator to be applicable, and a set of postconditions which specify the changes to the state once the operator is applied. This pre/post condition list gets around the frame problem because it explicitly states what subset of conditions must hold and what conditions will be changed.

The third major class of planning algorithms, hierarchical planning, allows plans to proceed at higher levels of abstraction in order to sketch out feasible solutions without going immediately into the details. The ABSTRIPS system (Sacerdoti 1974) used criticality values to rank the importance of various preconditions on operators. The idea is to first solve the entire planning problem taking into account only those at the highest levels of criticality, and then to proceed to finer and finer granularity by adding additional operators to satisfy the preconditions at the lower levels. While the proper setting of the criticality values is crucial, hierarchical planners have proven to be one of the most practical techniques for planning systems.

Summary

In this chapter we focused on reasoning techniques used with if-then rules. The main points include:

- *If-then rules* are the most successful form of knowledge representation. They are easy for people to create and understand.
- A rule has two parts, an *if* or *antecedent* part, and the *then* or *consequent* part. A condition in the rule is called a *clause*. If all of the antecedent clauses of a rule are true, it is *triggered* to fire. When a rule *fires*, its consequent clause is made true and a new fact is added to the knowledge base.
- *Certainty* or *confidence factors* are used as rule modifiers in situations where either the validity of the data or the applicability of the rule is uncertain.
- Reasoning systems are either *monotonic*, meaning they only ever add new facts to the working memory, or *nonmonotonic*, where they can retract facts when provided with new evidence. Nonmonotonic reasoning systems must

deal with *truth maintenance*, and track dependencies between facts in the working memory.

- *Forward chaining* is a data-driven inferencing method using rules. Starting with an initial set of facts in *working memory*, the *match* phase selects the *conflict set* of rules that are ready to fire. *Conflict resolution* selects a single rule to fire. The *act* phase fires the rule and adds a new fact to the working memory. This process is repeated until the conflict set is empty.
- *Backward chaining* is a goal-driven inferencing method using rules. Starting with a goal variable or clause, the algorithm chains through the rules from consequent clauses to antecedent clauses, trying to prove the rule true.
- We implemented a *Rule Applet* that uses *RuleBase*, *Rule*, and *RuleVariable* classes to perform forward and backward chaining in Java.
- *Fuzzy systems* are rule systems which use fuzzy logic rather than boolean logic to make decisions.
- *Planning* is a fundamental requirement for intelligent agents. Planning is difficult because there are sometimes interactions between subproblems or pieces of the plan. Three major types of planning algorithms have been studied. These include *linear*, *nonlinear*, and *hierarchical*.

Exercises

1. Using the Vehicles rule base, manually walk through a backward-chaining inference cycle with the goal of *vehicles = sedan*. How many times did you have to ask the user to provide a value for an unknown variable while inferencing?

2. Open your Java-enabled Web browser or applet viewer on the *RuleApplet.html* file. Select the Vehicles rule base and set the variables so that the vehicle is recognized as a sedan. Do this using forward chaining and backward chaining. What were the differences between the two? For backward chaining, do the trace results compare to exercise 4.1?

3. How could you extend the **Clause** class to support sensors and effectors? What other classes would have to be modified or extended to support these functions?

Chapter 5
Learning Systems

In Chapter 5, we focus on adaptive software and machine learning techniques. We start with an overview of the different paradigms for performing machine learning, including supervised, unsupervised, and reinforcement learning. We provide a general introduction to neural networks and then a more detailed discussion of back propagation and Kohonen maps. Next, we explore the use of information theory to construct decision trees from data. We design and implement a Java applet and the corresponding Java classes for back propagation, Kohonen maps, and decision tree algorithms. Last, we look at classifier systems, which add learning to rule-based systems using genetic algorithms.

Overview

One central element of intelligent behavior is the ability to adapt or learn from experience. For all of the sophisticated knowledge representation and reasoning algorithms that we develop, there is no way that we can know *a priori* all of the situations that our intelligent agent will encounter. Thus, being able to adapt to changes in the environment or to get better at tasks through experience becomes a significant differentiator for any software system. Any agent that can learn has an advantage from one that cannot. Adding learning or adaptive behavior to an intelligent agent elevates it to a higher level of ability. A learning agent can adapt to your likes and dislikes. It can learn which agents to trust and cooperate with, and which ones to avoid. A learning agent can recognize situations it has been in before and improve its performance based on prior experience.

There are many forms of learning. First, there is rote learning, where an example is given and the student (intelligent agent) copies the example and exactly reproduces the behavior. While rote learning is a simple form of learning it still can be powerful. For example, we could run a simulation 24 hours a day, 7 days a week, to generate new situations and the desired behavior, then present these examples to our agent. Our agent would be better able to respond to situations

one week after we started the training and would be even better one month later (assuming the additional knowledge didn't slow down its response times).

Another form of learning is parameter or weight adjustment. In this case, we may know *a priori* what factors are important in some decision, but we do not know how to weight their contribution to the answer. In this case, we can adjust the weighting factors over time so that we improve the likelihood of a correct decision or output. This technique is the basis for neural network learning.

Induction is a process of learning by example where we try to extract the important characteristics of the problem thereby allowing us to generalize to novel situations or inputs. Decision trees and neural networks both perform induction and can be used for classification or regression (prediction) problems. The key aspect of inductive methods is that the examples are processed and automatically transformed into an internal form (knowledge representation) which captures the essence of the problem.

Another type of learning is called clustering, chunking, or abstraction of knowledge. While people learn from very specific examples or situations, the ability to detect common patterns and generalize to new situations is a type of learning. By chunking ten cases into one more general case, we cut down on the amount of storage we need and also on the search or processing time. By thinking at higher or more abstract levels, we can think "great thoughts" without getting caught in the muddle of a million little details.

Clustering is another learning algorithm, which is a type of chunking. Clustering algorithms look at high-dimensional data (data with many attributes) and score them for similarity based on some criterion. The result is that each sample is assigned to a cluster or group with other examples deemed to be "similar." This similarity could be used as a way of assigning meaning to that group of samples. For example, clustering data of business customers may result in four distinct clusters. Examination of the customers who fell into each cluster may show that they are being grouped based on their interests (types of product they buy) or by their purchasing patterns (number of visits, sales totals, profitability) or some other criterion which may not have been apparent when looking at the original data. Another example would be clustering documents we found particularly useful. This could provide valuable information to improve the performance of a document search engine the next time we make a query.

All of these learning techniques, induction for classification and prediction, and clustering are used in data mining tools. Data mining is a process of extracting valuable, nonobvious information from large collections of data (Bigus 1996). The main contribution of data mining to is find patterns which were not known to exist, that is, to discover new information or knowledge (some people refer to data mining as *knowledge discovery*). So learning, as applied to data mining, can be thought of as a way for intelligent agents to automatically discover knowledge rather than having it predefined using predicate logic, rules, or some other representation.

Learning Paradigms

There are several major paradigms, or approaches, to machine learning. These include supervised, unsupervised, and reinforcement learning. In addition, many researchers and application developers combine two or more of these learning approaches into one system. How the training data is processed is a major aspect of these learning paradigms.

Supervised learning is the most common form of learning and is sometimes called *programming by example*. The learning agent is trained by showing it examples of the problem state or attributes along with the desired output or action. The learning agent makes a prediction based on the inputs and if the output differs from the desired output, then the agent is adjusted or adapted to produce the correct output. This process is repeated over and over until the agent learns to make accurate classifications or predictions. Supervised learning is the most common type of machine learning. Historical data from databases, sensor logs, or trace logs is often used as the training or example data. We provide two implementations of supervised learning algorithms later in this chapter: the back propagation neural network, and a decision tree.

Unsupervised learning is used when the learning agent needs to recognize similarities between inputs or to identify features in the input data. The data is presented to the agent, and it adapts so that it partitions the data into groups. The clustering or segmenting process continues until the agent places the same data into the same group on successive passes over the data. An unsupervised learning algorithm performs a type of feature detection where important common attributes in the data are extracted. We present a neural network implementation of an unsupervised learning technique called a Kohonen map.

Reinforcement learning is a type of supervised learning where the error information is less specific. It can also be used in cases where there is a sequence of inputs and the output or action is only taken after the specific sequence occurs. Because we provide less specific error information, reinforcement learning usually takes longer than supervised learning and is less efficient. However, in many situations, having exact prior information about the desired outcome is not possible. In many ways, reinforcement learning is the most realistic form of learning.

Another important distinction in learning agents is whether the learning is done on-line or off-line. On-line learning means that the agent is sent out to perform its tasks and that it can learn or adapt after each transaction is processed. On-line learning is like on-the-job training and places severe requirements on the learning algorithms. It must be very fast and very stable (we don't want a brain-dead agent in the middle of a big sales negotiation). Off-line learning, on the other hand, is more like a business seminar. You take your salespeople off the floor and place them in an environment where they can focus on improving their skills without distractions. After a suitable training period, they are sent out to apply their newfound knowledge and skills. In an intelligent agent context, this means that we would gather data from situations that the agents have experienced. We could then augment this data with information about the desired agent response to build a training data set. Once we have this database we can use it to modify the behavior of our agents.

Previous | Table of Contents | Next

Neural Networks

Neural networks provide an easy way to add learning ability to agents. There are many different types of neural networks, some which train quickly and some which require many passes over the data before they learn the assigned tasks. Neural networks can be used in supervised, unsupervised, and reinforcement learning scenarios. They can be used for classification, clustering, and prediction. In many applications, neural networks can replace expert systems, especially where sufficient data is available for training.

Not simply another learning algorithm or regression technique, neural networks actually represent a new and different computing model from the serial von Neumann computers we all know and love (and use every day). Borrowing heavily from the metaphor of the human brain, neural networks have hundreds or thousands of simple processors (called processing units, processing elements, or neurons) connected by hundreds or thousands of adaptive weights as illustrated in Figure 5.1. This network of processors and connections forms a parallel computer that can adaptively rewire itself when it is exposed to data. The adaptive weights are adjusted and form the memory of the neural network computer, playing the role that the synapse has in the human brain. Another difference between neural networks and traditional digital computers is that neural network processors are analog, not digital. They don't spit out binary 1s and 0s, they produce a continuous range of outputs, usually from 0.0 to 1.0. The infinite number of real values between 0 and 1 can be used to our advantage.

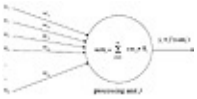


Figure 5.1 A neural processing unit.

The new parallel computing model notwithstanding, most neural network implementations today are simply programs running on serial computers. We can simulate the behavior of the neural network computer on our extremely flexible and powerful (and cheap) PCs and workstations. And while they could do wondrous things if implemented in parallel hardware, neural networks have proven useful in many commercial applications even when simulated on standard computers.

Back Propagation

Back propagation is the most popular neural network architecture for supervised learning. It features a feedforward connection topology, meaning that data flows through the network in a single direction, and uses a technique called the *backward propagation of errors* to adjust the connection weights (Rumelhart, Hinton, and Williams 1986). In addition to a layer of input and output units, a back-propagation network can have one or more layers of hidden units, which receive inputs only from other units, not the external environment. A back-propagation network with a single hidden layer of processing units can learn to model any continuous function when given enough units in the hidden layer. The primary applications of back-propagation networks are for prediction and classification.

Figure 5.2 shows a diagram of a back-propagation neural network and illustrates the three major steps in the training process. First, input data is presented to the input layer of units on the left, and flows through the network until it reaches the network output units on the right. This is called the forward pass. The *activations* or values of the output units represent the actual or predicted output of the network. The desired output value is also presented to the network, because this is supervised learning. Next, the difference between the desired and actual output is computed, producing the network error. This error term is then passed backwards through the network to adjust the connection weights. This process is described in gory detail later in this section.

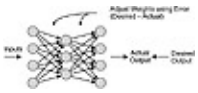


Figure 5.2 Back propagation neural network.

Each network input unit takes a single numeric value, x_i, which is usually scaled or normalized to a value between 0.0 and 1.0. This value becomes the input unit activation. Next, we need to propagate the data forward, through the neural network. For each unit in the hidden layer, we compute the sum of the products of the input unit activations and the weights connecting those input layer units to the hidden layer. This sum is the inner product (also called the dot or scalar product) of the input vector and the weights into the hidden unit. Once this sum is computed, we add a threshold value and then pass this sum through a nonlinear activation function, **f**, producing the unit activation y_j as shown in

Figure 5.1. The formula for computing the activation of any unit in a hidden or output layer in the network is:

where *i* ranges over all units leading into unit *j*, and the activation function is

$$\mathbf{f}(\text{sum}_j) = 1 / 1 + e^{-\text{sum}j}.$$

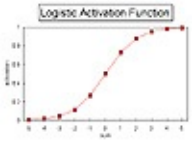


Figure 5.3 The logistic activation function.

As mentioned earlier, we use the S-shaped sigmoid or logistic function for **f**. As shown in Figure 5.3, for a range of sums from –5 to +5 we get a *y* value ranging from 0 to 1. Extremely large positive or negative sums get squashed (that's a technical term) into that same range. The effect of the threshold q is to shift the S-shaped curve left or right. For example, if the threshold is +5, then the sum would have to be -5 in order to produce an output value of 0.5.

The formula for calculating the changes to the weights is

where w_{ij} is the weight connecting unit *i* to unit *j*, η is the learn rate parameter, δ_j is the error signal for that unit, and y_i is the output or activation value of unit *i*. For units in the output layer, the error signal is the difference between the target output t_j and the actual output y_j multiplied by the derivative of the logistic activation function.

$$\delta_j = (t_j - y_j)\, f'_j(\text{sum}_j) = (t_j - y_j)\, y_j\, (1 - y_j)$$

For each unit in the hidden layer, the error signal is the derivative of the activation function multiplied by the sum of the products of the outgoing connection weights and their corresponding error signals. So for hidden unit *j*

where *k* ranges over the indices of the units receiving unit *j*'s output signal.

A common modification of the weight update rule is the use of a momentum term a, to cut down on oscillation of the weights. So, the weight change becomes a combination of the current weight change, computed as before, plus some fraction (α ranges from 0 to 1) of the previous weight change. This complicates implementation because we now have to store the weight changes from the prior step.

The mathematical basis for backward propagation is described in detail in Rumelhart, Hinton, and Williams (1986). When the weight changes are summed up (or batched) over an entire presentation of the training set, the error minimization function performed is called gradient descent. In practice, most people immediately update the network weights after each input vector is presented. While pattern updates, as this is called, can sometimes produce undesirable behavior, it usually results in faster training than using the batch updates.

Kohonen Maps

The self-organizing feature maps developed by Tuevo Kohonen have become one of the most popular and practical neural network models (Kohonen 1990). A Kohonen map is a single-layer neural network, comprised of an input layer and an output layer. Unlike back propagation, which is a supervised learning paradigm, feature maps perform unsupervised learning. Each time an input vector is presented to the network, its distance to each unit in the output layer is computed. Various distance measures have been used. The most common and the one used here is just the Euclidean distance. The output unit with the smallest distance to the input vector is declared the "winner." The winning unit and a set of units in the neighborhood weights are adjusted by moving the weights toward the input vector. Initially, the neighborhood and the learning rate are quite large and so many units are moved around in the input space to match the set of training vectors. As training progresses, the neighborhood shrinks and the learning rate is decreased. The Kohonen map self-organizes over time, and at the end of a successful training run, a topographic map is created. One attribute of such a map is that inputs which are near each other in the input space map onto output units which are in close proximity in the output layer of the neural

network.

In Figure 5.4, we show a schematic diagram of a Kohonen map. First the inputs are presented to the input layer. Second, the distance of the input pattern to the weights to each output unit is computed using the Euclidean distance formula below:

$$y_j = \| \mathbf{x} - \mathbf{w}_j \|^2$$

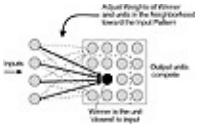


Figure 5.4 Kohonen map neural network.

where **x** is the input vector, w_j is the weight vector into output unit j, and y_j is the resulting distance. The output unit j with the minimum value y_j is declared the winner. The weights of the winner and the units in its neighborhood are then adjusted using:

where w_j is the weight vector into unit j at previous time t and current time $t + 1$, $\alpha(k)$ is the learn rate at iteration k, $C_{ij}(k)$ is the value of the neighborhood function for units i and j at iteration k, and y_j is the Euclidean distance between input vector **x** and weight vector w_j at time t. This neighborhood function $C_{ij}(k)$ is called a Gaussian function and is shaped like a Mexican hat or sombrero and is defined as:

where i and j are the coordinates of the units in the two-dimensional map, and k is the iteration number. $\sigma(k)^2$ is the width of the neighborhood function, which starts out as wide as the map when k is small and decreases to a final value encompassing a single unit when k is at its maximum value. The $\alpha(k)$ parameter is the learn rate for iteration k. This is computed as:

where the k_{max} term is the maximum number of iterations to be performed. The

learn rate $\alpha(k)$ exponentially decreases as the iteration number k gets larger. We start at iteration $k = 0$ with the $\alpha_{initial}$ learn rate and end at iteration $k = k_{max}$ with the α_{final} value. Typical values for $\alpha_{initial}$ and α_{final} are 1.0 and 0.05 respectively. These formulas are used as the basis for our Java implementation, described later in this chapter.

Decision Trees

Decision trees perform induction on example data sets, generating classifiers and prediction models. A decision tree examines the data set, and uses information theory to determine which attribute contains the most information on which to base a decision. This attribute is then used in a decision node to split the data set into two groups, based on the value of that attribute. At each subsequent decision node, the data set is split again. The result is a decision tree, a collection of nodes. The leaf nodes represent a final classification of the record. Examples of decision tree algorithms are ID3 (Quinlan 1986) and the C4.5 systems. A generic term for the class of decision tree algorithms is CART, for *C*lassification *A*nd *R*egression *T*rees.

Information Theory

Decision trees are based on information theory, a mathematical concept first introduced by Shannon and Weaver (1949). The unit of information is a bit, and the amount of information in a single binary answer is $\log_2 P(v)$, where $P(v)$ is the probability of event v occurring. The *information* content is based on the prior probabilities of getting the correct answer to a question or classification (Russell and Norvig 1995). Suppose we are trying to classify customers who will either renew their Internet service accounts or will cancel and switch to a competitor. Let's assume we had 1000 customers last year, 800 that renewed and 200 that canceled. We would have a training set with 1000 records, containing some set of attributes of those customers (their age, sex, income, average monthly connect time, years as customers, etc.) as well as the information on whether they renewed or canceled their service. Those who renewed we consider positive examples, p. Those who canceled we consider negative examples, n. If we had information on a customer and had to try to predict whether that customer would renew or cancel, how much information would be contained in a correct answer? The formula used is

Notice that the denominator in all terms is the total number of records. The first term is the probability that the positive case occurs (p / (p + n)) multiplied by the information content of that event $\log_2$ (p / (p + n)). The second term is an equivalent expression applied to the negative examples. In our Internet service example, the chance that any single factor (for example, age) would completely divide the group into those who would renew or cancel is small. We need to measure how much information we still need after the test. Any attribute A which has v distinct values divides the data set into v subsets according to the values of A. Each resulting subset of the training data has its own makeup of p and n outcomes. On average, after testing attribute A, we still need

bits of information where i goes from 1 to v, the number of discrete values that attribute A can take. The difference between the information needed before the attribute test and the remainder is called the *information gain* of the test.

$$\text{Gain}(A) = I\ (p\ /\ (p + n),\ n\ /\ (p + n) - \text{Remainder}\ (A)$$

As an example, suppose that men renew 90 percent of the time, and women renew 70 percent, and that our customer set is made up half of men and half of women. How much information gain would we get simply by testing whether a customer is male or female?

```
Gain(Sex) = 1 - [(500/1000) I(450/500, 50/500) + (500/1000) I (350/5
          = 1 - [(.5) I(.9, .1) + (.5) I (.7, .3)]
          = 1 - [(.5) 0.468996 + (.5) 0.881291]
          = 0.324857
```

Suppose that we had grouped the customers' usage habits into 3 groups: under 4 hours a month, from 4 to 10 hours, and over 10. Assume also that they were evenly split between all customers. When we look at their renewal rates, we see that the first group renews at 50 percent, the second at 90 percent, and the third at 100 percent. What information would we gain by testing on this attribute?

```
Gain(Usage) = 1 - [(.333) I(166/333, 166/333) + (.333) I (300/333, 3
            = 1 - [(.333) I(.5, .5) + (.333) I (.9, .1) + (.333) I(1
            = 1 - [(.333) 1.0 + (.333) 0.466133 + (.333) 0.0]
            = 0.511778
```

You can see that in this example, the second case gives us more information gain than the first. So, if we were building a decision tree, we would want to first split the data based on how much connect-time they used, and then on whether the customer was male or female. This process of splitting the data, based on the attribute-value pair containing the most information, is continued until we can classify the data to our desired degree of accuracy.

Learn Applet

In this section we describe an applet which demonstrates three learning techniques. These include two types of neural networks and a tree classifier. Before we get into the details of the implementations of the learning algorithms, we need to introduce some helper classes. Figure 5.5 shows the layout of our learning applet. It features two text display areas, one for the data used to train the neural networks or tree classifier, and one for trace information from the models themselves.

Figure 5.5 The Learn applet.

Before we get into the implementation details of the Learn applet, we need to introduce two subclasses of the Variable class for use in the learning algorithms. These classes deal specifically with discrete and continuous variables and support the behavior required for data normalization required by the neural network algorithms.

Continuous Variables

The **ContinuousVariable** class is a subclass of **Variable**. It provides the support necessary for variables which can take on a continuous real value ranging from some predefined minimum to a maximum value. The constructor takes the name of the variable as the only parameter. The *min* and *max* members can be set using the *setMin()* and *setMax()* methods directly, or they can be computed automatically if the **ContinuousVariable** is used as part of a **DataSet** by calling the *computeStatistics()* method. The *normalize()* method is used by the **DataSet** class when it is creating an all-numeric version of a data set. This method does a simple linear scaling of the input value to a value in the range of 0.0 to 1.0. The **ContinuousVariable** class inherits the *normalizedSize()* method from the **Variable** class, because the normalized size is always 1.

```
class ContinuousVariable extends Variable {

 float min = (float)0.0;
 float max = (float)0.0;

 ContinuousVariable(String name) { super(name); }
 void setMin(float Min) { min = Min; }
 void setMax(float Max) { max = Max; }

 public void computeStatistics(String inValue){
    float val = new Float(inValue).floatValue();
    if (val < min) min = val ;
    if (val > max) max = val ;
 }

 // scale the inValue to 0.0 and 1.0
 public int normalize(String inStrValue, float[] outArray, int inx)
    float outValue ;
    float inValue = Float.valueOf(inStrValue).floatValue();
    if (inValue <= min) {
        outValue = min ;
    } else if (inValue >= max) {
        outValue = max ;
    } else {
       float factor = max - min ;
       outValue = inValue / factor ;
    }
    outArray[ inx] = outValue ;
```

```
    return inx+1 ;
 }

};
```

Discrete Variables

The **DiscreteVariable** class is a subclass of **Variable**. It provides the support necessary for variables which can take on a predefined set of numeric or symbolic values. The constructor takes the name of the variable as the only parameter. The minimum and maximum values can be set using the *setMin()* and *setMax()* methods directly, or they can be computed automatically if the **DiscreteVariable** is used as part of a **DataSet** by calling the *computeStatistics()* method. The value is assumed to be symbolic. If the symbol is already in the labels then it is ignored; otherwise, it is added to the list. The *normalize()* method is used by the **DataSet** class when it is creating an all-numeric version of a data set. This method converts each symbol to its index value and then converts that into a one-of-N code. The *normalizedSize()* method returns the number of unique discrete values the variable can take on, which is also the size of the one-of-N code when the variable is normalized. The *getDecodedValue()* method is used to transform the output of a neural network back into a **String** value for display.

```
class DiscreteVariable extends Variable {

 Integer min ;
 Integer max ;

 DiscreteVariable(String name) { super(name);
                                 labels = new Vector();}

 void setMin(Integer Min) { min = Min; }

 void setMax(Integer Max) { max = Max; }
 public void computeStatistics(String inValue){
    if (labels.contains(inValue)) return ;
    else labels.addElement(inValue) ;
 };

 // translate inValue index into a one-of-N code
 public int normalize(String inValue, float[] outArray, int inx) {
    int index = getIndex(inValue) ; // look up symbol index

    float code[] = new float[labels.size()] ;
```

```
    if (index < code.length) code[index] = (float)1.0 ;
    // copy one of N code to outArray, increment inx
    for (int i=0 ; i < code.length; i++) {
         outArray[inx++] = code[i] ;
    }
    return inx ;  // return output index
 }

 public int normalizedSize() { return labels.size() ; }

 // turn an array of floats back into a string value
 public String getDecodedValue(float[] act, int start) {
    int len = labels.size() ;
    String value ;
    float max = (float)-1.0 ;
    value = String.valueOf(0) ;
    for (int i=0 ; i < len ; i++) {
        if (act[start+i] > max) {
            max = act[start+i] ;
            value = getLabel(i) ;
        }
    }
    return value ;
 }

};
```

The DataSet Class

We provide multiple data sets for use with the three learning techniques. These include the exclusive-OR data set, a vehicles data set that mirrors the vehicles rule base used in Chapter 4, a restaurant data set, a linear ramp data set, and a clustering data set. Each data set is used to point out specific aspects of our learning algorithms. Because all three algorithms make use of a training data set, we have designed and implemented a common Java class called **DataSet**. The **DataSet** class definition is shown below. It makes use of the *java.io* package for loading the data from flat text files into memory, and also uses the *java.awt* classes (**TextArea**) for displaying the data and for trace information. Each **DataSet** instance has a name, a member for storing the file name, and a boolean flag which indicates whether there is symbolic data in the file. Two **Vectors** are used to store the data read from the file and a normalized data set, which we will discuss later. The *variableList* **Hashtable** holds a set of **Variables** which define the logical data types for each field in the file. The *fieldList* member holds references to the same **Variables** as the *variableList*, but the variables are added

to the **Vector** in the order they are added to the **DataSet**, as shown in the *addVariable* method. This allows us to have a one-to-one correspondence between the fields in the file and the variables which define the data. The *fieldsPerRec* member is equivalent to the size of the *fieldList* and is provided for convenience. *NumRecords* is set to the number of records in the file.

There is a single constructor for the **DataSet** class. It takes as parameters the object name and the file name.

```
// A DataSet is read from a text file
// The variables must be defined and added in the correct order
// The string data is available in the data member
// The normalized numeric data is in the normalizedData member
public class DataSet extends Object {
 String name ;
 String fileName ;
 boolean allNumericData ;  // if true use float[] else String[]

 Vector data ;             // raw data from file
 Vector normalizedData ;   // scaled and translated data
 Hashtable variableList ;  // variable definitions
 Vector fieldList ;        // field definitions where index = column

 int fieldsPerRec=0;
 int normFieldsPerRec=0;
 int numRecords=0 ;

DataSet(String Name, String FileName) {
    name = Name;              // object name
    fileName = FileName ;     // text file name
    fieldsPerRec = 0 ;        // start with no variables defined
    allNumericData = true ;   // assume all numeric data
    data = new Vector() ;            // holds string data
    variableList = new Hashtable();  // for named lookup
    fieldList = new Vector();        // for ordered lookup
 }

// for trace purposes - display all variables and their value
 public void displayVariables() {

   Enumeration enum = variableList.elements() ;
   while(enum.hasMoreElements()) {
     String values ;
     Variable temp = (Variable)enum.nextElement() ;
     if(temp.labels != null) {
        values = temp.getLabels();
     } else {
```

```
        values =  “< real>” ;
      }
      trace(“\n” + temp.name + “( ” + values + “) ”) ;
    }
  }

  public int getClassFieldSize() {
   if (variableList.get(“ClassField”) == null) {
       trace(“DataSet ” + name + “does not have a ClassField”) ;
       return 0 ;
   } else      return((Variable)variableList.get(“ClassField”)).norma
  }

. . .

}
```

Previous | Table of Contents | Next

We include a static **TextArea** which is used by the applet to display information about the data set. The *trace* method simply writes a **String** to the **TextArea**.

```
static TextArea textArea1 ;

static public void trace(String text) { textArea1.appendText(text);}
static public void setDisplay(TextArea tx1) { textArea1 = tx1; }
```

One of the major methods in the **DataSet** class is the *loadDataFile()* method. This method takes no parameters because all the information necessary to read the file into memory is provided on the *DataSet()* constructor. Before we load the data file, we first read the file format definition using the *loadDataFileDefinition()* method. We use a **FileInputStream** to read data from the file. We use the *readLine()* method to read a line at a time and a **StringTokenizer** to parse out each field value. Depending on whether the file holds all numeric data or not, we instantiate a new **String** or float array to hold the record. This array is then added to the *data* **Vector**. As we are reading the file, we keep track of the size of the file and store this information in the *numRecords* member.

```
public void loadDataFile() {

  String tempRec[]=null ;   // used when data is symbolic

  loadDataFileDefinition();  // first read the .dfn,create the vars
  fieldsPerRec = fieldList.size();

  String line=null ;
  trace("\nReading file " + fileName + ".dat with " +
         fieldsPerRec + " fields per record\n ") ;
  DataInputStream in=null ;
  try {
    in = new DataInputStream(new FileInputStream(fileName + ".dat"))
  } catch (FileNotFoundException exc) {
    trace("Error: Can't find file " + fileName + ".dat" ) ;
  }

  int recInx = 0 ;
  int token= 0 ;
  StringTokenizer input = null ;
  do {
       try {
         line = in.readLine();
```

```
        if (line != null) {
           input = new StringTokenizer(line) ;
           tempRec = new String[fieldsPerRec] ;
           data.addElement(tempRec) ; // add record
        } else {
           break ;
        }
       } catch (IOException exc){
           trace("Error reading file: " + fileName + ".dat") ;
       }

      trace("\n Record " + recInx + ": ");
      for(int i= 0 ; i < fieldsPerRec ; i++) {
         tempRec[i] = input.nextToken() ;
      ((Variable)fieldList.elementAt(i)).computeStatistics(tempRec[i
         trace(tempRec[i] + " ");
      }
      recInx++ ;
  } while (token != StreamTokenizer.TT_EOF) ;
    numRecords = recInx ;
    trace("\nLoaded " + numRecords + " records into memory.\n") ;
    normalizeData() ; // now convert to numeric form
    displayVariables() ;
    displayNormalizedData();
 }
```

The data file definition is a simple text file that contains a list of the field data types and their names. A single field must be designated as the "ClassField." The data types can be either "continuous," "discrete," or "categorical." In the current implementation, discrete and categorical are treated the same. As each field type is read from the data definition file, a **Variable** of that type is instantiated with the specified name, and the **Variable** is then added to the **DataSet** using the *addVariable()* method. The fields are assumed to be defined in the same order as the corresponding data in the data file.

```
public void loadDataFileDefinition() {

  String tempRec[]=null ;   // used when data is symbolic

  String line=null ;
  trace("\nReading file definition " + fileName + ".dfn with " +
         fieldsPerRec + " fields per record\n ") ;
  DataInputStream in=null ;
  try {
    in = new DataInputStream(new FileInputStream(fileName + ".dfn"))
  } catch (FileNotFoundException exc) {
    trace("Error: Can't find file " + fileName + ".dfn" ) ;
```

```
  }

  int recInx = 0 ;
  int token= 0 ;
  StringTokenizer input = null ;
  do {
       try {
         line = in.readLine();
         if (line != null) {
            input = new StringTokenizer(line) ;
         } else {
            break ;
         }
       } catch (IOException exc){
            trace("Error reading file: " + fileName + ".dfn") ;
      }

      trace("\n Record " + recInx + ": ");
      String varType = input.nextToken() ;
      String varName = input.nextToken() ;
      if (varType.equals("continuous")) {
        addVariable(new ContinuousVariable(varName)) ;
      } else if (varType.equals("discrete")) {
         addVariable(new DiscreteVariable(varName)) ;
      } else if (varType.equals("categorical")) {
         addVariable(new DiscreteVariable(varName)) ;
      }
      trace(varType + " " + varName);
      recInx++ ;
  } while (token != StreamTokenizer.TT_EOF) ;
    fieldsPerRec = fieldList.size() ;
    trace("\nCreated " + fieldsPerRec + " variables.\n") ;

 }

public void addVariable(Variable var) {
   variableList.put(var.name, var) ;
   fieldList.addElement(var) ;   // add in order of arrival
   var.setColumn(fieldsPerRec);
   fieldsPerRec++ ;
 }
```

The following members are part of the **Variable** class introduced in the preceding chapter, and are used by the **DataSet** class.

```
public class Variable {
...
// used by the DataSet class
public void setColumn(int col) { column = col ; }
```

```
public abstract void computeStatistics(String inValue) ;
public abstract int normalize(String inValue, float[] outArray,
                              int inx);
public int normalizedSize() { return 1 ; }
...
}
```

The next three methods deal with normalized data. Neural networks require all numeric input data and so any data sets which contain symbols must be preprocessed. Also, neural networks usually require the input data be scaled to a specific range, in our case, from 0.0 to 1.0. This conversion of symbolic data and scaling of continuous data is called *normalization*. We use the variable definitions to determine how to normalize the data. When we have a **DiscreteVariable**, we take the symbol or number and get its index value. This index value is then converted into a one-of-N vector. For example, if we have { yes, no, maybe } as three possible strings in a field defined as **DiscreteVariable**, we would convert each symbol into a code of 1s and 0s whose length is equal to the number of discrete values. So “yes” would map to 1 0 0, “no” to 0 1 0, and “maybe” to 0 0 1. Also note that we have an expansion here. An input record with 10 **DiscreteVariable** fields, each having three possible values, would expand into a thirty-element array for presentation to the neural network.

Another conversion we need to do is to scale continuous data which is out of the 0.0 to 1.0 range. We do a simple linear scaling here. There are more sophisticated techniques which could be used. The IBM Neural Network Utility, for example, uses a bilinear scaling technique where the midpoint of the input data range is mapped to the midpoint (0.5) of the output data range. This tends to “center” the data around 0.5 in cases where there are outliers or other abnormalities. Notice that our one-of-N scaling used for discrete data is already in the 0 to 1 range, so no additional scaling is necessary.

The method *getNormalizedRecordSize()* computes the expansion that occurs during this normalization process. The *normalizeData()* method performs the actual scaling and translation to one-of-N codes. The *rawData* is the data read in from the file. The *normNumRec* is the array of floats which results after the preprocessing. When all is complete, the *normalizedData* vector holds a complete normalized data set corresponding to the original data set, ready for use by the neural networks. The *displayNormalizedData()* method is a utility method to display the normalized code for debugging.

```
public int getNormalizedRecordSize() {

   int sum = 0 ;
   Enumeration vars = variableList.elements() ;
   while (vars.hasMoreElements()) {
          Variable thisVar = (Variable)vars.nextElement() ;
          sum += thisVar.normalizedSize() ;
   }
   return sum ;
  }

 // walk through the file data and scale/translate it
 // to all numeric data ranging between 0 and 1
 public void normalizeData() {
    String tempRec[]= null ;

    normalizedData = new Vector() ;
    normFieldsPerRec = getNormalizedRecordSize();
    Enumeration rawData = data.elements() ;
    while(rawData.hasMoreElements()) {
       int inx = 0 ;
       float normNumRec[] = new float[normFieldsPerRec] ;
       Enumeration fields = fieldList.elements() ;
       tempRec = (String[])rawData.nextElement() ;

       for (int i=0 ; i < fieldsPerRec ; i++) {
          Variable thisVar = (Variable)fields.nextElement() ;
          inx = thisVar.normalize(tempRec[i], normNumRec, inx) ;

       }
       normalizedData.addElement(normNumRec) ;
    }
 }

 public void displayNormalizedData() {
```

```
    float tempNumRec[] ;

    Enumeration rawData = normalizedData.elements() ;
    while(rawData.hasMoreElements()) {
       int recInx = 0 ;
       trace("\n Record " + recInx + ": ");
       tempNumRec = (float[])rawData.nextElement() ;
       int numFields = tempNumRec.length ;
       for (int i=0 ; i < numFields ; i++) {
          trace(String.valueOf(tempNumRec[i]) + " ") ;
       }
       recInx++ ;
    }
 }
```

The *getClassFieldValue()* methods are used to display the results after a neural network training run. The first method takes the record index and returns the original "ClassField" value. The second method takes an array of floats and the starting index of the output unit. The *DiscreteVariable.getDecodedValue()* method is used to select the array element with the maximum value, convert it into a discrete variable index, and then retrieve the corresponding **String** value. This is essentially the inverse of the steps performed in the *normalizeData()* when a discrete value is turned into a one-of-N code.

```
public String getClassFieldValue(int recIndex) {
    Variable classField = (Variable)variableList.get("ClassField") ;
    return ((String[])data.elementAt(recIndex))[classField.column] ;
 }

 public String getClassFieldValue(float[] activations, int index) {
    String value ;
    Variable classField = (Variable)variableList.get("ClassField") ;
    if (classField.categorical()) {
        value = classField.getDecodedValue(activations, index) ;
    } else {
        value = String.valueOf(activations[index]) ;
    }
    return value ;
 }
```

In the following sections, we present our Java implementations of back propagation and Kohonen map neural network models. They are object-oriented, but not slavishly so. We would like these algorithms to run in a reasonable amount of time on standard PC hardware. So we don't have objects for each processing unit or layer. Our experience shows that we can have the advantages

of object-oriented design and the performance of optimized algorithmic code.

Back Prop Implementation

In this section, we describe our implementation of back propagation. The **BackProp** class has four different sets of data members or parameters. First is a set which is used to manage the data. We have a reference to a **DataSet** object, *ds*, the current record index, *recInx*, the total number of records in the data set, *numRecs*, and the number of fields per record, *fieldsPerRec*. Next is a set of network architecture parameters. These include *numInputs*, *numHid1*, and *numOutputs*, which define the number of units in each of the three network layers. Some implementations support multiple hidden layers. This design can be easily extended to provide this support. *NumUnits* is the sum of the three layers of units, and *numWeights* is the total number of weights in the network.

In our implementation, we provide the following control parameters: *mode*, *learnRate*, *momentum*, and *tolerance*. When the *mode* parameter is set to a value of 0, the network is in training mode and the connection weights are adjusted. When the *mode* is set to 1, the network weights are locked. The *learnRate* and *momentum* parameters are used to control the size of the weight updates as described earlier. The *tolerance* parameter is used to specify how close the predicted output value has to be to the desired output value before the error is considered to be 0. For example, if our desired output value was 1.0, with a *tolerance* of 0.1, any predicted output value greater than 0.9 would result in 0 error.

Next, we have a set of data and error arrays. The primary arrays are the *activations*, the *weights*, and the *thresholds*. These specify the current state of the network. The *teach* array holds the desired or target output values. The other arrays are used in the computation of the error and weight changes.

```
// standard backward propagation with momentum
public class BackProp extends Object {
    String name ;

  // data parameters
    static DataSet ds ;
    Vector data ;       // train/test data from file
    int recInx=0 ;      // current record index
    int numRecs=0 ;     // number of records in data
    int fieldsPerRec=0;
```

```
  // error measures
    float sumSquaredError ;  // total SSE for an epoch
    float aveRMSError ;      // average root-mean-square error
    int numPasses ;          // number of passes over the data set

  // network architecture parameters
    int numInputs ;
    int numHid1 ;
    int numOutputs ;
    int numUnits ;
    int numWeights ;

  // network control parameters
    int mode;
    float learnRate ;
    float momentum;
    float tolerance;

  // network data
    float activations[] ;
    float weights[] ;
    float wDerivs[] ;
    float thresholds[];
    float tDerivs[] ;
    float tDeltas[] ;
    float teach[];   // target output values
    float error[];
    float deltas[];  // the error deltas
    float wDeltas[] ;

    TextArea textArea1 ;

  BackProp(String Name) {
     name = Name ;
     data = new Vector() ;
  }

    public void show_array(String name, float[] arr) {
        textArea1.appendText("\n" + name  + "= ") ;
     for (int i=0 ; i < arr.length ; i++) {
        textArea1.appendText(arr[i] + "  ") ;
     }

    }

  public void display_network() {
//      show_array("weights",weights);
//      show_array("thresholds", thresholds);
```

```
        show_array("activations",activations) ;
//      show_array("teach",teach) ;
        textArea1.appendText("\n Passes Completed: " + numPasses +
                          "  RMS Error = " + aveRMSError + " \n") ;

        String desired = ds.getClassFieldValue(recInx-1);
        String actual = ds.getClassFieldValue(activations,
                                              numInputs+numHid1);
        textArea1.appendText("\n Desired: " + desired +
                             " Actual: " + actual ) ;
    }
. . .

}
```

Previous | Table of Contents | Next

To define a back propagation neural network, we provide the *createNetwork()* method. This method is provided rather than using the constructor so that the network's architecture can be changed without creating a new object. The *createNetwork()* method takes three parameters, the number of inputs, hidden, and output units. We compute some convenient data members, initialize the control parameters, and then allocate all of the arrays. Then we call the *reset()* method to initialize the network arrays.

```
// create a Back Prop network with specified architecture
public void createNetwork(int NumIn, int NumHid1, int NumOut){

    // set the network architecture
    numInputs = NumIn ;
    numHid1 = NumHid1;
    numOutputs = NumOut;
    numUnits = numInputs + numHid1 + numOutputs ;
    numWeights = (numInputs*numHid1) + (numHid1*numOutputs);

    // initialize control parameters
    learnRate = (float)0.2;
    momentum = (float)0.7 ;
    tolerance = (float)0.1 ;
    mode = 0 ;              // 0 = train mode, 1 = run mode
    aveRMSError = (float)0.0 ;
    numPasses = 0;

    // create weight and error arrays
    activations = new float[numUnits]; // unit activations
    weights = new float[numWeights];
    wDerivs = new float[numWeights];   // accumulated wDeltas
    wDeltas = new float[numWeights];   // weight changes
    thresholds = new float[numUnits];
    tDerivs = new float[numUnits];     // accumulated tDeltas
    tDeltas = new float[numUnits] ;    // threshold changes
    teach = new float[numOutputs] ;    // desired outputs
    deltas = new float[numUnits];
    error = new float[numUnits] ;

    reset() ;  // reset and initialize the weight arrays

   return;
}

public void reset() {
  int i ;
```

```
  for (i=0 ; i < weights.length ; i++) {
    weights[i] = (float)1.0 - (float)Math.random();
    wDeltas[i] = (float)0.0 ;
    wDerivs[i] = (float)0.0 ;
  }
  for (i=0 ; i < numUnits ; i++) {
    thresholds[i] = (float)1.0 - (float)Math.random();
    tDeltas[i] = (float)0.0 ;
    tDerivs[i] = (float)0.0 ;
  }
}
```

For the forward pass, three methods are used. The *readInputs()* method takes a record from the data set and copies the input values into the *activations* of the input units. It also copies the target values into the *teach* array. The *computeOutputs()* method does the complete forward pass through the network. Starting with the first layer, it computes the sum of the threshold and each input unit activation multiplied by the corresponding weight, and then calls the *logistic()* method to compute and set the activation value.

```
public float logistic(double sum) {
      return (float)(1.0 / (1 + Math.exp(-1.0 * sum)));
   }

   // move data from train/test set into network input units
   public void readInputs() {
    recInx = recInx % numRecs ; // keep index from 0 to n-1 records
    int inx=0 ;
    float[] tempRec = (float[])data.elementAt(recInx); //get record
    for (inx=0 ; inx < numInputs ; inx++ ) {
      activations[inx] = tempRec[inx] ;
    }
    for (int i=0 ; i < numOutputs; i++ ) {
      teach[i] = tempRec[inx++] ;
    }
    recInx++ ;
   }

   // do a single forward pass through the network
   public void computeOutputs() {
    int i, j ;
    int firstHid1 = numInputs ;
    int firstOut = numInputs + numHid1 ;

    // first layer
    int inx = 0 ;
    for(i = firstHid1 ; i < firstOut ; i++) {
```

```
      float sum = thresholds[i];
      for (j = 0 ; j < numInputs ; j++) { // compute net inputs
        sum += activations[j] * weights[inx++] ;
      }
      activations[i] = logistic(sum) ;  // compute activation
    }
    // second layer
    for(i = firstOut ; i < numUnits ; i++) {
       float sum = thresholds[i] ;
      for (j = firstHid1 ; j < firstOut ; j++) { // compute sum
        sum += activations[j] * weights[inx++] ;
      }
      activations[i] = logistic(sum) ;  // compute activation
    }
   }
```

To compute the errors, we start at the output layer and work back toward the input. First, the output errors are computed by taking the difference between the *activations* produced by *computeOutputs()* and the *teach* values. Note that we also sum the squared errors here for computing the average RMS error. The *deltas* are then computed using the derivative of the activation function.

```
// compute the errors using backward error propagation
// weight changes get placed in (or added to) wDerivs
// threshold changes get placed in (or added to) tDerivs
public void computeError() {
  int i, j ;
  int firstHid1 = numInputs ;
  int firstOut = numInputs + numHid1 ;

  // clear hidden unit errors
  for(i = numInputs ; i < numUnits ; i++) {
     error[i] = (float)0.0 ;
  }

  // compute output layer errors and deltas
  for(i = firstOut ; i < numUnits ; i++) {
     error[i] = teach[i-firstOut] - activations[i];
     sumSquaredError += error[i] * error[i];
     if (Math.abs(error[i]) < tolerance) error[i] = (float)0.0;
       deltas[i] = error[i] * activations[i] * (1 - activations[i]);
  }

  // compute hidden layer errors
  int winx = numInputs * numHid1 ;    // offset into weight array
  for(i = firstOut ; i < numUnits ; i++) {
     for (j = firstHid1; j < firstOut ; j++) {
         wDerivs[winx] += deltas[i] * activations[j] ;
```

```
            error[j] += weights[winx] * deltas[i] ;
            winx++ ;
        }
        tDerivs[i] += deltas[i] ;
    }

    // compute hidden layer deltas
    for (i = firstHid1 ; i < firstOut ; i++) {
      deltas[i] = error[i] * activations[i] * (1 - activations[i]);
    }

    // compute input layer errors
    winx = 0 ;                  // offset into weight array
    for(i = firstHid1 ; i < firstOut ; i++) {
        for (j = 0; j < firstHid1 ; j++) {
            wDerivs[winx] += deltas[i] * activations[j] ;
            error[j] += weights[winx] * deltas[i] ;
            winx++ ;
        }
        tDerivs[i] += deltas[i] ;
    }

}
```

Previous | Table of Contents | Next

To adjust the *weights*, the *adjustWeights()* method first computes the current weight deltas, *wDeltas*, and then adds those deltas to the weights. Notice that we use the *wDerivs* array, which holds the accumulated set of weight changes that should be made to the *weights*. In pattern update, *wDerivs* holds only the changes from the current pattern. However, if we wanted to use batch updating, this array would hold the changes from all of the patterns in the data set. After we adjust the weights, we zero out the *wDerivs* array. Next we adjust the threshold weights for each unit in the hidden and output layer. The computations are similar to those for the regular weights. Note: It is not necessary to have a separate threshold array. Some implementations just add additional weights to the weight arrays and add a single additional unit which has a constant activation value of 1 to each layer.

```
 // apply the changes to the weights
 public void adjustWeights() {
  int i ;

  // first walk through the weights array
  for(i = 0; i < weights.length ; i++) {
    wDeltas[i] = (learnRate * wDerivs[i]) + (momentum * wDeltas[i]);
    weights[i] += wDeltas[i] ;   // modify the weight
    wDerivs[i] = (float)0.0 ;
 }

 // then walk through the threshold array
 for(i=numInputs ; i < numUnits ; i++) {
    tDeltas[i] = learnRate * tDerivs[i]  + (momentum * tDeltas[i]);
    thresholds[i] += tDeltas[i] ; // modify the threshold
    tDerivs[i] = (float)0.0 ;
 }

 // if at the end of an epoch, compute average RMS Error
 if (recInx == numRecs) {
    numPasses++ ;  // increment pass counter
    aveRMSError = (float)Math.sqrt( sumSquaredError /
                                    (numRecs * numOutputs));
    sumSquaredError = (float)0.0 ;  // clear the accumulator
 }
}

public void process() {

    readInputs() ;   // set input unit activations
```

```
    computeOutputs() ;  // do forward pass through network
    computeError() ;  // compute error and deltas

    // only adjust if in training mode
    if (mode == 0) adjustWeights() ;   // apply changes to weights

}
```

To see how our back propagation network works, we can use our Learn applet. Using a java-enabled browser, open on the *LearnApple*t.html file. The Learn applet dialog as shown in Figure 5.5 appears. Select **Back prop** as the learning method and the **XOR** data set in the **Choice** control at the top of the panel. Click on the **Load** button to read in the **XOR** data. The *xor.dfn* file and the data will be loaded into the **DataSet** and displayed in the top **TextArea** control. Press the **Start** button to train the back propagation network. When training is complete, the desired and actual values, the contents of the activations array, and the final average RMS error will be displayed in the bottom **TextArea**.

You can select other data sets to work with back prop, including the Vehicles and Animals for classification, and the Linear ramp to see an example of a prediction problem. The contents of each data set are listed in Appendix B. Whenever you select a different data set, be sure to press the **Load** button. The **Reset** button will clear the bottom **TextArea** and reset the weights in the back prop network.

Kohonen Map Implementation

For our implementation of the Kohonen map neural network, we define a class named **KMapNet**. The structure of this class is similar to that of the **BackProp** class already presented in this chapter. The major difference between the **KMapNet** and the **BackProp** class is the unsupervised versus supervised learning paradigm. The Kohonen map performs unsupervised learning, so there is no notion of a desired output; there are only inputs.

The **KMapNet** class has four different sets of data members or parameters. First is a set which is used to manage the data. We have a reference to a data set, *ds*, the current record index, *recInx*, the total number of records in the data set, *numRecs*, and the number of fields per record, *fieldsPerRec*. Next is a set of network architecture parameters. These include *numInputs*, *numRows*, and *numCols*, which define the number of units in the input layer and the dimensions of the two-dimensional output layer. *NumUnits* is the sum of these two layers,

and *numWeights* is the total number of weights in the network. In our relatively simple implementation, our control parameters consist of the *initalLearnRate* and *finalLearnRate*, and the current *LearnRate*. The network *mode* is either train or test. *Sigma* is used in the computation of the neighborhood function.

Next we have a set of data and error arrays. The primary arrays are the *activations* and the *weights*, which specify the current state of the network. The *distance* array is a precomputed grid which defines the distances between units on the two-dimensional output grid. For example, in a network with a 4-by-4 output, the distances from unit 0,0 (top left) would be:

0	1	16	81
1	2	17	82
16	17	32	97
81	82	97	162

As you can see, the distance from unit to itself is 0, to those units horizontally and vertically adjacent it is 1, and the distances grow as you move away from the unit. The distance array is used in the *adjustNeighborhood()* method.

```
public class KMapNet extends Object {
   String name ;

 // data parameters
   static DataSet ds;
   Vector data ;
   int recInx=0 ;  // current record index
   int numRecs=0 ; // number of records in data
   int fieldsPerRec;
   int numPasses=0 ;

 // network architecture parameters
   int numInputs ;
   int numRows ;
   int numCols ;
   int numOutputs ;
   int numUnits ;

 // network control parameters
   int mode;
   float learnRate ;
   float initLearnRate = (float)1.0 ;
   float finalLearnRate = (float)0.05 ;
   float sigma ;
```

```
   int maxNumPasses=20 ;  // default

 // network data
   int winner ;          // index of the winning unit
   float activations[];
   float weights[];
   int   distance[];   // used in neighborhood computation

   static TextArea textArea1 ;

 KMapNet(String Name) {
      name = Name ;
      data = new Vector() ;
 }

   public void show_array(String name, float[] arr) {
       textArea1.appendText("\n" + name  + "= ") ;
    for (int i=0 ; i < arr.length ; i++) {
       textArea1.appendText(arr[i] + "  ") ;
    }

   }
   public void display_network() {
//        show_array("weights",weights);
         show_array("activations",activations) ;
         textArea1.appendText("\nWinner = " + winner + "\n") ;
   }

 . .

}
```

To define a Kohonen map neural network, we provide the *createNetwork()* method. This method is provided rather than using the constructor so that we can change the network's architecture without creating a new object. The *createNetwork()* method takes three parameters; the number of inputs, rows, and columns. We compute some convenience data members, initialize the control parameters, and then allocate the activations and weights arrays. The *computeDistances()* method initializes the distance array. Then we call the *adjustNeighborhood()* method to set the initial learn rate and then call *reset()* to initialize the weight array.

```
// create a Kohonen network with the specified architecture
    public void createNetwork(int NumIn, int NumRows, int NumCols){

        // set the network architecture
        numInputs = NumIn;
```

```
        numRows = NumRows;
        numCols = NumCols;
        numOutputs = numRows * numCols ;
        numUnits = numInputs + numOutputs;

        // initialize control parameters
        learnRate = (float)0.1;
        mode = 0 ;     // 0 = train mode, 1 = run mode

        // create arrays
        activations = new float[numUnits];
        weights = new float[numInputs*numOutputs];

        // fill in the Distance matrix
        computeDistances() ;

        adjustNeighborhood() ; // set the initial learnRate
        reset() ;              // reset and initialize weight arrays

        return ;  // all data in the network
    }

    public void reset() {
        int i ;
        for (i=0 ; i < weights.length ; i++) {
          weights[i] = (float)0.6 - ((float)0.2
                    *(float)Math.random());    // between 0.4 and 0.
        }
    }

// initialize a matrix of distances from each unit to all others
    public void computeDistances() {
        int i, j, xi, xj, yi, yj ;
        distance = new int[numOutputs* numOutputs] ;
        for(i=0 ; i < numOutputs ; i++) {
          xi = i % numCols ;
          yi = i / numRows ;
          for (j=0 ; j < numOutputs ; j++) {
            xj = j % numCols ;
            yj = j / numRows ;
            distance[i*numOutputs+j] =
                  (int) Math.pow((double)((xi-xj)*(xi-xj)),2.0)+
                  (int) Math.pow((double)((yi-yj)*(yi-yj)),2.0);
          }
        }
    }
```

To process an input pattern, three methods are used. The *ReadInputs()* method takes a record from the data set and copies the input values into the *activations* of the input layer units. Because Kohonen maps use unsupervised learning, there are no target values. The *computeOutputs()* method computes the Euclidean distance between the input vector and the weight vectors. The *computeWinner()* method chooses the output with the minimum activation (closest to the input vector) as the *winner*. Note, we could do this function in our *computeOutputs()* method in this case. However, if we add more sophisticated winner selection techniques, such as learning with a conscience, we would want this broken out into a separate step.

```
// move data from train/test set into network input units
    public void readInputs() {
     recInx = recInx % numRecs ; // keep index from 0 to n-1 records
     int inx=0 ;
     float[] tempRec = (float[])data.elementAt(recInx); // get recor
     for (inx=0 ; inx < numInputs ; inx++ ) {
       activations[inx] = tempRec[inx] ;
     }
     recInx++ ;
     if (recInx ==0){
        numPasses++ ;            // completed another pass
        adjustNeighborhood() ;
     }
    }

    // do a single forward pass through the network
    // compute the Euclidean distance from input vector
    // to all output units
    public void computeOutputs() {
     int index, i, j ;
     int lastOut = numUnits - 1;
     int firstOut = numInputs;

     //first layer
     for(i = firstOut ; i <= lastOut ; i++) {
        index = (i - firstOut) * numInputs;
        activations[i] = (float)0.0 ;
       for (j = 0 ; j < numInputs ; j++) { // compute net inputs
         activations[i] += (activations[j] - weights[index+j]) *
                           (activations[j] - weights[index+j]);
       }
     }
   }
```

```
    // find the output unit with the smallest activation
    // and declare it the “winner”
    public void selectWinner() {

       winner = 0;
       float min = activations[numInputs];

       for(int i = 0 ; i < numOutputs ; i++) {
         if (activations[i+numInputs] < min) {
            min = activations[i+numInputs] ;
            winner = i ;
         }
       }
    }
```

The *adjustNeighborhood()* method computes two crucial values as training progresses, the learn rate and the neighborhood width or *sigma*. First, a *ratio* is computed which tells how far along we are in the training. This *ratio* increases over time. Then, the current *learnRate* is computed, decreasing over time. Finally, the neighborhood value, *sigma*, is computed.

```
public void adjustNeighborhood() {
        double ratio = (double) (numPasses/ maxNumPasses) ;
        learnRate = initLearnRate *
                    (float)Math.pow((double)finalLearnRate /
                    (double)initLearnRate,ratio);
        sigma = (float)numCols * (float)Math.pow((0.20 /
                                  (double)numCols),ratio) ;

}
```

Once the *winner* is selected, the *weights* are adjusted. In the *adjustWeights()* method, we use the neighborhood, the distance from the winning unit, and the *learnRate* to adjust the weights of units in the neighborhood of the winner.

```
    public void adjustWeights() {

    // apply the changes to the weights
     int i, j, inx, base ;
     int numOutputs = numRows * numCols ;
     double dist, range, sigma_squared ;

     sigma_squared = sigma * sigma ;

     for(i = 0 ; i < numOutputs ; i++) {
        dist = Math.exp((distance[winner*numOutputs+i] * -1.0) /
```

```
                    (2.0 * sigma_squared));

            base = i * numInputs ; // compute the base index
            range = learnRate * dist ;
            for (j=0 ; j < numInputs ; j++) {
                inx = base + j ;
                weights[inx] += range * (activations[j]- weights[inx]);
            }
          }
        }

        public void cluster() {

         readInputs() ;   // set input unit activations
         computeOutputs() ;  // do forward pass through network
         selectWinner() ;  // find the winning unit

         // only adjust if in training mode
         if (mode == 0) adjustWeights() ;   // apply changes to weights

        }
```

Now we look at an example of Kohonen map training using our Learn applet. Using a java-enabled browser, open on the *LearnApplet.html* file. The Learn applet dialog as shown in Figure 5.5 appears. Select **Kohonen map** as the learning method and the **Cluster** data set in the **Choice** control at the top of the panel. Click on the **Load** button to read in the **Cluster** data. The *kmap1.dfn* file and the data will be loaded into the **DataSet** and displayed in the top **TextArea** control. Press the **Start** button to train the Kohonen map network. When training is complete, the *winner* output unit index and the contents of the *activations* array will be displayed for each record in the bottom **TextArea**.

Decision Tree Implementation

Now, let's turn to our implementation of a decision tree algorithm that handles discrete variables. There are two main classes: a **Node** which represents a decision tree node, and the **DecisionTree** class itself. The **Node** has a name or *label*, a **Vector** of links to other **Nodes** in the **DecisionTree**, a reference to the parent node, and a **Vector** containing references to children nodes.

```
  // decision tree node
class Node {

 String label ;      // name of the node
```

```
 Vector linkLabels;  // tests on links from parent to child
 Node parent ;       // parent node
 Vector children ;   // any children nodes

 public Node() { parent = null ;
                 children = new Vector();
                 linkLabels = new Vector(); }

 public Node(String Label) {
                 label = Label ;
                 children = new Vector() ;
                 parent = null;
                 linkLabels = new Vector() ; }

 public Node(Node Parent, String Label) {
                 parent = Parent ;
                 children = new Vector() ;
                 label = Label ;
                 linkLabels = new Vector() ; }
 void addChild(Node child, String linkLabel) {
                 children.addElement(child) ;
                 linkLabels.addElement(linkLabel) ;}
 boolean hasChildren() {
                 if (children.size() == 0) return false;
                 else return true; }
 void setLabel(String Label) { label = Label ; }

 static void displayTree(Node root) {
     if (root.children.size() == 0) {
        DecisionTree.appendText("\nLeaf node - " + root.label) ;
        return ;
     } else {
       Enumeration enum = root.children.elements() ;
       Enumeration enum2 = root.linkLabels.elements() ;
       DecisionTree.appendText("\nInterior node - " + root.label) ;
       while (enum.hasMoreElements()) {
         DecisionTree.appendText("\nLink - " +
                                 (String)enum2.nextElement()) ;
         displayTree((Node)enum.nextElement()) ;
       }
     }
 }

};
```

The **DecisionTree** class has a *name*, a reference to a **DataSet** object, as in the neural network classes, the goal **Variable**, *classVar*, and a list of all of the **Variable** definitions for the data set. The **Vector** of examples holds the set of

training records which is the result of a partitioning of the original training set, based on an attribute test. The **DecisionTree** has a simple constructor allowing the name to be set.

```
public class DecisionTree {
 String name ;
 static DataSet ds ;
 Variable classVar ; // the class Variable
 Hashtable variableList ;

 Vector ruleList ;           // list of all rules
 Vector conclusionVarList ;  // queue of variables
 Vector examples ;
 int fieldsPerRec;
 static TextArea textArea1 ;

 static public void appendText(String text) {
                                    textArea1.appendText(text);}
 String record[] ;  // one record of train/test data in string forma

 DecisionTree(String Name) {
    name = Name;
 }
 . . .
}
```

Previous | Table of Contents | Next

The *buildDecisionTree()* method is the major method in this class. It takes three parameters, a **Vector** of examples (the data set), a list of the field definitions, and the default value that should be returned in case the decision tree fails. The first thing we do is instantiate a root **Node** for the decision tree. If the training set is empty, then we return with the default value. Otherwise, if all of the records in the training set have an identical value for the class field, we return with a leaf **Node** with that value. If neither of these two conditions are true, we must build a decision tree. The first step is to select the best attribute or variable on which to split. The *chooseVariable()* method does this selection using information theory. We assign the root of this tree to a **Node** for that variable. Now that we have decided which **Variable** to split on, we must determine which discrete value of that attribute we should use for the split. The *for()* loop does this by taking each possible value of the variable, subsetting the training data (examples) into groups where all records have the desired value of that attribute, and computing the information gain for that value of the variable. We recursively call *buildDecisionTree()* to fill out branches until we have built a complete decision tree starting with our root **Node**.

```
public Node buildDecisionTree(Vector examples,
                              Hashtable variables,
                              Node defaultValue)
 {

 Node tree = new Node() ;

 if (examples.size() == 0) return defaultValue ;
 else if (identical(examples, classVar))
      return
     new Node(((String[])examples.firstElement())[classVar.column])
 else if (variables.size() == 0)
        return new Node(majority(examples)) ;
 else {
    Variable best = chooseVariable(variables, examples);
    tree = new Node(best.name) ;      // variable with most Gain
    Enumeration enum = best.labels.elements();
    int numValues = best.labels.size();
    for (int i=0 ; i < numValues; i++) {
        Vector examples1 = subset(examples, best, best.getLabel(i))
        Hashtable variables1 = (Hashtable)variables.clone();
        variables1.remove(best) ;
        Node subTree = buildDecisionTree( examples1,variables1,
                                          new Node(majority(examples1)))
```

```
        tree.addChild(subTree, best.name + "=" + best.getLabel(i) )
    }
 }
    return tree ;
 }
```

The *chooseVariable()* method takes two parameters: the list of variables to consider, and a subset of the training data. First we call *getCounts()* to compute the number of positive and negative examples of our class variable, which are returned as elements of an integer array. Next we compute the information value of this data set using the *p* and *n* returned by *getCounts()*. Note that in this implementation, we only have binary class variables. Finally we compute the remainder for each variable and select the one that results in the largest information gain.

```
// return the variable with most gain
Variable chooseVariable(Hashtable variables, Vector examples)
{
  Enumeration enum = variables.elements() ;
  double gain = 0.0, bestGain = 0.0 ;
  Variable best = null ;
  int counts[] ;
  counts = getCounts(examples) ;
  int pos = counts[0] ;
  int neg = counts[1] ;
  double info = computeInfo(pos, neg);
  textArea1.appendText("\nInfo = " + info ) ;

  while(enum.hasMoreElements()) {
     Variable tempVar = (Variable)enum.nextElement() ;
     gain = info - computeRemainder(tempVar, examples);
     textArea1.appendText("\n" + tempVar.name + " gain = " + gain) ;
     if (gain > bestGain) {
       bestGain = gain ;
       best = tempVar;
     }
  }
  textArea1.appendText("\nChoosing best variable: " + best.name) ;
  return best;  //

}
```

The *computeInfo()* method takes two integer parameters, *p* and *n*, as inputs. It computes the information value as defined in the formula for *I()* above. The only tricky part is the computation of the **log2 n**, which uses the mathematical fact that the log of any base can be computed using the natural log of n divided by

the natural log of the base 2. The *computeRemainder()* method takes a single variable and a data set as inputs. It computes the number of positive and negative cases for each unique value the discrete variable can take on. It sums the information value for each distinct value weighted by the prior probability of that value and returns it as the remainder.

```
// compute information content, given # of pos and neg examples
 double computeInfo(int p, int n) {

  double total = p + n ;
  double pos = p / total ;
  double neg = n / total;
  double temp;
  if ((p ==0) || (n == 0)) {
    temp = 0.0 ;
  } else {
   temp = (-1.0 * (pos * Math.log(pos)/Math.log(2))) -
                        (neg * Math.log(neg)/Math.log(2)) ;
  }
  return temp ;
 }

 double computeRemainder(Variable variable, Vector examples)
 {
   int positive[] = new int[variable.labels.size()] ;
   int negative[] = new int[variable.labels.size()] ;
   int index = variable.column ;
   int classIndex = classVar.column ;
   double sum = 0 ;
   double numValues = variable.labels.size() ;
   double numRecs = examples.size() ;

   for( int i=0 ; i < numValues ; i++) {
     String value = variable.getLabel(i); // get discrete value

     Enumeration enum = examples.elements() ;
     while (enum.hasMoreElements()) {
       String record[] = (String[])enum.nextElement();
       if (record[index].equals(value)) {
        if (record[classIndex].equals("yes")) {
            positive[i]++ ;
        } else  {
            negative[i]++;
        }
       }
     } /* endwhile */
     double weight = (positive[i]+negative[i]) / numRecs;
     double myrem = weight * computeInfo(positive[i], negative[i]);
```

```
     sum = sum + myrem ;
    } /* endfor */

   return sum ;
 }
```

Finally, we have several utility methods. The *subset()* method takes a data set, a variable definition, and a value for that variable and returns the subset of data which contains that value in every record. The *identical()* method takes the data set and a single variable as input, and returns a boolean indicating whether all of the training records contain the same value for the specified variable. The *majority()* method examines a data set and returns the label which has the most examples for the specified variable. The *getCounts()* method computes the number of times each discrete value appears for the specified variable in the data set.

```
// return a subset of examples with (variableName = value)
 Vector subset(Vector examples, Variable variable, String value)
 {
   int index = variable.column ;
   Enumeration enum = examples.elements() ;
   Vector matchingExamples = new Vector() ;

   while(enum.hasMoreElements()) {
      String[] record = (String[])enum.nextElement() ;
      if (value.equals(record[index])) {
        matchingExamples.addElement(record) ;
      }
   }
   textArea1.appendText("\n Subset - there are " +
                        matchingExamples.size() +
                        " records with " +
                        variable.name + " = " + value) ;

   return matchingExamples;
 }


public boolean identical(Vector examples, Variable variable)
 {
   int index = variable.column ;
   Enumeration enum = examples.elements() ;
   boolean same = true ;
   String value = ((String[])examples.firstElement())[index] ;
   while(enum.hasMoreElements()) {
      if (value.equals(((String[])enum.nextElement())[index]))
             continue ;
```

```
      else { same = false ; break ; }
  }
  return same;

}

public String majority(Vector examples)
{
  int index = classVar.column ;
  Enumeration enum = examples.elements() ;
  int counts[] = new int[classVar.labels.size()] ;

  while(enum.hasMoreElements()) {
     String value = ((String[])enum.nextElement())[index];
     int inx = ((Variable)classVar).getIndex(value) ;
     counts[inx]++;
  } /* enbwhile */

  int maxVal = 0 ;
  int maxIndex = 0 ;
  for(int i=0 ; i < classVar.labels.size() ; i++) {
    if (counts[i] > maxVal) {
       maxVal = counts[i] ;
       maxIndex = i ;
    } /* endif */
  } /* endfor */
  return classVar.getLabel(maxIndex);
}

public int[] getCounts(Vector examples)
{
  int index = classVar.column ;
  Enumeration enum = examples.elements() ;
  int counts[] = new int[classVar.labels.size()] ;

  while(enum.hasMoreElements()) {
     String value = ((String[])enum.nextElement())[index];
     int inx = ((Variable)classVar).getIndex(value);
     counts[inx]++;
  } /* enbwhile */

  return counts;
}
```

We use the Restaurant data set and our Learn applet to show an example of a decision tree. Using a Java-enabled browser, open the *LearnApplet.html* file. The Learn applet dialog as shown in Figure 5.5 appears. Select **Decision Tree** as the learning method and the **Restaurant** data set in the choice control at the top of

the panel. Click on the **Load** button to read in the **Restaurant** data. The *resttree.dfn* file and the data will be loaded into the **DataSet** and displayed in the top **TextArea** control. Press the **Start** button to build the decision tree. When training is complete, the structure of the decision tree will be displayed in the bottom **TextArea**.

Previous | Table of Contents | Next

The Learn Applet Implementation

The Learn applet is a Java applet that was developed using the Symantec Visual CafŽ development tool. The applet **Frame** consists of two **TextArea** controls for displaying the **DataSet** and training information respectively, three radiobutton controls for selecting the learning algorithm, a **Choice** control for selecting the **DataSet**, and four **awt.Buttons** as shown in Figure 5.5. When the **Start** button is clicked, a corresponding *test()* method is called, depending on the selected algorithm.

```
public class LearnApplet extends Applet {
     void button1_Clicked(Event event) {

       // start -- first get the selected data set
       String dsName = choice1.getSelectedItem() ;
       dataSet = (DataSet)dataSets.get(dsName) ;

        if (radioButton1.getState() == true)
               testBackProp(dataSet, textArea2);
        if (radioButton2.getState() == true)
               testKMapNet(dataSet,textArea2) ;
       if (radioButton3.getState() == true)
               testDecisionTree(dataSet, textArea2);
     }

     // stop
     void button2_Clicked(Event event) {
            // for future use
     }
     // reset
     void button3_Clicked(Event event) {

          //{{CONNECTION
          // Clear the text for TextArea
          textArea2.setText("");
          //}}
     }

     // load the selected data set
    public void button4_Clicked(Event event) {
        String dsName = choice1.getSelectedItem() ;
        dataSet = (DataSet)dataSets.get(dsName) ;
        if (dataSet.numRecords > 0) {
          textArea1.appendText("\nDataSet '" + dsName +
```

```
                          "' was already loaded!\n") ;
   } else {
     dataSet.loadDataFile() ; // load the data set
   }
}

public void init() {
     super.init();

     //{{INIT_CONTROLS
     setLayout(null);
     addNotify();
     resize(518,495);
     label1 = new java.awt.Label("Data Set");
     label1.reshape(12,12,168,25);
     add(label1);
     textArea1 = new java.awt.TextArea();
     textArea1.reshape(0,48,516,120);
     add(textArea1);
     textArea2 = new java.awt.TextArea();
     textArea2.reshape(0,204,516,168);
     add(textArea2);
     button1 = new java.awt.Button("Start");
     button1.reshape(36,444,96,36);
     add(button1);
     button2 = new java.awt.Button("Stop");
     button2.reshape(192,444,109,32);
     add(button2);
     button3 = new java.awt.Button("Reset");
     button3.reshape(372,444,97,36);
     add(button3);
     radioButtonGroupPanel1 =
         new symantec.itools.awt.RadioButtonGroupPanel();
     radioButtonGroupPanel1.setLayout(null);
     radioButtonGroupPanel1.reshape(12,384,499,43);
     add(radioButtonGroupPanel1);
     Group1 = new CheckboxGroup();
     radioButton1 = new java.awt.Checkbox("back prop",
                                          Group1, false);
     radioButton1.reshape(24,12,135,24);
     radioButtonGroupPanel1.add(radioButton1);
     radioButton2 = new java.awt.Checkbox("Kohonen map",
                                          Group1, false);
     radioButton2.reshape(180,12,114,22);
     radioButtonGroupPanel1.add(radioButton2);
     radioButton3 = new java.awt.Checkbox("Decision Tree",
                                          Group1, false);
     radioButton3.reshape(348,12,132,24);
     radioButtonGroupPanel1.add(radioButton3);
     choice1 = new java.awt.Choice();
```

```
      add(choice1);
      choice1.reshape(180,12,182,24);
      label2 = new java.awt.Label("Status ");
      label2.reshape(0,168,416,31);
      add(label2);
      button4 = new java.awt.Button("Load");
      button4.reshape(396,12,108,28);
      add(button4);
      //}}
     radioButton1.setState(true) ; // select back prop
    DataSet.setDisplay(textArea1) ; // top text area
    dataSets = new Hashtable() ;
    choice1.addItem("XOR") ;   // select XOR as default
    dataSets.put("XOR", new DataSet("XOR","xor")) ;
    choice1.addItem("Vehicles");
    dataSets.put("Vehicles",
                 new DataSet("Vechicles", "vehicles")) ;
    choice1.addItem("Restaurant");
    dataSets.put("Restaurant",
                 new DataSet("Restaurant","resttree")) ;
    choice1.addItem("Linear Ramp") ;
    dataSets.put("Linear Ramp",
                 new DataSet("Linear Ramp","ramp2")) ;
     choice1.addItem("Cluster Data") ;
    dataSets.put("Cluster Data",
                 new DataSet("Cluster Data","kmap1")) ;
    choice1.addItem("Animal Data") ;
    dataSets.put("Animal Data",
                 new DataSet("Animal Data","animal")) ;
    choice1.addItem("XOR Tree Data") ;
    dataSets.put("XOR Tree Data",
                 new DataSet("XOR Tree Data","xortree")) ;
    }

public boolean handleEvent(Event event) {
  if (event.target == button1 && event.id==Event.ACTION_EVENT) {
       button1_Clicked(event);
       return true;
  } if (event.target == button2 && event.id==Event.ACTION_EVENT)
       button2_Clicked(event);
       return true;
  }if (event.target == button3 && event.id==Event.ACTION_EVENT) {
       button3_Clicked(event);
       return true;
  }if (event.target == button4 && event.id==Event.ACTION_EVENT) {
       button4_Clicked(event);
       return true;
  }
  return super.handleEvent(event);
```

```
  }

      //{{DECLARE_CONTROLS
    java.awt.Label label1;
    java.awt.TextArea textArea1;
    java.awt.TextArea textArea2;
    java.awt.Button button1;
    java.awt.Button button2;
    java.awt.Button button3;
    symantec.itools.awt.RadioButtonGroupPanel radioButtonGroupPanel1
    java.awt.Checkbox radioButton1;
    CheckboxGroup Group1;
    java.awt.Checkbox radioButton2;
    java.awt.Checkbox radioButton3;
    java.awt.Choice choice1;
    java.awt.Label label2;
    java.awt.Button button4;
    //}}

    Hashtable dataSets ;
    DataSet dataSet = null ;

public static void testBackProp(DataSet dataSet, TextArea bottomText
    BackProp testNet = new BackProp("Test Back Prop Network");
    testNet.textArea1 = bottomText ;
    testNet.ds = dataSet ;
    testNet.numRecs = dataSet.numRecords ;
    testNet.fieldsPerRec = dataSet.normFieldsPerRec ;
    testNet.data = dataSet.normalizedData ; // get vector of data
    int numOutputs = dataSet.getClassFieldSize() ;
    int numInputs = testNet.fieldsPerRec - numOutputs;

    testNet.createNetwork(numInputs,numInputs,numOutputs) ;
    int maxNumPasses = 2500 ;  // default -- could be on applet
    int numSteps = maxNumPasses * testNet.numRecs ;
    for (int i=0 ; i < numSteps; i++) {
      testNet.process() ;   // train
    }

    testNet.mode = 1 ;  // lock the network
    // do a final pass and display the results
    for (int i=0 ; i < testNet.numRecs ; i++) {
        testNet.process() ;   // test
        testNet.display_network() ;
    }
}

public static void testKMapNet(DataSet dataSet, TextArea bottomText)
         KMapNet testNet = new KMapNet("Test Kohonen Map Network");
```

```
        testNet.textArea1 = bottomText ;
        testNet.ds = dataSet ;
        testNet.numRecs = dataSet.numRecords ;
        testNet.fieldsPerRec = dataSet.fieldsPerRec ;
        testNet.data = dataSet.normalizedData ; // get vector of dat

        // create network, all fields are inputs
         testNet.createNetwork(testNet.fieldsPerRec, 4, 4) ;
        int maxNumPasses = 20 ;  // default -- could be on applet
         int numCycles = maxNumPasses * testNet.numRecs ;

        // train the network
         for (int i=0 ; i < numCycles; i++) {
           testNet.cluster() ;   // train
        }

         testNet.mode = 1 ;  // lock the network weights
         for (int i=0 ; i < testNet.numRecs ; i++) {
          testNet.cluster() ;   // test
          testNet.display_network() ;
        }
     }

public static void testDecisionTree(DataSet dataSet,
                                    TextArea bottomText)
 {
    DecisionTree tree = new DecisionTree("\n Test Decision Tree ") ;
    tree.textArea1 = bottomText ;
    tree.ds = dataSet ;

  tree.textArea1.appendText("Starting DecisionTree ") ;

  tree.fieldsPerRec = dataSet.fieldsPerRec;
  tree.examples = dataSet.data ; // get vector of data
  tree.variableList = dataSet.variableList ;
  tree.classVar = (Variable)tree.variableList.get("ClassField") ;
  tree.variableList.remove("ClassField") ;

  // recursively build tree
  Node root = tree.buildDecisionTree(tree.examples,
                                    tree.variableList,
                                    new Node("default")) ;

  // now display the results
  tree.textArea1.appendText("\nDecisionTree -- classVar = "
                           + tree.classVar.name);
  Node.displayTree(root) ;
  tree.textArea1.appendText("\nStopping DecisionTree - success!") ;
 }
```

```
}
```

Classifier Systems

Classifier systems were developed by John Holland as a way to introduce learning to rule-based systems. His mechanism was based on a technique known as genetic algorithms.

Genetic Algorithms

Genetic algorithms use a biological process metaphor to derive solutions to problems. First, the problem state must be encoded into a string which is usually a binary string, but could be a string of real values. There must be an evaluation or objective function which takes that encoded string as input and produces a "goodness" or biological "fitness" score. This score is then used to rank a set of strings (individuals in the population of strings) based on their fitness. If you are familiar with any of Darwin's theories you can probably see where this is going.

The strings (individuals) which are most fit are randomly selected to survive and even procreate into the next generation of strings. Two genetically inspired operators are used to modify the selected string, in an attempt to improve its fitness and get even closer to the goal or optimum solution. First, there is mutation, which, like biological mutation, performs random mutations or changes in the chromosome of the individual. In the usual binary string case, we randomly flip a few bits. Second, there is crossover, where two particularly fit individuals are selected and genetic material from them is combined, forming a new individual or child. The intuitive rationale for this operation is that because both parents were fit individuals, the chances are good that the offspring will also be well-endowed in terms of providing a good solution to the problem at hand.

Genetic algorithms are particularly suited to optimization problems because they are, in essence, performing a parallel search in the state space. The crossover operator injects large amounts of noise into the process to make sure that the entire search space is covered. The mutation operator allows fine-tuning of fit individuals in a manner similar to hill-climbing search techniques. Control parameters are used to determine how large the population is, how individuals are selected from the population, and how often any mutation and crossover is

performed.

By representing knowledge in rule form and then internally representing the rules as binary strings, Holland used the genetic algorithms to adapt his rules (Holland et al. 1986). The optimizing powers were focused on improving the applicability of the knowledge base itself.

Summary

In this chapter, we discussed machine learning techniques such as neural networks and decision trees. The main points include:

- There are several different forms of learning. *Rote learning* is based on memorization of examples. *Feature extraction* and *induction* are used to find important characteristics of a problem and to build a model that can be used for prediction in new situations. *Clustering* is a way to organize similar patterns into groups.
- *Supervised learning* relies on a teacher that provides the input data as well as the desired solution. *Unsupervised learning* depends on input data only and makes no demands on knowing the solution. *Reinforcement learning* is a kind of supervised learning, where the feedback is more general.
- *On-line learning* means that the agent is adapting while it is working. *Off-line learning* involves saving data while the agent is working and using that data later to train the agent.
- *Neural networks* are parallel computing models that adapt when presented with training data. They operate in supervised, unsupervised, and reinforcement learning modes. A neural network is comprised of a set of simple processing units and a set of adaptive, real-valued connection weights. Learning in neural networks is accomplished through the adjustment of the connection weights.
- *Back propagation networks* are supervised, feedforward neural networks that use the backward propagation of errors algorithm to adjust the connection weights. They are used for classification and prediction problems.
- *Kohonen map networks* are unsupervised, feedforward neural networks that self-organize and learn to map similar inputs onto output units which are in close proximity to each other. The inputs are measured using Euclidean distance in a high-dimensional input space and are mapped onto

a two-dimensional output space. Kohonen maps perform clustering of input data.

• *Decision trees* use *information theory* to determine where to split data sets in order to build classifiers and regression trees.

• *Classifier systems* are rule systems which use *genetic algorithms* to modify the rule base. Genetic algorithms use a biological metaphor to perform parallel search in the state space. The problem state is encoded as binary strings and is modified using genetic operators such as crossover and mutation.

Exercises

1. Given the following situations, which learning paradigm (supervised, unsupervised, or reinforcement) is most appropriate and why?

- Ranking articles based on specific user feedback
- Splitting customers into groups with similar preferences
- Playing tic-tac-toe or checkers

2. Add a new data set to the **LearnApplet** for classifying coins by their attributes. This means a new data definition file must also be created. Use the XOR data set in the **LearnApplet** as an example. Train a back propagation network to classify the coins. How many examples did you need to get satisfactory results?

3. Use the same data set of examples and cluster the data with a Kohonen feature map.

4. Use the same data set from exercise 2 and build a decision tree. Compare the classification accuracy of the back propagation network. How did the training time compare?

Chapter 6
Intelligent Agents

In this chapter we explore the relationship between artificial intelligence and intelligent agents in detail. We show how artificial intelligence topics such as knowledge representation, reasoning, and learning can be combined to construct intelligent agents. We discuss the requirements for autonomous intelligent agents including perception, reasoning, and the ability to take actions. We describe one of the earliest multiagent architectures, blackboard systems, and then discuss some current ideas about agent communications, specifically the Knowledge Query and Manipulation Language. Finally, we look at systems where multiple intelligent agents communicate, cooperate, and compete.

From AI to IA

In the preceding chapters, we have explored the major topics in artificial intelligence, search techniques, knowledge representations, reasoning algorithms, and learning. In the introductory chapter, we described how intelligent agents can be used to enhance the capabilities of applications and to help users to get their work done. In this section, we point out the explicit links between the basic AI technologies and the corresponding intelligent agent (IA) behaviors. We also point out the features of the Java language that are applicable or important for implementing these behaviors.

In Chapter 2, we discussed state-based problem solving using search. In Chapter 3, we explored several forms of knowledge representation. For our intelligent agents, knowledge representation is a crucial issue. As in any AI application, what our agent is expected to do, and in what domain, will have a significant impact on the type of knowledge representation we should use. If our agent has a limited number of situations it needs to respond to, maybe hardcoding the intelligence into procedural program code is the solution. If our agent has to build or use sophisticated models of the problem domain and solve problems at different levels of abstraction, then frames or semantic nets are the answer.

However, if the agent has to answer questions or generate new facts from the existing data, then predicate logic or if-then rules should be considered. In many applications, we'll need to use a mixture of these knowledge representations. If our agent is going to interact with other agents and must share knowledge then it should probably be able to read and write KIF data.

The amount of intelligence, in terms of both domain knowledge and power of the reasoning algorithms, required by our agent is related to the degree of autonomy and, to a lesser extent, mobility it has. If our agent is going to have to deal with a wide range of situations, then it needs a broad knowledge base and a flexible inference engine. If it is mobile, there may be a premium on having a small, compact knowledge representation and a light-weight reasoning system in terms of code size. At the same time, mobility can introduce security concerns which may make explicit if-then rules less desirable than another knowledge representation. For example, a neural network's "black-box" attributes, which are seen as a disadvantage from a traditional symbolic AI perspective, become a distinct advantage when viewed from a knowledge-security vantage point.

Whether learning is a desirable function depends on the domain the intelligent agent will work in, as well as the environment. If the agent is long-lived and will perform similar tasks many times during its lifetime, then learning can be used to improve its performance. But adding learning would be overkill if the agent will be used only occasionally. If we know the major rules of thumb, why not just program them into the agent and have it be capable of performing well from the start, instead of adding the complexity and costs associated with learning algorithms? But there are many domains, such as personal assistants, where we do not know the user's preferences beforehand. One option is to ask the user to explicitly state what her preferences are. But this may be tedious for the user and, depending on the domain, may be impossible. Often, people know what they like but can't readily express it in words. That is where the advantage of a learning agent surfaces. By watching what the user does in certain situations, the intelligent agent can learn what the user prefers in a much less obtrusive way than by playing a game of "20 Questions."

If there will be a large amount of uncertainty in the problem domain, then using Bayesian networks or if-then rules with certainty factors may be appropriate. If the agent must find an optimal answer, then state-based search techniques or biologically based genetic algorithms should be used. Planning methods, such as those discussed in Chapter 4, would be called for if the agent has to execute a

complex series of actions. Reinforcement learning could also be used to learn sequences of operations required to achieve a goal.

To summarize, all of the artificial intelligence techniques discussed in this book are applicable to intelligent agents. But, they are just tools which must be used by a skilled designer to craft a solution that meets the needs of a specific application. In the example Java code, we provide the simple building blocks to start your exploration of intelligent agents. In Part 2, we will combine these basic elements into an architecture for producing more powerful behavior. In the next sections, we look at specific agent capabilities, perception and action, and discuss the requirements they place on intelligent agent implementations.

Perception

In order for a software agent to take some intelligent action, it first has to be able to perceive what is going on around it, to have some idea of the state of the world. For animals, this problem is solved by the senses of touch, smell, taste, hearing, and sight. The next problem is to not get overwhelmed by the constant stream of information. Because of the large amount of raw sensory input we get, one of the first things humans learn is to filter out and ignore inputs that are expected or usual. We develop an internal model of the world, of what the expected consequences are when we take an action. As long as things are going as expected, we can get by without paying too much attention (for example, someone driving a car while talking on a cellular phone). But we also have learned to focus immediately on unexpected changes in the environment, such as a car appearing in our peripheral vision while entering an intersection.

Our intelligent agent must have an equivalent source of information about the world in which it lives. This information comes in through its *sensors*, which may or may not be grounded in the physical world. Our intelligent agent does not need to have senses in the same way we do, but it still has to be able to gather information about its environment. This could be done actively, by sending messages to other agents or systems, or it could be done passively by receiving a stream of event messages from the system, the user, or other agents. Just like people, the software agent must be able to distinguish the normal events (mouse movements) from the significant events (double-click on an action icon). In a modern GUI environment such as Windows or Macintosh, the user generates a constant stream of events to the underlying windowing system. Our agents can monitor this stream and must recognize sequences of basic user actions (mouse movement, pause, click, mouse movement, pause, double-click) as signaling some larger-scale semantic event or user action.

If our agent works in the e-mail or newsgroup monitor domain, it will have to recognize when new documents arrive, whether the user is interested in the subject matter or not, and whether to interrupt the user at some other task to inform him of the newly available information. All of this falls into the realm of perception. Being able to notice or recognize information hidden in data is not easy. It requires intelligence and domain knowledge. Thus being called “perceptive” is a compliment usually reserved for intelligent people. To be useful personal assistants, agents must be perceptive.

Of course, we can design our agents and their messages in such a way that they don’t have to be *very* perceptive. We can require the user to explicitly tell our agent what to do and how to do it. The events that are generated could contain all the information the agent needs to determine the current state and the appropriate action. However, one of the major reasons we want and need intelligent agents is to free the user from having to be so explicit. Spelling out the sequence of actions necessary for a computer to complete a task, in gory detail, is not fun (it’s programming!). That is the power and pleasure of delegating to others. You can say, “schedule a trip to Raleigh,” and your agent gets it done.

If perception is the ability to recognize patterns, and our agent receives its inputs as streams of data, then we are entering the realm of machine learning and

pattern recognition. Learning is not only useful for adapting behavior, it is also quite handy when we want to recognize changes in our environment. A learning system can associate certain states of the world with certain situations. These situations can be normal and expected, or they can be novel and unexpected. Simply logging the occurrence of events and then computing the likelihood or probability of an event or situation can add useful function to an agent. Building a table or map between these states and the associated action is even better. Supervised learning algorithms, such as neural networks and decision trees, can be used to automatically make these associations. Once again, if we know all of the situations our agent can get into before we send it out into cyberspace, then we can just build this into its knowledge base. But in a complex environment with a large number of possible situations, learning may be the only way to go.

Action

Once our agent has perceptively recognized that a significant event has occurred, the next step is to take some action. This action could be to realize that there is no action to take, or it could be to send a message to another agent to take an action on our behalf. Like people, agents take actions through *effectors*. For people, an effector is our muscles, when we take physical action in the world. Or we can take action through speech (using different muscles) or by sending e-mail (different muscles again). By communication with other people, we can cause them to act and change the environment.

But suppose we take an action by giving a command to the operating system or to an application. Delete a file. Send e-mail. Do we assume that the action has completed? Do we change our current model of the state of the world? What if we ask another agent to do something for us? Is it prudent to think that everything will work correctly (the agent is trustworthy and bug-free, the network server is up, the transaction goes across the wire)? Or, do we take the cynical view that Murphy's Law is always in effect? The answer is, "it depends." Just as we know that if we put a book down, it will be there the next time we walk by, if our agent takes an action directly under its control, we can probably consider it done. But when we are dealing with intermediaries, whether other agents or unknown systems, then some extra precautions and checking are probably in order.

Multiagent Systems

So far in this chapter, we have looked at what a single intelligent agent must be capable of to help a user get her work done. But, just as we have companies where the unique talents of many people are combined to solve problems, we can have multiple intelligent agents working together toward a common goal.

Whenever two autonomous, moderately intelligent beings try to communicate, problems arise. First, they have to speak the same language that uses the same set of tokens or symbols. Then, they must agree on what those symbols mean. They must also develop a way to exchange communications in that language. They can't both talk at the same time. They can't both try writing on the same page at the same time. These are all fundamental problems in communication, and the first artificial intelligent applications that tried to use multiple agents had to deal with and solve each of these issues. In the next section, we describe blackboard systems, the first multiagent AI application architecture.

Blackboards

While blackboards and chalk are now somewhat out of fashion, replaced by whiteboards and dry erase markers, they are still a useful metaphor for an important artificial intelligence architecture. The blackboard architecture was first introduced in the Hearsay II speech recognition project (Erman et al. 1980). It featured a system with multiple *knowledge sources* or independent agents, each with a specific domain of expertise related to speech analysis. The *blackboard* is a data structure that is used as the general communication mechanism for the multiple knowledge sources and is managed and arbitrated by a *controller* (Jagannathan, et al. 1989).

As each agent works on its part of the problem, it looks to the blackboard to pick up new information posted by other agents, and they, in turn, post their results to the blackboard. So, much like a blackboard in a classroom environment, the blackboard is an information-sharing device, with multiple writers and multiple readers. The agents, like students, each work at their own pace on the problems that are of most interest to them or where they have knowledge to apply, and add information to the blackboard when they can. Other agents use this information to further their own work. Thus, the blackboard architecture allows multiple agents to work independently and cooperate to solve a problem.

The blackboard model is a generalization of the architecture used in Hearsay II. It is an opportunistic problem-solving technique because at each control cycle any kind of reasoning method can be used. An event model is used to signal when changes are made to the blackboard and to notify the knowledge sources or agents that something has changed. An event could trigger the activation of a set of agents or the controller could dynamically determine which agent to start. The controller also limits access to the blackboard so that two agents don't try to write on the same space at the same time.

One final point regarding blackboard systems is that the knowledge sources or agents are very tightly coupled through the blackboard data structure and its interfaces. If you are building a single large application and want to modularize the knowledge bases, then blackboards are fine. However, if you want an environment where agents with very different structures and with no knowledge of a blackboard can work together, we need more formal interfaces such as those described in the next section.

Communication

When our agents need to talk to each other, they can do this in a variety of ways. They can talk directly to each other, provided they speak the same language. Or they can talk through an interpreter or facilitator, providing they know how to talk to the interpreter, and the interpreter can talk to the other agent.

There is a level of basic language—the syntax and format of the messages, and there is a deeper level— the meaning or semantics. While the syntax is often easily understood, the semantics are not. For example, two English-speaking agents may get confused if one talks about the boot and bonnet, and the other about the hood and trunk of an automobile. They need to have a shared vocabulary of words and their meaning. This shared vocabulary is called an ontology.

KQML

The Knowledge Query and Manipulation Language (KQML) provides a framework for programs and agents to exchange information and knowledge.

Like KIF, KQML (these acronyms sort of roll off the tongue, don't they?) came out of the DARPA Knowledge Sharing Effort. Whereas KIF deals with knowledge representations, KQML focuses on message formats and message-handling protocols between running agents. But only the message protocols are specified. The content or language used to represent the content is not. KQML defines the 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. Facilitators act as matchmakers or secretaries for the agents they service.

KQML messages are called performatives. Each message is intended to implicitly perform some specified action. There are a large number of performatives defined in KQML, and most agent-based systems support only a small subset. The performatives, or message types, are reserved words in KQML. Using performatives, agents can ask other agents for information, tell other agents facts, subscribe to the services of agents, and offer their own services.

KQML uses ontologies, explicit specifications of the meaning, concepts, and relationships applicable to some specific domain, to insure that two agents communicating in the same language can correctly interpret statements in that language. For example, in English, when we say "Java," are we referring to coffee, an island, or a programming language? To eliminate this ambiguity, every KQML message explicitly states what ontology is being used.

KQML messages encode information at three different architectural levels: content, message, and communication. An example of a KQML message from agent *joe* asking about the price of a share of SUN stock might be encoded as:

```
(ask-one
  :sender joe
  :content (real price = sun.price())
  :receiver stock-server
  :reply-with sun-stock
  :language java
  :ontology NYSE-TICKS)
```

The KQML performative is *ask-one;* the receiver of the message is an agent named *stock-server*. The *:content* parameter completely defines the content level. The *:reply-with*, *:sender*, and *:receiver* parameters specify information at the communication level. The performative name, the *:language* specification,

and the *:ontology* name are part of the message level. Note that we could send the exact same message with a different language and content piece, or the same language but different ontology, and thus subtly or dramatically change the meaning of the message. But the performative, *ask-one*, would still be a query from agent *joe* to agent *stock-server*.

Two agents who want to communicate using KQML require the services of a KQML facilitator or matchmaker (Kuokka and Harada 1995). The agents communicate with the matchmaker using standard KQML messages. They can register themselves as a provider of services or information using the *advertise* performative. Agents can also ask the matchmaker to recommend other agents using the *recommend*, *recruit*, and *broker* performatives. The matchmaker provides a centralized meeting-place for agents, and establishes a community where agents can interact. Note that the matchmaker cannot vouch for the trustworthiness of the agents that advertise their services, nor guarantee that an agent can provide a specific piece of information. Nevertheless, the matchmaker plays an important role in a multiagent system. The alternative would be for every agent to query every other agent whenever it needed to collaborate.

Cooperating Agents

The case for having multiple agents cooperate to do a job is compelling. Why have one agent slave away when we can have fifty working together to accomplish the same task? What's more, why have one huge artificial intelligence knowledge base and application if we can solve the same problem by constructing and maintaining a collection of much smaller and simpler agents? Of course, the idea of breaking a problem into smaller pieces was not an invention of artificial intelligence. Divide and conquer is an ancient military strategy. In computer science, this was one of the basic tenets of top-down structured design, used in everything from COBOL to Pascal. However, with agents, the advantages of the small, independent knowledge sources must be weighed against the considerable disadvantages of introducing a language barrier between them. But in several domains, the benefits have far outweighed the costs.

As client/server and distributed computing models have evolved over the last two decades, problems with system management became apparent. System managers, who were used to performance monitors which provided clear views of the state of the centralized host, had a difficult time handling applications that

ran across networks of workstations. The solution was to deploy system monitor agents on each of these distributed systems. A centralized controller could communicate (usually by polling) with the remote monitor's agent to get information on the status of each remote computer. The distributed system management application could integrate that information to provide an overview of the current state of the system. While starting out as very simple agents, these monitor agents have grown more sophisticated over time.

Previous | Table of Contents | Next

In electronic commerce (e-commerce), the intelligent agents represent both consumers and providers of products and services. In a simple form, an agent may represent a traveler who wants to book a flight, and who queries various airline reservation systems to find the best price and itinerary. In a more complex form, the buying agent must deal with a selling agent, who has its own agenda (to make as much money as possible per transaction for its owner) and strategies. In this marketplace context, the agents are cooperating to serve the interests of both parties. At the same time, there is an undercurrent of competition. The Kasbah system developed at MIT Media Lab is an example of this kind of electronic marketplace (Chavez and Maes, 1996).

Multiagent design systems are another area where multiple independent agents have proved their value (Lander 1997). In a collaborative design system, many designers must work on overlapping pieces of the design at the same time. A change in one part may impact several others. The entire design system includes the human designers as well as the intelligent agents, so human-computer interactions are important. Often, personal assistant agents are used to provide the interface between the designers and the agents in the system. A important aspect of multiagent design systems is that each intelligent agent has its own perspective on what a “good” or “optimal” design is, depending on its area of expertise. This is not very different from cross-functional development teams. Thus, each agent must be willing to compromise in order to achieve a better overall or global design. This also means that somewhere there is an agent that can provide some measure of the quality of the overall design and can arbitrate conflicts between the more specialized agents.

Competing Agents

As soon as we have agents that do our bidding, other people will have agents to do theirs. Our agent wants to get the best deal for us. Other agents want to get the best deal for their owners. Whose agent wins? Probably the one with the most intelligence, the most specialized knowledge about the task it is trying to perform, the most powerful reasoning system to apply that knowledge to problem solving in the domain, and, in all likelihood, the one that can learn from experience and get better over time.

This assumes of course that we have a level playing field. It wouldn't be fair if your agent was on a server negotiating with another agent that could access a large local database of information while your agent could not. Or, if your agent was scanned by the server, and information about your negotiating position (prices, strategies, rules, etc.) was placed in the hands of the other agent. Would you send your agent out on the Internet if you thought you could lose or be taken advantage of so easily? Probably not. In our opinion, there will have to be secure agent marketplaces, where agents register and are guaranteed to have equal access to server services and information.

There is also competition between agents at a less direct level. For example, suppose two knowledge workers at competing companies are both trying to finish a project proposal for a customer. If one has a set of intelligent agents that can help him find the information required in half the time as his competitor, that gives him a competitive advantage. This is just an extension of the current use of information technology to help a company win in the marketplace. When everyone has intelligent agents working for them, the winner will be the one with the best agents.

Summary

In this chapter, we explored how artificial intelligence techniques are used to build intelligent agents. The main points are:

- There are many alternative *artificial intelligence techniques* for knowledge representation, reasoning, and learning. The specific functions and requirements of an intelligent agent are the prime determinant of which AI techniques should be used.
- The amount of *intelligence* required by an agent, in terms of the size of the knowledge base and sophistication of the reasoning algorithms, is significantly impacted by the degree of *autonomy* and *mobility* the agent has. Mobile agents place special requirements on the security of the knowledge base as it travels through the network.
- *Learning* is most useful when an agent is used in complex environments to perform repetitive tasks, or when the agent must adapt to unknown situations. Personal assistants that model user preferences and collaborative agents will also benefit from learning capabilities.
- Agents must be able to perceive the physical or virtual world around them using *sensors*. A fundamental part of perception is the ability to recognize

and filter out the expected events and attend to the unexpected ones.

- Intelligent agents use *effectors* to take actions either by sending messages to other agents or by calling application programming interfaces or system services directly.
- The *blackboard architecture* allows a group of agents to cooperate to solve problems. The *blackboard* is a centralized data structure used for communication and data sharing between agents. The *controller* manages the blackboard as well as the generation and dispatching of events to the agents in the system.
- *Knowledge Query and Manipulation Language* (*KQML*) provides a framework for a set of independent agents to communicate and cooperate on a problem using messages called *performatives*. KQML specifies information at the *communication level* (sender and receiver), the *message level* (language and ontology), and the *content level* (language-specific sentences).
- *Cooperation* among agents allows a community of specialized agents to pool their capabilities to solve large problems, but with the additional cost of communication overhead. Distributed systems management, electronic commerce, and multiagent design systems are three application areas where cooperating agents have been applied.
- *Competition* between agents will occur as soon as intelligent agents are deployed by individuals or companies with different agendas, and those agents interact in the e-commerce environment. Intelligent agents will be used to provide competitive advantages for individuals and businesses.

Exercises

1. How does artificial intelligence relate to intelligent agents in terms of their perceptiveness and action-taking abilities? Is AI essential or superfluous?
2. What role does the controller play in a blackboard system?
3. What are the strengths of KQML as an agent communication language? The weaknesses?
4. What infrastructure would be required to set up an auction or marketplace for use by multiple intelligent agents?

Chapter 7
Intelligent Agent Framework

In this chapter, we develop an intelligent agent architecture using object-oriented design techniques. We start with a generic set of requirements and refine them into a set of specifications. We explicitly state our design philosophy and goals and consider various design and implementation alternatives under those constraints. We explore how intelligent agents can be used to expand the capabilities of traditional applications and how they can serve as the controller for a group of applications. With minor modifications, we reuse the artificial intelligence functions we developed in Java in Part 1 of this book.

Requirements

The first step in any software development project is the collection of requirements from the intended user community. In our case, this is made difficult because our readers cannot provide this kind of feedback until we have already designed and developed the product (this book). However, we have made some obvious decisions concerning the audience and intended purpose of this book, as indicated by the title. We are going to develop intelligent agents using Java. We are also going to provide the ability to add intelligence to applications or applets written in Java. An additional fundamental requirement is that we should reuse the artificial intelligence code we developed in Part 1.

Another requirement is that our intelligent agent framework be practical. Not practical in the sense that it is product-level code ready to put into production, but in that the basic principles and thrust of our design is applicable to solving real-world problems. While providing a stimulating learning experience is a goal, we are not interested in exploring purely academic or, more accurately, esoteric issues. If you understand what we are doing and why we are doing it, you should be able to use these techniques to develop your own intelligent agent applications.

Focus on the topic at hand is another requirement. This is a book about

intelligent agents. We would be doing the reader a disservice if we spent large amounts of time developing communications code, an object-oriented database, or a mechanism for performing remote procedure calls. We will try to maximize the amount of code dealing with intelligent agents, and minimize the code not directly related to the topic. At the same time, because this is a book about Java programming, we want to use the features and capabilities found in Java and the JDK 1.1 development environment.

Having just said all this, we also acknowledge that providing a decent user interface is also a requirement. Luckily, with the current Java development environments, providing a usable GUI is not a major problem. We will use the Symantec Visual CafŽ tool to create these interfaces. Because most of the GUI code is generated, we will not spend much time or energy discussing this aspect of our applications. There are certainly other visual builder tools that can generate Java code and they could be used instead of the Symantec product.

The next requirement comes from the authors, who also spend much time reading programming books. We will not create a complex design and describe it in minute detail. While this would be an interesting exercise for us, we doubt it would provide value to you, our readers. We'll call this the "keep it simple, stupid" requirement, and hope that explicitly listing it here will shame us into following this maxim when we get too carried away.

The last requirement is that our architecture must be flexible enough to support the applications presented in the next three chapters. As a preview, we will be building intelligent agents to handle management of a PC, information filtering over the Internet, and simple multiagent electronic commerce transactions.

To summarize the requirements, we want a simple yet flexible architecture that is focused on intelligent agent issues. It must be practical so it can solve realistic problems and must have a decent user interface so its functions and limitations will be readily apparent to the users. In the next section, we talk about our goals from a technical perspective.

Design Goals

Requirements come from our users and tell us what functions or properties our product must have in order to be successful. Having a validated set of requirements is useful, because it focuses our energy on the important stuff. It is

just as important to have a clear set of design goals which we can use to guide the technical decisions which must be made as we develop the solution that meets those requirements. Just as with requirements, we must explicitly state our design goals and assumptions.

There are some fundamental issues which will drive our design. The first is that we can view our intelligent agents either as adding value to a single standalone application, or as a freestanding community of agents which interact with each other and other applications. The first is an application-centric view of agents, where the agents are helpers to the application (and therefore of the users of the application). This approach is the least complex because we can view the agent as a simple extension of the application functionality. By providing our intelligent agent functions as an object-oriented framework, we can easily add intelligent behavior to any Java application.

The second approach is more agent-centric, where the agents call the shots and monitor and drive the applications. Here our agent manager is an application in its own right and must interface with other applications which are driven by the agents. The complexity here is that we must define a generic mechanism for application communications through our agent manager. One method is to require every application to modify its code to be “agent-aware.” Another is for us to provide a common way to interface with the unique application programming interfaces.

The CIAgent framework must be easily understood and straightforward to use. The primary aim of developing this framework is to illustrate how intelligent agents and the different AI techniques can be used to enhance applications. Our coding style will be straightforward. Data members will be public and accessor functions will be used only when strict encapsulation is required. This is not commercial-level code. Bullet-proofing code can sometimes make even simple logic seem complex. Our applications will work as designed, but they will not be able to handle all unexpected input data or error conditions.

We will construct the agent framework so that interagent communication can be supported as well as the mobility of our agents across networks. Flexibility also includes the ability to easily add support for new applications, AI techniques, and other features.

Functional Specifications

In this section, we take the requirements and our design goals, and turn them into a list of functions which satisfy those requirements and goals. This defines what we have to build. The functional specifications are a contract between the development team and the user community. Here is the functionality we think we need:

1. It must be easy to add an intelligent agent to an existing Java application.
2. A graphical construction tool must be available to compose agents out of other Java components and other agents.
3. The agents must support a relatively sophisticated event-processing capability. Our agent will need to handle events from the outside world, other agents, and signal events to outside applications.
4. We must be able to add domain knowledge to our agent using if-then rules, and support forward and backward rule-based processing with sensors and effectors.
5. The agents must be able to learn to do classification, clustering, and prediction using learning algorithms.
6. Multiagent applications must be supported using a KQML-like message protocol.
7. The agent should be persistent. That is, once an agent is constructed, there must be a way to save it in a file and reload its state at a later time.

Intelligent Agent Architecture

Now that we have specified the functions that our intelligent agent architecture must provide, we must make our design decisions. We will take the function points in order and discuss the various issues and tradeoffs we must make.

1. It must be easy to add an intelligent agent to an existing Java application.
The easiest way for us to add an agent to an existing application is to have the application instantiate and configure the agent and then call the agent's methods as service routines. That way the application is always in control,

and it can use the intelligent functions as appropriate. This is easy, but this is hardly what we would consider an intelligent agent. It is embedded intelligence, but there is no autonomy. Another possibility is to have the application instantiate and configure the agent and then start it up in a separate thread. This would give the agent some autonomy, although it would be running in the application's process space. The application could yield to the agent when necessary, and the agent would yield when it was done processing so that the application could continue. A third possibility is to have the agent run in a separate thread, but use events to communicate between the application and agent.

In Java, this can be done using the Observer/Observable framework. The agent would be an Observer, and whenever the agent was notified that an event occurred, it would be executed. A disadvantage of this approach is that the application would have to be a subclass of Observable.

2. A graphical construction tool must be available to compose agents out of other Java components and other agents.

There are graphical development tools such as Symantec Visual CafŽ and IBM's VisualAge for Java that allow you to construct applications using a "construction from parts" metaphor. However, the JDK 1.1 release of Java provides a basic component capability through its *java.beans* package. JavaBeans is a Java component model that allows software functions to be treated as "parts" which can be put together to construct an application. Each JavaBean has a well-defined interface which allows a visual builder tool to manipulate that object. It also has a defined runtime interface which allows applications comprised of JavaBeans to run.

Another nice feature of Beans is that they can be nested. This meets our requirement for the ability to compose agents out of other agents. This allows us to develop special-purpose agents which can be reused in other higher-level agents. For example, we can have low-level agents which use neural networks for learning and high-level agents which use rules to determine what actions to take. This function is roughly equivalent to the Composite design pattern as specified by Gamma et al. (1995).

The BeanBox is part of the Bean Development Kit (BDK) and provides a rudimentary, but effective, default graphical environment for working with Beans. We can use the BeanBox as our visual construction environment.

3. The agents must support a relatively sophisticated event-processing capability. Our agent will need to handle events from the outside world, other agents, and signal events to outside applications.

The JDK 1.1 release features a powerful new event-processing model called

the Delegation Event Model. This new framework was actually driven by the requirements of the JavaBeans component model. This model is based on event **sources** and event **listeners**. There are many different classes of events with different levels of granularity. We will use the JavaBeans event model in our agent framework.

4. We must be able to add domain knowledge to our agent using if-then rules, and support forward and backward rule-based processing with sensors and effectors.

The **RuleBase**, **Rule**, and **RuleVariable** classes we developed in Chapter 4 can be used in our agents to provide forward and backward rule-based inferencing. We will have to extend the functionality to support sensors and effectors. The main reason for providing this functionality is to provide a nonprogramming way for users to specify conditions and actions. If we build a JavaBeans property editor for our RuleBase class, any user could easily construct a set of **RuleVariables** and **Rules** to perform the logic behind an intelligent agent's behavior. We provide this functionality, but do not exploit it. That is, the task of developing a property editor for use in a visual Bean development environment is left as an exercise for the reader.

5. The agents must be able to learn to do classification, clustering, and prediction using learning algorithms.

In Chapter 5, we designed and developed decision tree classifiers, and neural clustering and prediction algorithms in Java. We can use the **DecisionTree**, **BackPropNet**, and **KMapNet** classes to provide these functions to our agents.

6. Multiagent applications must be supported using a KQML-like message protocol.

In order to provide this functionality, we will have to go back to the drawingboard, and come up with an agent that can handle tasks like a KQML facilitator or matchmaker. We would like to use our existing rule capabilities to help provide this function, if possible. We can use the JavaBean event model to provide the communication mechanism between agents and the facilitator and define our own event objects to hold the message content.

7. The agent should be persistent. That is, once an agent is constructed, there must be a way to save it in a file and reload its state at a later time.

The 1.1 JDK supports serialization of Java objects. In addition, the JavaBeans framework makes the saving and loading of objects very easy to do. Actually, any data members which are not explicitly declared as static or transient will be saved out to a file. So here is another advantage of using

the JavaBean component model as our base.

Previous | Table of Contents | Next

The CIAgent Framework

While trying not to be too cute, we have selected the name **CIAgent** for our intelligent agent framework, where CIAgent stands for "*C*onstructing *I*ntelligent *Agents*." Many other names suggested themselves to us, but this seems like a reasonable choice, given the title of the book. If it really bugs you, we hope it doesn't interfere with your understanding and use of our design.

To summarize the decisions we made in the preceding section, we are going to construct our intelligent agents so they can interact with the JavaBeans component model. This design decision, combined with reuse of code from Part 1 of this book, allows us to meet most of our functional specifications. We have to make some enhancements to our rule processing, as well as develop a facilitator for our multiagent applications. However, we are well on our way to providing a usable intelligent agent framework.

The next step is to sketch out our class structure and interfaces. We begin with an abstract base class that defines the common interface used by the elements of our architecture.

The CIAgent Base Class

CIAgent is the name of the base class which defines a common programming interface and behavior for all the agents in our framework. In terms of design patterns, **CIAgent** uses a composite design. This means that we can use a single **CIAgent** or compose them into groups, and still treat the group as if it was a single logical **CIAgent** object. This design pattern is very powerful because it allows us to build up a hierarchy of CIAgents using other specialized **CIAgent** classes in the process.

The **CIAgent** class implements the Runnable interface, which requires that our **CIAgent** subclasses provide a *run()* method which can serve as the body of a **Thread**. This is the mechanism we use to give our agents autonomy. To communicate with other CIAgents and other JavaBeans, we implement the **CIAgentEventListener** interface. This interface extends the standard Java **EventListener** interface used by all AWT components and JavaBeans. Although

we do not extend any JavaBeans class, the **CIAgent** class is a JavaBean, by virtue of our **EventListener** interface and the public, zero-argument default constructor *CIAgent()* method. A **Vector** of listeners holds all Java objects which implement the **CIAgentEventListener** interface and which have registered themselves using the *addCIAgentEventListener()* method. Any **CIAgent** object can be the event source for **CIAgentEvents**, and any **CIAgent** object can be a registered listener for those events. The **CIAgent** class provides *addCIAgentEventListener()* and *removeCIAgentEventListener()* methods so that other **CIAgents** can be added to the multicast event notification list. These methods fully support the JavaBeans event API, so CIAgents can be wired up using the BeanBox or any visual builder tool that supports JavaBeans.

The *notifyCIAgentEventListeners()* method is used to send events to registered listeners. Note that, like the *addCIAgentEventListener()* and *removeCIAgentEventListener()* methods, *notifyCIAgentEventListener()* must be synchronized to control access to the listener's **Vector** in a multithreaded environment.

Each **CIAgent** has a **String** member or property for its *name*, and we implement the standard JavaBean methods for setting and getting the *name* through the BeanBox or other visual builder tool. We use the JavaBean **PropertyChangeSupport** class to make the *name* a bound property. When the *name* is changed, other listeners will be notified.

The other methods we define include *initialize()* and *reset()* methods for getting the agent to a known state, and *process()* and *stop()* methods for starting the agent-processing thread or stopping it. Note that we could have used the Java *init()* method as defined in the **Applet** class, but we chose to avoid any name collisions. In fact, we could easily have extended our **CIAgent** class from **Applet** to make all of our CIAgents applets. The CIAgents provided in this book could be easily turned into applets or by subclassing an **awt.Component** they could be made into visible JavaBeans. As implemented here, our CIAgents are invisible Beans, meaning that they can be used in the visual builder environment, but they do not represent graphical components in the application GUI.

The Java implementation of our **CIAgent** class follows. It contains members and methods for the management of **EventListeners** and event processing, object constructors, support for tracing utilized by our agent applications in the following chapters, and the agent **Thread** support.

```
public class CIAgent implements CIAgentEventListener, Runnable {

 private Vector listeners = new Vector() ; // list of listeners


 public synchronized void
   addCIAgentEventListener(CIAgentEventListener ciaEventListener) {
          listeners.addElement(ciaEventListener) ;
 }

 public synchronized void
   removeCIAgentEventListener(CIAgentEventListener ciaEventListener
 {
          listeners.removeElement(ciaEventListener) ;
 }

 protected void notifyCIAgentEventListeners() {
    Vector l ;
    CIAgentEvent e = new CIAgentEvent(this) ;

    synchronized(this) { l = (Vector)listeners.clone(); }
    for (int i=0 ; i < l.size() ; i++) { // deliver the event
      ((CIAgentEventListener)l.elementAt(i)).ciaEventFired(e);
    }
 }

 // deliver the ciagent event to registered listeners
 protected void notifyCIAgentEventListeners(CIAgentEvent e) {
    Vector l ;
    synchronized(this) { l = (Vector)listeners.clone(); }
    for (int i=0 ; i < l.size() ; i++) { // deliver the event
      ((CIAgentEventListener)l.elementAt(i)).ciaEventFired(e);
    }
 }

 public  void ciaEventFired(CIAgentEvent e) {
     System.out.println("CIAgent: CIAgentEvent received by " +
                        name + " from " + e.getSource() +
                        " with args " + e.getArgObject()) ;
 }

 String name ;          // the agent's name
 public String getName() { return name ; }
 public void setName(String Name) {
   String oldName = name ;
   name = Name ;
   changes.firePropertyChange("name", oldName, name);
 }

 private PropertyChangeSupport changes =
```

```
                        new PropertyChangeSupport(this);

  public CIAgent() { name = “CIAgent”; }
  public CIAgent(String Name) { name = Name ;
                       }

  // used for tracing and display of agent status
  Object market ;
  int traceLevel=0 ;   //  0 = summary, 1 = detailed
  TextArea textArea ;
  public void setDisplay(Object mkt, int trace) {
    market = mkt ;  traceLevel = trace ;
  }

  // used for displaying message in our applications
  public synchronized void trace(String msg) {
    // comment this line out if using this outside Marketplace app
    if (market != null) ((Marketplace)market).trace(msg) ;
    if (textArea != null) textArea.appendText(msg) ;
  }

  public void process() {};   // start the agent processing thread

  public  void reset() {};    // reset the agent to a known state

  public  void initialize() {};  // initialize the agent

  // required by Runnable interface and Threads
  Thread runnit = new Thread(); // start()ed in the process() method
  boolean stopped = false; // control flag
  public void stop() {} ;  // stop the agent thread
  public void run() {} ;   // body of thread

};
```

Previous | Table of Contents | Next

CIAgentEvent

The **CIAgentEvent** is derived from the Java **EventObject** class, as required by the JavaBeans specification. The *CIAgentEvent()* constructors take either a single parameter, a reference to the object sending the event, or a pair of parameters defining the source and an event argument object. We define this as an **Object** so that subclasses of **CIAgent** can send any object as an argument in a **CIAgentEvent**. We cannot know in advance what will be needed in a subclass. In Chapter 10, we will use this flexibility to define a KQML-like message object.

```
public class CIAgentEvent extends java.util.EventObject  {
   Object argObject ;

   public Object getArgObject() { return argObject; }

   // for extremely simple events
   CIAgentEvent(CIAgent source) {
       super(source) ;
   } ;

   // for more complex events
   CIAgentEvent(CIAgent source, Object arg) {
     super(source) ;
     argObject = arg ;
   }

};
```

CIAgentEventListener

The **CIAgentEventListener** interface extends the **EventListener** interface and requires that a single method be implemented by the **CIAgent** class, *ciaEventFired()*. This method takes a **CIAgentEvent** object as the only parameter.

```
public interface CIAgentEventListener extends java.util.EventListene
{
      void ciaEventFired(CIAgentEvent e) ;
};
```

RuleBase Enhancements

In this section, we describe the enhancements to our rule classes. We include support for sensors and effectors in rules, and for facts as part of the rule base. We define two Java interfaces, **Sensor** and **Effector**, to support this function. We also extend the **Clause** class with a **SensorClause** and **EffectorClause**. Our support is as follows:

> if **sensor**(sensorName, RuleVariable) then **effector**(effectorName, parameters)

where **sensor** is an instance of **SensorClause**, and **effector** is an instance of **EffectorClause**. The **SensorClause** makes a call to a sensor method which is defined by any class that implements the **Sensor** interface and registers it with the **RuleBase**. At runtime the **RuleBase** looks up the *sensorName* and calls the method on the registered sensor object. A similar technique is used for the effectors.

In extending the behavior of the **Rule** class to support the **SensorClause** and **EffectorClause** classes, we uncovered a case where we broke the rule of encapsulation of knowledge about implementation. In the *Rule.display()* method, the **Clause** was formatted for display. This code included references to a *lhs*, *condition*, and *rhs* in the **Clause**. However, our **SensorClause** and **EffectorClause** do not set these data members so the *Rule.display()* method failed. The solution to this is to define a *display()* method on the **Clause** class and its subclasses and for *Rule.display()* to use that method. This encapsulates the knowledge about the **Clause** and how it should be formatted for display.

To support facts, we add a new class called **Fact** whose constructor takes a single clause as a parameter. A fact can be an assignment of a value to a **RuleVariable**, a sensor call, or an effector call. The **Facts** are defined as part of the **RuleBase** with the other Rules. But the Facts are also registered in the **RuleBase**. The *initializeFacts()* method is called to set the facts before an inferencing cycle is performed.

```
public class Fact {
  RuleBase rb ;
  String name ;
  Clause fact ;     //only 1 clause allowed
  Boolean truth;       // states = (null=unknown, true, or false)
```

```
  boolean fired=false;
  Fact(RuleBase Rb, String Name, Clause f) {
    rb = Rb ;
    name = Name ;
    fact = f ;
    rb.addFact(this) ;  // add self to fact list
    truth = null ;
  }

  // assert the fact
  public void assert(RuleBase rb) {
   if (fired == true) return ; // only assert once
   RuleBase.appendText("\nAsserting fact " + name ) ;
   truth = new Boolean(true) ;
   fired = true ;
   if (fact.lhs == null) {
      // it's an effector
      ((EffectorClause)fact).perform(rb);  // call effector method
   } else {
      // set the variable value and update clauses
      fact.lhs.setValue(fact.rhs) ;
      // now retest any rules whose clauses just changed
   }
 }

  // display the fact in text format
  void display(TextArea textArea) {
    textArea.appendText(name +": ") ;
    textArea.appendText(fact.display() + "\n") ;
  }

};
```

The **Effector** interface allows any **Class** to be called as an effector by a **Rule** in a **RuleBase**. It must implement the interface by providing a single *effector()* method that takes an **Object**, an effector name (for cases where the implementing class may support multiple effectors), and a **String** argument parameter. An alternate implementation that may be useful would be to support an **Object** as the argument parameter.

```
public abstract interface Effector  {

  public long effector(Object obj, String eName, String args) ;
};
```

The **EffectorClause** class extends **Clause** by adding members for the name of the effector to call, a data member containing a reference to the object which

implements the *effector()* method, and a **String** of arguments. The *EffectorClause()* constructor takes two parameters, the name of the effector to call, and the arguments **String**. The *perform()* method is provided to call the *effector()* method on the object that has registered to provide that effector function.

```
public class EffectorClause extends Clause {
    Effector object ;       // object to call
    String effectorName ; // method to call
    String arguments ;  // parameters to pass

    EffectorClause(String eName, String args)
    {
     ruleRefs = new Vector() ;
     truth = new Boolean(true);    // always true
     consequent = new Boolean(true);  // must be consequent
     effectorName = eName ;
     arguments = args ;
    }

    public String display() {
        return “effector(” + effectorName + “,” + arguments + “) ” ;
    }

     // call the effector method on the target object
    Boolean perform(RuleBase rb) {
         object = (Effector)(rb.getEffectorObject(effectorName)) ;
         object.effector(this, effectorName, arguments) ;
         return truth ;  // always true

    }
};
```

Previous | Table of Contents | Next

The **Sensor** interface consists of a single method *sensor()* which takes three parameters: the object (**SensorClause**) which called the method, the name of the sensor to use, and a **RuleVariable** in which to store the results, if any. The *sensor()* method returns a boolean value.

```
public abstract interface Sensor {
  public Boolean sensor(Object obj, String sName, RuleVariable lhs)
};
```

The **SensorClause** class extends **Clause** by adding an object data member that implements the **Sensor** interface and a **String** that specifies the *sensorName*. A *SensorClause()* takes two parameters: the name of the sensor to test and a **RuleVariable** to hold the truth value. The *check()* method is called by a **Rule** to evaluate the truth value of the **SensorClause**.

```
public class SensorClause extends Clause {
     Sensor object ;
     String sensorName ;

     SensorClause(String sName, RuleVariable Lhs)
    {
     lhs = Lhs ;
     cond = new Condition("=") ;
     rhs = " ";
     lhs.addClauseRef(this) ;
     ruleRefs = new Vector() ;
     truth = null ;
     consequent = new Boolean(false) ;
     sensorName = sName ;
    }

   public String display() {
        return "sensor(" + sensorName + "," + rhs + ") " ;
    }

    Boolean check() {
       if (consequent.booleanValue() == true) return null ;

       if (lhs.value == null) {
       RuleBase rb = ((Rule)ruleRefs.firstElement()).rb ;
       object = (Sensor)(rb.getSensorObject(sensorName)) ;
       truth =  object.sensor(this, sensorName, lhs) ;
       }
       return truth ;
```

```
    }
};
```

The following code fragment gives an example of how sensors and effectors could be used in the Vehicles RuleBase:

```
Rule EffectorTest = new Rule(rb, “EffectorTest”,
     new Clause(num_wheels, cEquals, “4”),
     new EffectorClause(“display”, “It has 4 wheels!”)) ;

rb.addEffector( new TestEffector(), “display”) ;  // test it

RuleVariable sensor_var = new RuleVariable(“sensor_var”) ;
 sensor_var.setLabels(“2 3 4”) ;
 sensor_var.setPromptText(“What does the sensor say?”) ;
 rb.variableList.put(sensor_var.name,sensor_var) ;

Rule SensorTest = new Rule(rb, “SensorTest”,
     new Clause(num_wheels, cEquals, “4”),
     new SensorClause(“sensor_var”, sensor_var),
     new EffectorClause(“display”, “It’s an automobile!!!!”)) ;

 rb.addSensor( new TestSensor(), “sensor_var”) ;  // test it

Fact f1 = new Fact(rb, “f1”,
     new Clause(num_wheels, cEquals, “4”)) ;
Fact f2 = new Fact(rb, “f2”,
     new SensorClause(“sensor_var”, sensor_var)) ;
Fact f3 = new Fact(rb, “f3”,
     new EffectorClause(“display”, “Just the facts man !”)) ;
```

We use the effectors in the Marketplace application in Chapter 10.

Summary

In this chapter we described the CIAgent intelligent agent framework. The main points include:

- We described our requirements and high-level design goals and translated them into a set of *specifications*. These included:

    1. It must be easy to add an intelligent agent to an existing Java application.
    2. A graphical construction tool must be available to construct agents.
    3. Agents must support a relatively sophisticated event-processing

capability.

4. We must support forward and backward rule-based processing with sensors and effectors.

5. Agents must be able to learn to do classification, clustering, and prediction.

6. Multiagent applications must be supported using a KQML-like message protocol.

7. Agents must be persistent, making it possible to save them in a file and restore their state later on.

- We described the *CIAgent* base class, the *CIAgentEvent* class, and the *CIAgentEventListener* interface to provide support for the intelligent agents used in the applications in the following chapters.
- We extended our if-then rule processing function by adding the *Facts*, *SensorClause*, and *EffectorClause* classes, and the *Effectors* and *Sensors* interfaces. This allows our rules to call functions to test conditions and perform actions.

Exercises

1. What are some of the design decisions which limit the usefulness or applicability of the **CIAgent** architecture?

2. How could you extend the **CIAgent** architecture to implement an information filtering agent? a personal computer manager agent?

3. In the **CIAgent** architecture, we chose to use the JavaBeans delegation event model. How could you provide equivalent function using the Java **Observer/Observable** framework? What are the advantage and disadvantages of this approach compared to using JavaBeans event notification?

4. Compare your design from exercise 4.3 for adding sensors and effectors to the implementation provided in this chapter. Did you use any Java interfaces in your solution?

Chapter 8
PCManager Application

In this chapter, we illustrate how we can use the *CIAgent* architecture to construct an application that uses intelligent agents to assist a user with PC management and local application activities. Two simple autonomous intelligent agents are developed, one based on a timer and another that watches the PC file system. When a trigger event occurs, agents can signal each other, alert the user, or execute a system command.

Introduction

The first intelligent agent application we develop is a personal assistant designed to aid us in managing our personal computer. Our goal is to provide basic functionality while illustrating the general concepts required for this domain. We develop two new intelligent agents by extending the **CIAgent** base class introduced in Chapter 7. The **PCManager** application class provides a graphical user interface using the Java AWT framework, and provides the interface to TimerAgents and FileAgents to help us manage the resources in our PC.

The PCManager allows a user to specify two major functions, Alarms and Watches. Alarms use the **TimerAgent** to manage the time-keeping aspects of the function. An alarm is a time-based event that can occur at a single specific time or at repeating intervals. For example, we could set an alarm to go off at 6:00 A.M. tomorrow morning, or we could specify an alarm to go off every 10 minutes. Watches use the capabilities of a **FileAgent** to detect changes to the state of a file or directory in the PC file system. A watch could be set to signal when a file is changed or deleted, or when it grows over a specified size.

Figure 8.1 The PCManager application.

Figure 8.1 shows the main panel of the **CIAgent PCManager** application. The application interface consists of two parts. The top **awt.List** box displays the currently specified Alarms and Watches. The bottom **awt.TextArea** is used to display status and trace messages.

There are two major secondary panels or dialogs used in the PCManager application, the **AlarmDialog** and the **WatchDialog**, which are used to specify the parameters required for Alarms and Watches respectively.

Figure 8.2 shows the **AlarmDialog**. Each alarm has a name, a type, either one-shot alarm or interval, and an associated action. A one-shot alarm must have the time of the alarms specified in hours and minutes. The interval alarm requires the specification of the number of seconds between alarms. The user can select from one of three actions when the alarm fires. It can alert the user via an **AlertDialog**. It can execute a system command or run an application program on the user's behalf. Or, it can send a **CIAgentEvent** to another **CIAgent**. When the user clicks on **OK**, an instance of a **TimerAgent** is created and its methods are called to set the interval or date, trigger condition, and action. The Alarm name is used as the name of the **TimerAgent** instance.

Figure 8.3 illustrates the **WatchDialog**. Like an Alarm, a Watch has a name, a parameter to specify the file or directory to be watched, the trigger condition on the file, and the action to take when the trigger occurs. If the user does not know the filename or directory path, she can click on the **Browse** pushbutton and open a standard **awt.FileDialog** on the PC file system. The trigger conditions include if the file is deleted or modified, or if its size exceeds a specified threshold. Watches support the same three actions as Alarms, alerts, executing commands, and signaling events. When the user clicks **OK** on the **WatchDialog**, a **FileAgent** is instantiated and the user parameters are passed to the instance. The FileAgent's name is set to the Watch name.

Figure 8.2 The AlarmDialog panel.

Two additional dialogs associated with Alarm and Watch actions are used in this application. The **AlertDialog** displays an Alert message to the user. The

ExecuteDialog shows the results of an Execute action. Both action dialogs allow the user either to acknowledge them by clicking **OK**, or to cancel the Alarm or Watch with the **Cancel** button. These dialogs are shown in the example in the next section.

Figure 8.3 The WatchDialog panel.

An Example

The basic functions provided by Alarms and Watches can be combined to produce interesting behavior. In this section, two Alarms and a Watch are used to illustrate this by example. The Alarms will be used to copy one of two files into a target file. This target file will be the focus of a Watch. When all three agents are active in the **PCManager** application, the scenario is as follows:

1. Alarm1 copies the file *test1.dat* to *test.dat*.
2. Watch1 detects that the file *test.dat* was modified and signals an alert to the user.
3. Alarm2 copies the file *test2.dat* to *test.dat*.
4. Watch detects this change to the file, and signals another alert to the user.
5. This sequence repeats indefinitely.

The following figures show the parameter dialogs and the Alert and Execute dialogs generated by this example. Figures 8.4 and 8.5 show the parameters used to set up the TimerAgents for Alarm1 and Alarm2 respectively. Figure 8.6 shows the Watch1 settings.

Figure 8.4 Alarm1 TimerAgent parameters.

Figure 8.5 Alarm2 TimerAgent parameters.

When the three agents are created they immediately start running. Figure 8.7 shows the PCManager main panel when all of the agents are configured and running. As Alarm1 goes off, the **TimeAgent** executes the specified parameter string which copies the file *test1.dat* over *test.dat* (Figure 8.8). The Watch1 agent wakes up every 15 seconds to check out the status of the *test.dat* file. When it sees that the file has been modified (it uses the Java *File.getLastChanged()* method) it signals an Alert. This **AlertDialog** is shown in Figure 8.9.

Figure 8.6 Watch1 FileAgent parameters.

Figure 8.7 PCManager application example.

Now that you have seen an example of the PCManager in action, we take a closer look at the implementation of the two agents, and the **PCManager** application class.

Alarms: The TimerAgent

Our first agent, the **TimerAgent**, is not very intelligent but is certainly autonomous and provides an extremely useful function. A **TimerAgent** can be used to set one-shot alarms for any specified time, or it can be used to fire recurrent alarms at specified intervals. When an alarm condition occurs, the **TimerAgent** can take one of three actions. It can display an Alert, where a dialog pops up on the screen to inform the user that a Timer alarm has gone off. It can execute an arbitrary system command or invoke an application program on the system, passing parameters as required. The **TimerAgent** can also simply fire a **CIAgentEvent** to signal another **CIAgent** or other Java object.

Figure 8.8 Alarm1-generated ExecuteDialog.

Figure 8.9 Watch1-generated AlertDialog.

The **TimerAgent** has several data members. The *interval* holds a value, in milliseconds, for interval alarms. The *date* member holds the date and time for one-shot alarms. The *action* member defines which of the three actions the agent should take when the alarm goes off. The Thread member *runnit* and the boolean *stopped* are inherited from the **CIAgent** base class. The *process()* method is used to spin a new thread when the **TimerAgent** is started, and the *stop()* method is used to interrupt the **runnit** thread when it is sleeping (using the *Thread.sleep()* method in the *TmerAgent.run()* method).

We provide a set of accessor methods for each of these properties for use in the BeanBox or other JavaBean environment.

```
public class TimerAgent extends CIAgent {

  public final int ALERT = 0;    // display a dialog
```

```
  public final int EXECUTE = 1;  // start a process
  public final int EVENT = 2;    // send an event to another agent

  int action = EVENT ;   // default action is to send an event
  String parms ;
  int interval ;  // for interval timers
  Date time ;     // for one shot timers
  Dialog actionDialog ;

  public TimerAgent() { name = "Timer"; }
  public TimerAgent(String Name) { super(Name); } ;

  // return a string for display in a list box
  public String getDisplayString() {
    String out = new String() ;
    out = name + " int=" + (interval/1000) +
          " date=" + time + " action=" + action + "parms=" + parms ;
    return out ;
  }

 public void setInterval(int secs) {
        interval = secs * 1000 ;
 }

 public int getInterval() { return interval ; }
 public void setTime(Date t) { time = t ; }

 public Date getTime() { return time ; }
 public void setAction(int act) { action = act; }

 public int getAction() { return action ; }

 public void setParms(String params) { parms = params; }
 public String getParms() { return parms ; }

 public void setDialog(Dialog dlg) { actionDialog= dlg ; }

 public void process() {
    stopped = false ;
    // start a thread running
    trace("Starting " + name + "\n") ;
    runnit.start() ;
 }
 public void stop() {
    stopped = true;
    trace(name + " stopped \n") ;
    runnit.stop() ;
}

};
```

The *run()* method contains the body of the **TimerAgent** thread. It consists of two alternate parts, one for the interval timer and one for the one-shot timer. If an *interval* value is set, it enters a *while()* loop, where it goes to *sleep()* for the specified number of milliseconds, performs the desired *action*, and then repeats until the *stop()* method is called. If the *interval* member equals 0, then a one-shot alarm is set. A **Date** object is used to get the current date and time which is then used to calculate the amount of time until the one-shot timer goes off. The **TimerAgent** goes to *sleep()* once and wakes up at the appointed time, performs the desired action and exits the thread.

The *ciaEventFired()* method simply displays any events it receives from other CIAgents. We could easily enhance this method to allow other agents to start and stop timers using CIAgentEvents instead of method calls. However, this function is not required in the PCManager application.

Previous | Table of Contents | Next

The *performAction()* method contains a switch statement to handle the three possible actions. For an ALERT, the dialog text is specified using the *parms* **String**, and is displayed using the *show()* method. The **AlertDialog** allows the user to either acknowledge the alert and close the window by pressing the **OK** button, or to cancel the Alarm, thereby stopping the **TimerAgent** thread, by clicking on the **Cancel Alarm or Watch** button. The EXECUTE case calls the *execCmd()* method which starts up a new process and passes the *parms* **String** to the system command processor for execution. Any valid system command or application name with parameters can be specified in this string. An **ExecuteDialog** is displayed and the output, if any, from the command is displayed in a **TextArea** in the dialog box. The user again has the option of acknowledging the alarm or canceling the **TimerAgent**, if it was an interval alarm. The third case, EVENT, causes the **TimerAgent** to send a **CIAgentEvent** to any registered listeners.

```
// method of Runnable (Thread) interface
 public void run() {
    if (interval > 0) {
      while(stopped == false){     // interval timer
        try {
           Thread.sleep((long)interval) ;  // in milliseconds
           trace(name + ": Interval sleep over ") ;
        }
        catch (InterruptedException e)
        {
          // interrupted
        }
            performAction() ;
      }
    } else {                       // one-shot alarm
       try {
           Date date = new Date() ; // get current date/time
           int curDay = date.getDay() ;
           int curHour = date.getHours() ;
           int alarmHour = time.getHours() ;
           int deltaHour = ((alarmHour - curHour) % 24) * 60; // min
           int curMin = date.getMinutes() ;
           int alarmMin = time.getMinutes() ;
           long deltaMin = (alarmMin - curMin) % 60 ;
           long delta = (deltaHour + deltaMin) * 60 * 1000 ;
           trace("going to sleep for " + delta + " msecs") ;
           Thread.sleep(delta) ;  // in milliseconds
           date = new Date() ; // get current date/time
```

```
            trace(name + ":Alarm sleep over at " + date) ;
            notifyCIAgentEventListeners(new
                    CIAgentEvent(this, "alarm")); // signal listeners
          }
          catch (InterruptedException e)
          {
            // interrupted
          }
              performAction() ;
      }

  }
   public  void ciaEventFired(CIAgentEvent e) {
       trace(name + ":  CIAgentEvent received by " + name + " from "
             e.getSource() + " with arg " + e.getArgObject()) ;
   }

   // perform the action specified by the user
   void performAction() {
    switch (action) {
           case 0:
                trace(name + ": Alert fired \n") ;
                ((AlertDialog)actionDialog).label1.setText(parms) ;
                actionDialog.show() ;
                if (((AlertDialog)actionDialog).cancel == true)stop()
                break ;
           case 1:
                trace(name + ": Executing command \n") ;
                executeCmd(parms) ;
                break ;
           case 2:
                notifyCIAgentEventListeners(
                     new CIAgentEvent(this,"interval")); // signal
                break ;
          }
   }

public  int executeCmd(String cmd) {
     Process process ;
     String line ;
     textArea = ((ExecuteDialog)actionDialog).textArea1 ;
     actionDialog.show() ;
     try {
         // this prefix works for Win 95/NT only ???
         process = Runtime.getRuntime().exec("command.com /c " +
                                            cmd + "\n");

         DataInputStream data =
               new DataInputStream(process.getInputStream());
         while ((line = data.readLine()) != null)
```

```
        {
            trace(line);
        }
        data.close();

    } catch (IOException err) {
        trace("Error: EXEC failed, " + err.toString()) ;
        err.printStackTrace();
       return -1 ;
    }
     if (((ExecuteDialog)actionDialog).cancel == true) stop() ;
     return process.exitValue() ;
}
```

Note that the **TimerAgent** provides a flexible set of capabilities. An alarm can be set to start system backups at midnight or an interval alarm can be set to notify the user every hour on the hour. Alarms can also be used to kick off other CIAgents to do their thing. Again, there is not much intelligence here, but it is useful software nonetheless.

Watches: The FileAgent

The next **CIAgent** is named **FileAgent** because its purpose in life is to watch a file system and to let the application know when some specified event occurs. The agent can alert the user with the **AlertDialog** whenever a file is modified, or if the size of a file gets too large (monitoring the size of a swap file, for example). It can also alert the application or another agent whenever the target file is deleted. Like the **TimerAgent**, the **FileAgent** can perform one of three actions when it detects the watch condition. It can display an alert, execute a command or start an application, or notify another **CIAgent** of the condition.

Much of the logic of the **FileAgent** is similar to the **TimerAgent**, so we will only discuss the unique data members and the *process()* method. Three Watch conditions are supported: if the file is MODIFIED, if the file is DELETED (it does not exist in the file system), and if the file exceeds a threshold size in bytes. The name of the file or directory is stored in the *fileName* **String** member, and information about when it was changed is stored in *lastChanged*.

The *run()* method goes to sleep every 15 seconds and tests the specified file condition each time it wakes up. If the watch condition is met, then the specified *action* is performed as it was in the **TimerAgent**.

```
public class FileAgent extends CIAgent {

   // conditions to check for
   public final int MODIFIED = 0 ;
   public final int DELETED = 1 ;
   public final int THRESHOLD = 2 ;

   // actions to take when conditions are true
   public final int ALERT = 0;    // display a dialog
   public final int EXECUTE = 1;  // start a process
   public final int EVENT = 2;    // send an event to another agent

   int action = EVENT ;   // default action is to send an event

   public FileAgent() { name = "Watch" ; };
   public FileAgent (String Name) { super(Name) ;}

   String fileName ;
   File   file ;
   long   lastChanged ;
   int    condition ;
   int    threshold ;
   Dialog actionDialog ;
   String parms ;

 // return a string for display in a list box
 public String getDisplayString() {
    String out = new String() ;
    out = name + " fileName=" + fileName + "cond=" + condition ;
    return out ;
 }

   public void setFileName(String fName) { fileName = fName;
                                           file = new File(fName) ;
                                           lastChanged = file.lastMo

   public String getFileName() { return fileName; }
   public boolean exists() { return file.exists() ; }
   public boolean changed() { long changeTime = lastChanged ;
                              lastChanged = file.lastModified() ;
                              return !(lastChanged == changeTime) ;
   }
```

```
  public long length() { return file.length(); }  // file size
  public boolean isDirectory() { return file.isDirectory(); }
  public long lastModified() { return file.lastModified(); }

  public void setCondition(int cond) { condition = cond; }
  public int getCondition() { return condition ; }

  public void setThreshold(int thresh) { threshold = thresh; }
  public int getThreshold() { return threshold; }

  public void setAction(int act) { action = act ; }
  public int getAction() { return action ; }

  public void setParms(String params) { parms = params; }
  public String getParms() { return parms; }

  public void setDialog(Dialog dlg) { actionDialog= dlg ; }

public void stop() {
   stopped = true;
   trace(name + " stopped \n") ;
   runnit.stop() ;
 }

public void process() {
   stopped = false ;
   // start a thread running
   System.out.println("Starting Watch \n") ;
   runnit.start() ;
}

  // start an interval timer and watch the specified file/dir
  // for the condition every 15 seconds.  If the condition occurs
  // do the specified action.
  public void run() {

    int interval = 15 * 1000 ;   // 15 seconds
    while(stopped == false){     // interval timer
       try {
          Thread.sleep((long)interval) ;  // in milliseconds
          trace(name + ": checking " + fileName + " \n") ;
       }
       catch (InterruptedException e)
       {
         // interrupted
       }
       boolean cond = checkCondition() ;
       if (cond == true) {
          performAction() ;
       }
```

```
      }
      return ;
    }

    boolean checkCondition() {
        boolean truth = false ;
        switch (condition) {
         case MODIFIED:
            truth = changed() ;
            break ;
         case DELETED:           // was file deleted?
            truth = !exists() ;  // see if file exists
            break ;
         case THRESHOLD:
            truth = threhold > length() ;
            break ;
        }
        return truth ;
    }
 void performAction() {
    switch (action) {
        case 0:
             trace(name + ": Alert fired \n") ;
             ((AlertDialog)actionDialog).label1.setText(parms) ;
             actionDialog.show() ;
             if (((AlertDialog)actionDialog).cancel == true) stop()
             break ;
        case 1:
             trace(name + ": Executing command \n") ;
             executeCmd(parms) ;
             break ;
        case 2:
             notifyCIAgentEventListeners(
                    new CIAgentEvent(this,"interval")); // signal
             break ;
          }
}

 public  int executeCmd(String cmd) {
     Process process ;
     String line ;
     textArea = ((ExecuteDialog)actionDialog).textArea1 ;
     actionDialog.show() ;
     try {
         // this prefix works for Win 95/NT only ???
         process = Runtime.getRuntime().exec("command.com /c " +
                   cmd + "\n");
         DataInputStream data =
            new DataInputStream(process.getInputStream());
         while ((line = data.readLine()) != null)
```

```
            {
                trace(line);
            }
            data.close();

        } catch (IOException err) {
            trace("Error: EXEC failed, " + err.toString()) ;
            err.printStackTrace();
           return -1 ;
        }
        if (((ExecuteDialog)actionDialog).cancel == true) stop() ;
        return process.exitValue() ;
    }

    public  void ciaEventFired(EventObject e) {
         System.out.println("FileAgent:  CIAgentEvent received by " +
                              name + " from " + e) ;
     }

}
```

PCManager Application

The **PCManager** class contains the *main()* application which constitutes our PCManager application. We used Symantec Visual Café to generate the GUI code and used our **TimerAgent** and **FileAgent** under the covers to provide the Alarm and Watch functions respectively. Most of the code is generated and is not particularly interesting. We point out several points of interest. The **PCManager** contains two **Hashtables**, one for Alarms and one for Watches. When the user selects the **Create an Alarm** or **Create a Watch** from the MenuBar, a secondary dialog is displayed and the user specifies the parameters associated with the Alarm or Watch. The specification of an Alarm or Watch results in the creation of a new agent to provide that function. Thus we could have tens of agents running at the same time, each using its own thread and coming alive to check the specified condition or to notify the user or other agents of some occurrence.

The *main()* method simply displays the application **Frame**. All user events come through the MenuBar actions and are processed by the *action()* method and routed to the specific event handling methods.

```
public class PCManager extends Frame {
     void Alarm_Action(Event event) {

          //{{CONNECTION
          // Create with title, show as modal...
          AlarmDialog dlg = new AlarmDialog(this, “Create Alarm”, tr
          //}}
          dlg.show() ;
          TimerAgent agent = dlg.getAgent() ;
          if (agent != null) {
         alarms.put(agent.name, agent) ;
         list1.addItem(agent.getDisplayString()) ;
         agent.textArea = textArea1 ;
         agent.process() ;
          }
     }

     void Watch_Action(Event event) {

          //{{CONNECTION
          // Create with title, show as modal...
          WatchDialog dlg = new WatchDialog(this, “Create Watch”, tr
          //}}
          dlg.show() ;
          FileAgent agent = dlg.getAgent() ;
          if (agent != null) {
         watches.put(agent.name, agent) ;
         list1.addItem(agent.getDisplayString()) ;
         agent.textArea = textArea1 ;
         agent.process() ;
          }
     }

     void Open_Action(Event event) {
          //{{CONNECTION
          // Action from Open... Show the OpenFileDialog
          openFileDialog1.show();
          //}}
     }

    void Cut_Action(Event event) {
        if (event.target == list1) {
```

```
      int index = list1.getSelectedIndex() ;
      String item = list1.getSelectedItem() ;
      list1.delItem(index) ;

      StringTokenizer tok = new
      StringTokenizer(item,":") ;
      String itemName = tok.nextToken() ;
      // need to actually remove it from the
      // hashtable or vector here !!!!!
      watches.remove(itemName) ;
      alarms.remove(itemName) ;

    }
}

void About_Action(Event event) {
     //{{CONNECTION
     // Action from About Create and show as modal
     (new AboutPCManagerDialog(this, true)).show();
     //}}
}

void Exit_Action(Event event) {
     //{{CONNECTION
     // Action from Exit Create and show as modal
     (new QuitDialog(this, true)).show();
     //}}
}

public PCManager() {

//{{INIT_CONTROLS
setLayout(null);
addNotify();
resize(insets().left + insets().right + 405,insets().top + inset
openFileDialog1 = new java.awt.FileDialog(this, "Open",FileDialo
//$$ openFileDialog1.move(36,276);
list1 = new java.awt.List(0,false);
add(list1);
list1.reshape(insets().left + 12,insets().top + 24,372,70);
textArea1 = new java.awt.TextArea();
textArea1.reshape(insets().left + 12,insets().top + 144,374,135)
add(textArea1);
label1 = new java.awt.Label("Activity Log");
label1.reshape(insets().left + 12,insets().top + 120,196,20);
add(label1);
label2 = new java.awt.Label("Watches/Alarms");
label2.reshape(insets().left + 12,insets().top + 0,180,20);
add(label2);
setTitle("CIAgent PCManager Application");
```

```
 //}}

 //{{INIT_MENUS
 mainMenuBar = new java.awt.MenuBar();

 menu1 = new java.awt.Menu(“File”);
 menu1.add(“New”);
 menu1.add(“Open...”);
 menu1.add(“Save”);
 menu1.add(“Save As...”);
 menu1.addSeparator();
 menu1.add(“Exit”);
 mainMenuBar.add(menu1);

 menu2 = new java.awt.Menu(“Edit”);
 menu2.add(“Cut”);
 mainMenuBar.add(menu2);

 menu4 = new java.awt.Menu(“Create”);
 menu4.add(“Watch...”);
 menu4.add(“Alarm...”);
 mainMenuBar.add(menu4);

 menu3 = new java.awt.Menu(“Help”);
 mainMenuBar.setHelpMenu(menu3);
 menu3.add(“About”);
 mainMenuBar.add(menu3);
 setMenuBar(mainMenuBar);
 //$$ mainMenuBar.move(4,277);
 //}}
 }

 public PCManager(String title) {
     this();
     setTitle(title);
 }

public synchronized void show() {
 move(50, 50);
 super.show();
}

 public boolean handleEvent(Event event) {
if (event.id == Event.WINDOW_DESTROY) {
      hide();         // hide the Frame
      dispose();      // free the system resources
      System.exit(0); // close the application
      return true;
}
     return super.handleEvent(event);
```

```
}

public boolean action(Event event, Object arg) {
     if (event.target instanceof MenuItem) {
          String label = (String) arg;
          if (label.equalsIgnoreCase("Alarm...")) {
               Alarm_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase("Watch...")) {
               Watch_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase("Open...")) {
               Open_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase("About")) {
               About_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase("Exit")) {
               Exit_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase("Cut")) {
               Cut_Action(event);
               return true;
          }
  }
  return super.action(event, arg);
}

static public void main(String args[]) {
    (new PCManager()).show();
}

//{{DECLARE_CONTROLS
java.awt.FileDialog openFileDialog1;
java.awt.List list1;
java.awt.TextArea textArea1;
java.awt.Label label1;
java.awt.Label label2;
//}}

//{{DECLARE_MENUS
java.awt.MenuBar mainMenuBar;
java.awt.Menu menu1;
java.awt.Menu menu2;
java.awt.Menu menu4;
```

```
    java.awt.Menu menu3;
    //}}

    Hashtable watches = new Hashtable() ;
    Hashtable alarms = new Hashtable() ;
}
```

Previous | Table of Contents | Next

The AlarmDialog Class

The **AlarmDialog** class is created and called by the *PCManager.Alarm_Action()* method in response to the user's selection of the **Create an Alarm** menu option. Most of this code was generated by Symantec Visual Café. The only data member is the **TimerAgent** which is instantiated when the **OK** button is clicked (in the *button1_Clicked()* method). The user data is extracted from the **TextField** and **Choice** controls and **TimerAgent** methods are called to set the values on the agent itself. When this information is passed on to the **TimerAgent** instance the **AlarmDialog** is closed using the *hide()* method. The **PCManager** then retrieves the newly created and configured **TimerAgent** from the dialog by calling the *AlarmDialog.getAgent()* method.

```
public class AlarmDialog extends Dialog {

    // Cancel
     void button2_Clicked(Event event) {

          //{{CONNECTION
          // Hide the Dialog
          hide();
          //}}
     }

    // OK
     void button1_Clicked(Event event) {

        String name = textField4.getText() ;
        agent = new TimerAgent(name) ;
        if (radioButton1.getState() == true) {
            // one time
            String time = textField2.getText() ;
            StringTokenizer tok = new StringTokenizer(time,":") ;
            int hour = new Integer(tok.nextToken()).intValue() ;
            int min = new Integer(tok.nextToken()).intValue() ;
            Date date = new Date() ;
            date.setHours(hour) ;
            date.setMinutes(min) ;
            agent.setTime(date) ;
        } else {
            // interval
            String interval = textField3.getText() ;
            agent.setInterval(new Integer(interval).intValue()) ;
```

```
        }
        int action = choice2.getSelectedIndex() ;
        agent.setAction(action) ;
        if (action == 0) agent.setDialog(new AlertDialog((Frame)this
ent(),name + ": Alert", false)) ;
        if (action == 1) agent.setDialog(new ExecuteDialog((Frame)th
ent(),name + "Execute", false)) ;

        String parms = textField1.getText() ;
        agent.setParms(parms) ;

          //{{CONNECTION
          // Hide the Dialog
          hide();
          //}}
     }

     public AlarmDialog(Frame parent, boolean modal) {

         super(parent, modal);

          //{{INIT_CONTROLS
          setLayout(null);
          addNotify();
          resize(insets().left + insets().right + 433,insets().top +
insets().bottom + 351);
          setBackground(new Color(12632256));
          label3 = new java.awt.Label("Action");
          label3.reshape(insets().left + 48,insets().top + 228,84,36
          add(label3);
          label4 = new java.awt.Label("Parameters");
          label4.reshape(insets().left + 36,insets().top + 264,108,2
          add(label4);
          textField1 = new java.awt.TextField();
          textField1.reshape(insets().left + 156,insets().top + 264,
          add(textField1);
          label5 = new java.awt.Label("Interval");
          label5.reshape(insets().left + 84,insets().top + 168,96,24
          add(label5);
          choice2 = new java.awt.Choice();
          choice2.addItem("Alert");
          choice2.addItem("Execute");
          choice2.addItem("Fire CIAgent Event");
          add(choice2);
          choice2.reshape(insets().left + 156,insets().top + 228,134
          label2 = new java.awt.Label("Time");
          label2.reshape(insets().left + 84,insets().top + 108,89,24
          label2.setBackground(new Color(16777215));
          add(label2);
          Group1 = new CheckboxGroup();
```

```
        radioButton1 = new java.awt.Checkbox(“One Time”, Group1, f
        radioButton1.reshape(insets().left + 24,insets().top + 48,
        add(radioButton1);
        radioButton2 = new java.awt.Checkbox(“Repeating”, Group1,
        radioButton2.reshape(insets().left + 24,insets().top + 144
        add(radioButton2);
        textField2 = new java.awt.TextField();
        textField2.reshape(insets().left + 216,insets().top + 108,
        add(textField2);
        textField3 = new java.awt.TextField();
        textField3.reshape(insets().left + 216,insets().top + 168,
        add(textField3);
        label6 = new java.awt.Label(“seconds”);
        label6.reshape(insets().left + 348,insets().top + 168,84,2
        label6.setBackground(new Color(16777215));
        add(label6);
        button2 = new java.awt.Button(“Cancel”);
        button2.reshape(insets().left + 252,insets().top + 312,84,
        add(button2);
        label7 = new java.awt.Label(“Name”);
        label7.reshape(insets().left + 24,insets().top + 12,152,28
        add(label7);
        textField4 = new java.awt.TextField();
        textField4.reshape(insets().left + 180,insets().top + 12,1
        add(textField4);
        button1 = new java.awt.Button(“OK”);
        button1.reshape(insets().left + 108,insets().top + 312,84,
        add(button1);
        label8 = new java.awt.Label(“hh:mm”);
        label8.reshape(insets().left + 348,insets().top + 108,78,2
        add(label8);
        setTitle(“”);
        setResizable(false);
        //}}

        textField4.setText(“Alarm”) ;
 }

 public AlarmDialog(Frame parent, String title, boolean modal) {
     this(parent, modal);
     setTitle(title);
 }

public synchronized void show() {
 Rectangle bounds = getParent().bounds();
 Rectangle abounds = bounds();
 move(bounds.x + (bounds.width - abounds.width)/ 2,
      bounds.y + (bounds.height - abounds.height)/2);
 super.show();
```

```
    }

     public boolean handleEvent(Event event) {
         if(event.id == Event.WINDOW_DESTROY) {
             hide();
             return true;
         }
          if (event.target == button1 && event.id == Event.ACTION_EV
                button1_Clicked(event);
                return true;
          }
          if (event.target == button2 && event.id == Event.ACTION_EV
                button2_Clicked(event);
                return true;
          }
          return super.handleEvent(event);
     }

     //{{DECLARE_CONTROLS
     java.awt.Label label1;
     java.awt.Choice choice1;
     java.awt.Label label3;
     java.awt.Label label4;
     java.awt.TextField textField1;
     java.awt.Label label5;
     java.awt.Choice choice2;
     java.awt.Label label2;
     java.awt.Checkbox radioButton1;
     CheckboxGroup Group1;
     java.awt.Checkbox radioButton2;
     java.awt.TextField textField2;
     java.awt.TextField textField3;
     java.awt.Label label6;
     java.awt.Button button2;
     java.awt.Label label7;
     java.awt.TextField textField4;
     java.awt.Button button1;
     java.awt.Label label8;
     //}}

     protected TimerAgent agent ;

     TimerAgent getAgent() { return agent; }
}
```

The WatchDialog Class

The **WatchDialog** class is created and called by the *PCManager.Watch_Action()* method in response to the user's selection of the **Create a Watch** menu option. Like the **AlarmDialog** class, most of this code was also generated by Symantec Visual Café. The only data member is the **FileAgent** which is instantiated when the **OK** button is clicked (in the *button2_Clicked()* method). The user data is extracted from the **TextField** and **Choice** controls and **FileAgent** methods are called to set the values on the agent itself. When this information is passed on to the **FileAgent** instance, the **WatchDialog** is closed using the *hide()* method. The **PCManager** then retrieves the newly created and configured **FileAgent** from the dialog by calling the *getAgent()* method.

```
public class WatchDialog extends Dialog {
    // Browse -- open File dialog
     void button1_Clicked(Event event) {

        //{{CONNECTION
          // Create with title, show as modal...
          FileDialog dlg = new FileDialog((Frame)this.getParent(), “
or Directory”, FileDialog.LOAD);
          //}}
        dlg.show() ;
        String dir = dlg.getDirectory() ;
        String fName = dlg.getFile() ;
        textField1.setText(dir+fName) ;
     }

    // cancel
     void button3_Clicked(Event event) {

          //{{CONNECTION
          // Hide the Dialog
          hide();
          //}}
     }

    // OK
     void button2_Clicked(Event event) {

        String name = textField4.getText() ;
        agent = new FileAgent(name) ;
```

```
        String fileOrDirName = textField1.getText() ;
        agent.setFileName(fileOrDirName) ;

        int cond = choice2.getSelectedIndex() ;
        agent.setCondition(cond) ;
        String threshold = textField2.getText() ;
        if (threshold.length() == 0) threshold = “0” ;
        agent.setThreshold(new Integer(threshold).intValue());

        int action = choice1.getSelectedIndex() ;
        agent.setAction(action) ;
        if (action == 0) agent.setDialog(new AlertDialog((Frame)this
ent(),name + “: Alert”, false)) ;
        if (action == 1) agent.setDialog(new ExecuteDialog((Frame)th
ent(),name + “: Execute”, false)) ;
        String parms = textField3.getText() ;
        agent.setParms(parms) ;
          //{{CONNECTION
          // Hide the Dialog
          hide();
          //}}
    }

    public WatchDialog(Frame parent, boolean modal) {

        super(parent, modal);

          //{{INIT_CONTROLS
          setLayout(null);
          addNotify();
          resize(insets().left + insets().right + 430,insets().top +
insets().bottom + 309);
          setBackground(new Color(12632256));
          label1 = new java.awt.Label(“File or Directory”);
          label1.reshape(insets().left + 24,insets().top + 48,135,24
          add(label1);
          label2 = new java.awt.Label(“Action”);
          label2.reshape(insets().left + 24,insets().top + 180,144,3
          add(label2);
          choice1 = new java.awt.Choice();
          choice1.addItem(“Alert”);
          choice1.addItem(“Execute”);
          choice1.addItem(“Fire CIAgent Event”);
          add(choice1);
          choice1.reshape(insets().left + 36,insets().top + 216,146,
          textField1 = new java.awt.TextField();
          textField1.reshape(insets().left + 24,insets().top + 72,20
          add(textField1);
          label3 = new java.awt.Label(“Condition”);
          label3.reshape(insets().left + 264,insets().top + 48,139,2
```

```
        add(label3);
        choice2 = new java.awt.Choice();
        choice2.addItem("Modified");
        choice2.addItem("Deleted");
        choice2.addItem("Over threshold");
        add(choice2);
        choice2.reshape(insets().left + 264,insets().top + 72,138,
        label4 = new java.awt.Label("Threshold");
        label4.reshape(insets().left + 264,insets().top + 108,116,
        add(label4);
        textField2 = new java.awt.TextField();
        textField2.reshape(insets().left + 264,insets().top + 132,
        add(textField2);
        button1 = new java.awt.Button("Browse...");
        button1.reshape(insets().left + 24,insets().top + 120,117,
        add(button1);
        label5 = new java.awt.Label("Parameters");
        label5.reshape(insets().left + 252,insets().top + 192,124,
        add(label5);
        textField3 = new java.awt.TextField();
        textField3.reshape(insets().left + 264,insets().top + 216,
        add(textField3);
        button2 = new java.awt.Button("OK");
        button2.reshape(insets().left + 108,insets().top + 264,72,
        add(button2);
        button3 = new java.awt.Button("Cancel");
        button3.reshape(insets().left + 252,insets().top + 264,72,
        add(button3);
        label6 = new java.awt.Label("Name");
        label6.reshape(insets().left + 24,insets().top + 12,142,25
        add(label6);
        textField4 = new java.awt.TextField();
        textField4.reshape(insets().left + 180,insets().top + 12,1
        add(textField4);
        setTitle("");
        setResizable(false);
        //}}
        textField4.setText("Watch") ;  // default name
 }

 public WatchDialog(Frame parent, String title, boolean modal) {
     this(parent, modal);
     setTitle(title);
 }

public synchronized void show() {
 Rectangle bounds = getParent().bounds();
 Rectangle abounds = bounds();
```

```
     move(bounds.x + (bounds.width - abounds.width)/ 2,
          bounds.y + (bounds.height - abounds.height)/2);

     super.show();
    }

     public boolean handleEvent(Event event) {
         if(event.id == Event.WINDOW_DESTROY) {
             hide();
             return true;
         }
         if (event.target == button1 && event.id == Event.ACTION_EVE
                button1_Clicked(event);
                return true;
           }
           if (event.target == button2 && event.id == Event.ACTION_E
                button2_Clicked(event);
                return true;
           }
           if (event.target == button3 && event.id == Event.ACTION_E
                button3_Clicked(event);
                return true;
           }
           return super.handleEvent(event);
     }

     //{{DECLARE_CONTROLS
     java.awt.Label label1;
     java.awt.Label label2;
     java.awt.Choice choice1;
     java.awt.TextField textField1;
     java.awt.Label label3;
     java.awt.Choice choice2;
     java.awt.Label label4;
     java.awt.TextField textField2;
     java.awt.Button button1;
     java.awt.Label label5;
     java.awt.TextField textField3;
     java.awt.Button button2;
     java.awt.Button button3;
     java.awt.Label label6;
     java.awt.TextField textField4;
     //}}

     protected FileAgent agent ;

     FileAgent getAgent() { return agent ; }
}
```

Discussion

There are other approaches we could have used to provide the function provided by the PCManager application. In this chapter, we consciously decided to take the most straightforward approach in the interest of clarity of presentation. In this section we discuss some of the possible alternatives.

A popular method for creating these types of simple agent applications is to use if-then rules as the engine. From this perspective, the **TimerAgent** would be used only as an event-generating agent, and the **FileAgent** would be used as sensors in the antecedent of a rule. Whenever the **TimerAgent** sent a **CIAgentEvent** to the **PCManager** application, it would invoke a forward-chaining inference cycle, with rules specifying the various Alarm or Watch conditions. The three actions would have to be defined as effector methods in the **RuleBase**. When a rule fires, the effector method would be called and the **AlertDialog**, the **ExecuteDialog**, or a **CIAgentEvent** would be performed.

This design would have certainly impacted our user interface. Instead of filling in fields in the **AlarmDialog** or **WatchDialog** panels, the user would probably have filled out an if-then rule template. The **PCManager** would then be a big **RuleBase** that receives **CIAgentEvents** and performs inferencing (and actions) in response to those events. We make use of similar function in the Marketplace application discussed in Chapter 10.

Another alternative would be to move the common **AlertDialog** and **ExecuteDialog** processing out of the **TimerAgent** and **FileAgent** into some other separate Java classes. Perhaps we could have developed an **AlertAgent** whose role in life was to wait for a **CIAgentEvent** and then display a message to a user. This **AlertAgent** could even signal the **Cancel** message back to the requester agent. A **CommandAgent** or **ExecuteAgent** likewise could be started and then wait for a **CIAgentEvent** asking for a command string to be run. When it completed, it could send the output back to the requester agent.

When designing intelligent agent applications, or object-oriented applications in general, it is not always clear cut how the function should be partitioned. In many cases, it comes down to a judgment call as to how often the particular function will be reused by other classes. If it has wide applicability, then

separating it out is probably the thing to do.

Summary

In this chapter we developed a simple PCManager application using two **CIAgent**-based intelligent agents. The main points include:

- The PCManager application allows a user to perform two major functions: set Alarms to go off at specified times or intervals and set Watches on files or directories.
- Each Alarm represents an instance of an autonomous **TimerAgent**. The **TimerAgent** spins its own Thread and then goes to sleep once for a one-shot alarm, or repeatedly for an interval alarm.
- Each Watch represents an instance of an autonomous **FileAgent**. The **FileAgent** spins its own Thread and then goes to sleep for 15-second intervals. When it wakes up, it checks the state of the target file or directory.
- This application represents an example of the utility of autonomous, but not necessarily intelligent, agents.

Exercises

1. What other functions could the **TimerAgent** perform for an application? Implement one.
2. How would you extend the **FileAgent** to handle more complex conditions or watches on multiple files?
3. Sketch out a design for the PCManager application using the **RuleBase** approach described in the Discussion section. Would the application be bigger or smaller? Would the agents be bigger or smaller?
4. The PCManager Alarms go away when the application ends. How could you make the TimerAgents and the Alarms persistent across invocations of the application?

Chapter 9
NewsFilter Application

The Internet is the focus of the intelligent agent application in this chapter. A basic Internet news reader application is designed, implemented, and augmented with an intelligent agent that assists a user by filtering information. The user can specify a profile of keywords and provide positive and negative feedback on each news article. Three alternate methods are provided for filtering the articles, including keyword match, clustering, and predictive modeling based on the user feedback.

Introduction

The **CIAgent NewsFilter** application developed in this chapter provides the basic functionality of a network news reader. The **NewsFilter** can connect to a specified news server, request the articles from a specific news group (such as *comp.lang.java.api*), and download them to the personal computer. The subject line from each news article is displayed in an **awt.List** control, and the user can browse individual articles by selecting them from the list. So far, we have described a standard news reader.

The goal of this application is to help the user deal with all of the electronic noise generated in the news groups. Wouldn't it be great if when you downloaded a news group, all you saw were the articles that genuinely interested you? All of the posts from that jerk on the West Coast would disappear. All of the spam postings offering great cellular phone service or the greatest software since sliced bread would never again waste your time. On the other hand, you would never miss a news article or post that discussed the topics that interest you. That is the motivation behind the **CIAgent NewsFilter**. In the remainder of this chapter we'll we show how to apply intelligent agents to score and filter out the unwanted news articles. We will also explore the basic mechanisms for reading Internet news groups using the Net News Transport Protocol (NNTP).

Figure 9.1 CIAgent NewsFilter application.

The main panel of the **NewsFilter** application, shown in Figure 9.1, consists of three main GUI controls. The top left contains a **List** of news groups. The top right contains a **List** of article subject lines from the selected news group. The bottom **TextArea** is used to display the selected news article and to display trace information as the articles are scored and the various types of filters are created.

Under the **File** pull-down menu pictured in Figure 9.2, the user can open and close connections with the selected News Host (a server running NNTP server software) and also add the name of a news group to the list of news groups. To actually load the news group, the user must double-click on the item in the **Newsgroups awt.List**. Individual articles can be **Save**d and **Load**ed from the **NewsFilter** application. This function is provided primarily as a debugging aid, so that selected articles can be scored and added to the user profile data file. Both **Save Article** and **Load Article** options bring up a **FileDialog** so the user can specify the directory path and file to write or read.

The **Profile** menu shown in Figure 9.3 contains a list of actions related to the maintenance of the user's **NewsFilter** profile. The **Keywords...** option displays a secondary dialog that allows the user to enter a list of words or terms which are of interest. The Keyword dialog allows the user to add or remove keywords from the list. Specifying the keywords is the first thing a user should do when starting to build a customized user profile. A default set of terms is provided in the NewsFilter application. The ten default keywords are: { "java," "agents," "fuzzy," "intelligent," "neural," "network," "genetic," "Symantec," "Cafe," "Beans" }.

Figure 9.2 NewsFilter File menu.

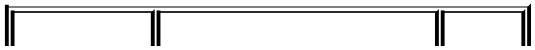
Figure 9.3 NewsFilter Profile menu.

Each article is searched and the number of times each keyword appears is counted and summed. This total number of keyword hits is used as the raw match score for keyword filtering.

The **Create profile** option will destroy the current profile and create a new one, using the keywords described in the Keyword dialog. The Filter profile consists of two text files. The first, *newsfilter.prf,* contains the **DataSet** definition (as described in Chapter 5), which is a list of data types, either continuous or discrete, and the corresponding field names. This file describes the layout of the data in the *newsfilter.dat* file. The user can add records to the *newsfilter.dat* file by selecting the **Add article** or **Add all articles** menu item. **Add article** will append a single record containing the match counts, the current feedback value, and the current filter score to *newsfilter.dat*. **Add all articles** will append the profile records for all articles in the currently loaded news group to the *newsfilter.dat* file. This file is used as the input data set for building the neural cluster filter and the neural feedback filter.

Score article and **Score all articles** will apply the currently selected Filter (either **Keyword**, **Cluster**, or **Feedback**) to the specified article or articles and update their match score. For the **Keyword** filter, the score is simply the sum of all keyword matches as described earlier. For the **Cluster** filter, this score is the average of the keyword scores for all articles that fall into that cluster (see Chapter 5 for the complete discussion of neural clustering). The score for the **Feedback** filter is the back propagation neural network output unit activation, a value ranging from 0.0 to 1.0. We will discuss this in more detail in the next section.

The **Filter** menu contains the options related to whether filtering is applied to articles in the news groups, which type of filter is used, and building the models required for the various filter methods. **Filter articles**, as shown in Figure 9.4, is a checkbox menu item that defaults to off. When checked, the articles will be filtered using the technique indicated in the next menu grouping. The default is **using Keywords**, but the user can select **using Clusters** or **using Feedback** as the filter method to be applied.



The next set of menu options is used to build the internal models required. **Build Keyword filter** simply forces a count of all of the keyword matches in the loaded articles. **Build Cluster filter** instantiates a Kohonen map neural network and reads the data in the *newsfilter.dat* file to train the network**. Build Feedback filter** creates and trains a back propagation neural network that uses the feedback data in the *newsfilter.dat* file as the target of a prediction model.

The **Feedback** menu displayed in Figure 9.5 allows the user to assign a value corresponding to the usefulness of the article. Each article has a feedback value that is automatically set when the news group is read. This value ranges from 0 for articles where there are no matches to a 1 for articles that have more than 5 matches. This automatic assignment is provided as a convenience to the user so that a value doesn't have to be set for each article.

Figure 9.4 NewsFilter Filter menu.

Figure 9.5 NewsFilter Feedback menu.

The user is allowed to override the automatic value through the **Feedback** menu. The feedback value is used when training the back propagation model in the Feedback filter. The label **Useless** maps to a feedback value of 0, **Not very useful** to 0.25, **Neutral** to 0.5, **Mildly Interesting** to 0.75, and **Interesting** to 1.0.

An Example

In this section, we describe the process used to create and use the three filters provided by the NewsFilter application. To start the application, run the following command:

java NewsFilter

When the **NewsFilter** main panel is displayed, first go to **File** and select the **Open News Host...** option. The NewsHost dialog is displayed showing a default news server. Users should enter the name of their Internet Service Provider's News Server here. When **OK** is clicked, a message will appear in the **TextArea** stating that the NewsFilter is trying to connect to the server. The response from the server will be echoed to the **TextArea**.

Next, go to the **Profile** menu, and select **Keywords**. A list of the 10 default keywords hardcoded into the NewsFilter application is displayed. Users can select any keyword they wish (single words only, no phrases). Next, select **Create profile** from the same menu. This initializes the two profile files, *newsfilter.dat* and *newsfilter.dfn*. You are ready to start building you personal filter profile.

The **Newsgroup** list on the top left is primed with five default Internet news groups. They are: *comp.ai.fuzzy*, *comp.ai.neural-nets*, *comp.lang.java.api*, *comp.lang.java.misc*, and *comp.lang.java.tech*. These news groups were chosen because they are reasonable places to find news posts or articles containing our default keyword list. To load the first news group, *comp.lang.ai*, double-click on that item in the list. The **NewsFilter** then begins requesting all of the articles the news host has in the news group. As each article is downloaded, the subject line is parsed from the header and is displayed in the news articles list on the top right, and the body or text of the message is read and displayed in the **TextArea**. The subject line and body are stored in the **NewsArticle** object.

Now that a set of news articles is loaded, we can see how well they match our specified keywords. The match score for the whole set can be computed using **Profile/Score all articles** or, for an individual article, using **Profile/Score article**. The keyword match is displayed in the bottom of the **TextArea**. Filter the articles by going to the **Filter** menu and checking the **Filter articles** menu item at the top of that menu. The articles are sorted by their match score and redisplayed in descending order in the **News Articles** list. The top scoring item is selected, and the body of that article is displayed in the **TextArea**. When the **Filter articles** menu item is deselected, the article list reverts to the original order.

In order to use either the Cluster filter or Feedback filter, profile data must first

be saved on a reasonably sized set of articles. By default, the NewsFilter reads only 20 articles from each news group (this is a constant that can be easily changed in the code). If the five default news groups are read, that gives 100 article profile records. To store the article profiles to the *newsfilter.dat* text file, select the **Add all articles** menu option in the **Profile** menu. Assuming that the news server had 20 articles in the *comp.lang.ai* news group, 20 records would be written.

Continue by downloading each of the next four news groups in turn, selecting **Score all articles**, and then **Add all articles** from the **Profile** menu. At this point there should be approximately 100 records in the *newsfilter.dat* file.

Now that there is profile data, we can use the Cluster filter. First, select the **Build Cluster Filter** option on the **Filter** menu. A neural network is created that reads the *newsfilter.dat* file and clusters the article profile records into 4 segments. (This was an arbitrary decision, it could easily have been 9, 16, or 25). The raw match score for each article in each cluster is summed and an average is computed. The score for each article is set to the average value of the cluster it fell into. This step produces the segmentation model used by the Cluster filter. However, in order to use the Cluster filter, it must also be selected as the filter type on the **Filter** menu. If **Filter articles** is checked, the news articles list will be refreshed with the articles from the top-scoring cluster first, the second-best cluster second, and so on.

The last filter provided in the **NewsFilter** application is the Feedback filter. This too uses a neural network model to determine the article scores. A back propagation regression model is created using the article profile records as input and the feedback score as the target output value. If none of the default feedback settings were overridden before the articles were added to the profile, the model will predict a score ranging from 0 to 1, where any articles having more than 5 keyword matches are a 1. If different values were provided for selected articles by explicit user feedback using the five levels defined in the **Feedback** menu, the results could be slightly different. For example, if articles that contained the word “neural” were rated as more interesting than ones that contained “java,” they would have higher scores from the Feedback filter, even though each had an identical raw match score.

After the Feedback filter model is built, it must be selected as the desired filter technique by checking the **using Feedback** option on the **Filter** menu. If **Filter**

articles is turned on, the articles in the top right list box will be reordered according to the score they got from the neural prediction model.

Previous | Table of Contents | Next

NewsFilter Class

The **NewsFilter** class is the main application class in this application. It extends **Frame** using two **List** controls and a large **TextArea** as the main interface elements, and a **MenuBar** of options to allow the user to perform the desired actions. Much of the GUI code is generated by the Symantec Visual Café product, and so we will only talk about it at a high level.

There are many action methods used in this application because of the large number of menu options provided. The *action()* method examines the label of the event object and the appropriate method is called when a high-level semantic event is generated by a user action. By looking at the **Strings** tested in the *action()* method, you can easily determine the corresponding event-handler method.

Actions that deal with communicating and reading data from the news server are handled by the **NewsFilter** class. Actions related to building the user profile, scoring articles, or building the neural network filters are handled by the **FilterAgent**. Most of these methods calls are handled synchronously. That is, the **NewsFilter** application calls the **FilterAgent** on its main processing thread and control does not return until the **FilterAgent** method completes. The exceptions to this model are the *BuildClusterFilter_Action()* and *BuildFeedbackFilter_Action()* methods. They call methods in the **FilterAgent** that simply turn on boolean switches or flags for the agent to see when it wakes up periodically looking for work. This approach is used because training the neural networks can be a long-running action and by handling these tasks asynchronously, the **NewsFilter** application is not tied up waiting for these to complete. The **FilterAgent** signals when it is done by calling the *clusterNetBuilt()* and *scoreNetBuilt()* methods. These enable the corresponding **Filter** menu options, so that Cluster and Feedback filtering can be selected.

```
public class NewsFilter extends Frame {
     void AddNewsGroup_Action(Event event) {

          //{{CONNECTION
          // Create with title, show as modal...
          NewsGroupDialog dlg = new NewsGroupDialog(this,
                  “Enter name of New Group to add to list”, true);
```

```
        dlg.show();
        //}}
        String newsGroup = dlg.textField1.getText() ;
        if (newsGroup != null) {
            list1.addItem(newsGroup) ;
        }
    }

    void OpenNewsHost_Action(Event event) {

        //{{CONNECTION
        // Create with title, show as modal...
        NewsHostDialog dlg = new NewsHostDialog(this,
                  "Enter address or name of News Server", true) ;
        //}}
        dlg.setNewsHost(newsHost) ;  // set to current value
        dlg.show() ;
        String temp = dlg.newsHost ;
        if (temp != null) {
        newsHost = temp ;
        }
        if (newsHost != null) {
      textArea1.appendText("\nConnecting to " + newsHost + " \n" );
           connectToNewsHost() ;
      }
    }

  void Keywords_Action(Event event) {

        //{{CONNECTION
        // Create with title, show as modal...
        KeywordDialog dlg =
           new KeywordDialog(this,"Specify Filter Keywords",true);
        //}}
        dlg.setKeywords(keywords) ;  // initialize the list
        dlg.show() ;
        keywords = dlg.getKeywords();
        if (keywords != null) {
          textArea1.appendText(keywords[0]) ;
        }
   }
  // user selected a news group -- download it
  void NewsGroup_Action(Event event) {
     list2.clear() ;
     textArea1.setText("") ;
     articles = new Vector() ;
     String newsGroup = list1.getSelectedItem() ;
     readNewsGroup(newsGroup) ;
  }
```

```
// user selected a news item -- put body in textArea
void NewsItem_Action(Event event) {
   int index = list2.getSelectedIndex();
   if (index != -1) {
     currentArt = (NewsArticle)articles.elementAt(index) ;
     textArea1.setText(currentArt.body); // clear the display
     filterAgent.score(currentArt, keywords, filterType);
   } else {
     currentArt = null ;
   }
}

void Feedback_Action(double feedback) {
    if (currentArt != null) {
        currentArt.feedback = feedback ;
    }
 }

 void FilterArticles_Action(Event event) {
     filterArticles() ;
  }

 void FilterType_Action(int fType) {
     switch (fType) {
      case 0:   // keyword
         keywordFilter.setState(true) ;
         clusterFilter.setState(false) ;
         feedbackFilter.setState(false) ;

      break ;
      case 1:   // cluster
         keywordFilter.setState(false) ;
         clusterFilter.setState(true) ;
         feedbackFilter.setState(false) ;
      break ;
      case 2:   // feedback
         keywordFilter.setState(false) ;
         clusterFilter.setState(false) ;
         feedbackFilter.setState(true) ;
      break ;
     }
     filterType = fType ;
     if (filterArt.getState() == true) filterArticles() ;
  }

void AddArticle_Action(Event event) {
     // open the profile file and append the
     // score data for this article
     filterAgent.addArticleToProfile(currentArt) ;
```

```
  }

void AddAllArticles_Action(Event event) {
    // open the profilefile and append the
    // score data for all articles
     filterAgent.addAllArticlesToProfile(articles) ;
  }

void ScoreArticle_Action(Event event) {
     filterAgent.score(currentArt, keywords, filterType) ;
  }

void ScoreAllArticles_Action(Event event) {
     // do it unconditionally
     filterAgent.score(articles, keywords, filterType) ;
     scored = true ;
  }

void CreateProfile_Action(Event event) {
      // empty the newsfilter.prf file
     File prf = new File("newsfilter.prf") ;
     prf.delete() ;
     File dat = new File("newsfilter.dat") ;
     dat.delete() ;
     // create a newsfilter.prf file using current keywords
     filterAgent.writeProfileDataDefinition(keywords) ;
  }

void BuildKeywordFilter_Action(Event event) {
     filterAgent.score(articles, keywords, filterType);
  }

void BuildClusterFilter_Action(Event event) {
     clusterFilter.disable();  // wait for completion
     filterAgent.buildClusterNet();  // do asynchronously
  }

void BuildFeedbackFilter_Action(Event event) {
     feedbackFilter.disable(); // wait for completion
     filterAgent.buildScoreNet(); // do asynchronously
  }

void clusterNetBuilt() {
   clusterFilter.enable() ; // FilterAgent has completed
}

void scoreNetBuilt() {
   feedbackFilter.enable() ; // FilterAgent has completed
}
```

```
void Open_Action(Event event) {
      //{{CONNECTION
      // Action from Open... Show the OpenFileDialog
      openFileDialog1.show();
      //}}
 }
void LoadArticle_Action(Event event) {
      //{{CONNECTION
      // Action from Open... Show the OpenFileDialog
      openFileDialog1.show();
      //}}
      String fileName = openFileDialog1.getFile();
      if (fileName != null) {
        NewsArticle art = new NewsArticle(fileName) ;
      art.readArticle(fileName) ;
      list2.addItem(fileName) ;
      textArea1.setText(art.body) ;
      articles.addElement(art) ;
    }
 }
void SaveArticle_Action(Event event) {
      //{{CONNECTION
      // Action from Save... Show the SaveFileDialog
      saveFileDialog1.show();
      //}}
      String fileName = saveFileDialog1.getFile();
      if (fileName != null) {
       int index = list2.getSelectedIndex();
       if (index != -1) {
     NewsArticle art = (NewsArticle)articles.elementAt(index) ;
        art.writeArticle(fileName) ;
       }
      }
 }

 void About_Action(Event event) {
      //{{CONNECTION
      // Action from About Create and show as modal
      (new AboutNewsFilterDialog(this, true)).show();
      //}}
 }

 void Exit_Action(Event event) {
   closeNewsHost() ;
      //{{CONNECTION
      // Action from Exit Create and show as modal
      (new QuitDialog(this, true)).show();
      //}}
 }
```

```
public NewsFilter() {

      //{{INIT_CONTROLS
      setLayout(null);
      addNotify();
      resize(insets().left + insets().right + 587,insets().top +
             insets().bottom + 404);
      setBackground(new Color(16777215));
      openFileDialog1 =
         new java.awt.FileDialog(this, “Open”,FileDialog.LOAD);
      saveFileDialog1 =
         new java.awt.FileDialog(this, “Save”,FileDialog.SAVE);

      //$$ openFileDialog1.move(36,336);
      list1 = new java.awt.List(0,false);
      add(list1);
      list1.reshape(insets().left + 12,insets().top + 36,276,88)
      list1.setBackground(new Color(16777215));
      list2 = new java.awt.List(0,false);
      add(list2);
      list2.reshape(insets().left + 300,insets().top + 36,276,88
      list2.setBackground(new Color(16777215));
      textArea1 = new java.awt.TextArea();
      textArea1.reshape(insets().left + 12,insets().top + 144,56
      add(textArea1);
      label1 = new java.awt.Label(“Newsgroup”);
      label1.reshape(insets().left + 12,insets().top + 12,93,24)
      add(label1);
      label2 = new java.awt.Label(“News Item”);
      label2.reshape(insets().left + 300,insets().top + 12,86,24
      add(label2);
      setTitle(“CIAgent News Filter Application”);
      setResizable(false);
      //}}

      //{{INIT_MENUS
      mainMenuBar = new java.awt.MenuBar();

      menu1 = new java.awt.Menu(“File”);
      menu1.add(“Reset”);
      menu1.add(“Open News Host...”);
      menu1.add(“Close News Host”);
      menu1.add(“Add News Group...”);
      menu1.addSeparator();
      menu1.add(“Save Article...”);
      menu1.add(“Load Article...”);
      menu1.addSeparator();
      menu1.add(“Exit”);
```

```
mainMenuBar.add(menu1);

menu6 = new java.awt.Menu(“Profile”);
menu6.add(“Keywords...”);
menu6.addSeparator();
menu6.add(“Add article”);
menu6.add(“Add all articles”);
menu6.addSeparator();
menu6.add(“Score article”);
menu6.add(“Score all articles”);
menu6.addSeparator();
menu6.add(“Create profile”);
mainMenuBar.add(menu6);

menu2 = new java.awt.Menu(“Edit”);
menu2.add(“Cut”);
menu2.add(“Paste”);
mainMenuBar.add(menu2);

menu4 = new java.awt.Menu(“Filter”);
filterArt =
      new java.awt.CheckboxMenuItem(“Filter articles”);
filterArt.setState(false);
menu4.add(filterArt);
menu4.addSeparator();
keywordFilter =
      new java.awt.CheckboxMenuItem(“using Keywords”);
keywordFilter.setState(true);
menu4.add(keywordFilter);
clusterFilter =
      new java.awt.CheckboxMenuItem(“using Clusters”);
clusterFilter.setState(false);
clusterFilter.disable(); // disable until model is built
menu4.add(clusterFilter);
feedbackFilter =
      new java.awt.CheckboxMenuItem(“using Feedback”);
feedbackFilter.setState(false);
feedbackFilter.disable(); // disable until model is built
menu4.add(feedbackFilter);
menu4.addSeparator();
menu4.add(“Build Keyword filter”);
menu4.add(“Build Cluster filter”);
menu4.add(“Build Feedback filter”);
mainMenuBar.add(menu4);

menu5 = new java.awt.Menu(“Feedback”);
menu5.add(“Useless”);              // 0
menu5.add(“Not very useful”) ;     // 0.25
menu5.add(“Neutral”) ;             // 0.5
```

```
      menu5.add(“Mildly interesting”) ;  // 0.75
      menu5.add(“Interesting”) ;         // 1
      mainMenuBar.add(menu5);

      menu3 = new java.awt.Menu(“Help”);
      mainMenuBar.setHelpMenu(menu3);
      menu3.add(“About”);
      mainMenuBar.add(menu3);
      setMenuBar(mainMenuBar);
      //$$ mainMenuBar.move(0,336);
      //}}

      // set default values here
      newsHost = “news-s01.ny.us.ibm.net” ;
      filterAgent.newsFilter = this ;
   filterAgent.textArea = textArea1 ;
   list1.addItem(“comp.ai.fuzzy”) ;
   list1.addItem(“comp.ai.neural-nets”) ;
   list1.addItem(“comp.lang.java.api”) ;
   list1.addItem(“comp.lang.java.misc”) ;
   list1.addItem(“comp.lang.java.tech”) ;
   keywords = new String[10] ; // default list
   keywords[0] = “java” ;
   keywords[1] = “agents” ;
   keywords[2] = “fuzzy” ;
   keywords[3] = “intelligent” ;
   keywords[4] = “neural” ;
   keywords[5] = “network” ;
   keywords[6] = “genetic” ;
   keywords[7] = “Symantec” ;
   keywords[8] = “Cafe” ;
   keywords[9] = “Beans” ;
   filterAgent.process() ; // start the Filter agent thread
 }

 public NewsFilter(String title) {
     this();
     setTitle(title);
 }

public synchronized void show() {
 move(50, 50);
 super.show();
 }
 public boolean handleEvent(Event event) {
 if (event.id == Event.WINDOW_DESTROY) {
      hide();         // hide the Frame
      dispose();      // free the system resources
      System.exit(0); // close the application
      return true;
```

```
}
     return super.handleEvent(event);
}

public boolean action(Event event, Object arg) {

     if (event.target instanceof MenuItem) {
         String label = ((MenuItem)event.target).getLabel();

          if (label.equalsIgnoreCase(“Add News Group...”)) {
               AddNewsGroup_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase(“Open News Host...”)) {
               OpenNewsHost_Action(event);
               return true;
          } else
               if (label.equalsIgnoreCase(“Close News Host”)) {
               closeNewsHost();
               return true;
          } else
               if (label.equalsIgnoreCase(“Open...”)) {
               Open_Action(event);
               return true;
          } else

              if (label.equalsIgnoreCase(“Save article...”)) {
               SaveArticle_Action(event);
               return true;
          } else

          if (label.equalsIgnoreCase(“Load article...”)) {
               LoadArticle_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase(“Add article”)) {
               AddArticle_Action(event);
               return true;
          } else
          if (label.equalsIgnoreCase(“Add all articles”)) {
               AddAllArticles_Action(event);
               return true;
          } else
        if (label.equalsIgnoreCase(“Score article”)) {
             ScoreArticle_Action(event);
             return true;
          } else
          if (label.equalsIgnoreCase(“Score all articles”)) {
               ScoreAllArticles_Action(event);
```

```
     return true;
} else
if (label.equalsIgnoreCase(“About”)) {
     About_Action(event);
     return true;
} else
if (label.equalsIgnoreCase(“Exit”)) {
          Exit_Action(event);
          return true;
} else
if (label.equalsIgnoreCase(“Keywords...”)) {
          Keywords_Action(event);
          return true;
} else
if (label.equalsIgnoreCase(“Create profile”)) {
          CreateProfile_Action(event);
          return true;
} else
 if (label.equalsIgnoreCase(“Useless”)) {
           Feedback_Action(0);
           return true;
} else
if (label.equalsIgnoreCase(“Not very useful”)) {
          Feedback_Action(0.25);
          return true;
} else
if (label.equalsIgnoreCase(“Neutral”)) {
          Feedback_Action(0.5);
          return true;
} else
if (label.equalsIgnoreCase(“Mildly interesting”)) {
        Feedback_Action(0.75);
          return true;
} else

if (label.equalsIgnoreCase(“Interesting”)) {
          Feedback_Action(1.0);
          return true;
} else

if (label.equalsIgnoreCase(“Filter articles”)) {
          FilterArticles_Action(event);
          return true;
} else
  if (label.equalsIgnoreCase(“using Keywords”)) {
          FilterType_Action(0);
          return true;
} else  if (label.equalsIgnoreCase(“using Clusters”))
         FilterType_Action(1);
```

```
                    return true;
            } else  if (label.equalsIgnoreCase(“using Feedback”))
                     FilterType_Action(2);
                    return true;
            } else
         if (label.equalsIgnoreCase(“Build Keyword filter”)) {
                     BuildKeywordFilter_Action(event);
                     return true;
            } else
             if (label.equalsIgnoreCase(“Build Cluster filter”)) {
                     BuildClusterFilter_Action(event);
                     return true;
            } else
         if (label.equalsIgnoreCase(“Build Feedback filter”)) {
                     BuildFeedbackFilter_Action(event);
                     return true;
            }

        }
        if (event.target instanceof List) {
            if (event.target == list1) {
                // load selected news group
                NewsGroup_Action(event) ;
                return true;
            }
            if (event.target == list2) {
                // load selected article
                NewsItem_Action(event);
                return true ;
            }
        }
        return super.action(event, arg);
}

static public void main(String args[]) {
    (new NewsFilter()).show();
}

//{{DECLARE_CONTROLS
java.awt.FileDialog openFileDialog1;
java.awt.FileDialog saveFileDialog1;
java.awt.List list1;
java.awt.List list2;
java.awt.TextArea textArea1;
java.awt.Label label1;
java.awt.Label label2;
//}}

//{{DECLARE_MENUS
java.awt.MenuBar mainMenuBar;
```

```
java.awt.Menu menu1;
java.awt.Menu menu6;
java.awt.Menu menu2;
java.awt.Menu menu4;
java.awt.CheckboxMenuItem filterArt;
java.awt.CheckboxMenuItem keywordFilter;
java.awt.CheckboxMenuItem clusterFilter;
java.awt.CheckboxMenuItem feedbackFilter;
java.awt.Menu menu5;
java.awt.Menu menu3;
//}}
```

Previous | Table of Contents | Next

The **NewsFilter** class's data members include the **newsHost**, which defaults to the IBM Global Network news server, the *news* **Socket**, which is used to connect and talk to the news server, and the *newsIn* and *newsOut* streams, which are used to send data over the **Socket** connection. Two **Vectors**, *articles* and *originalArticleList*, hold the sorted and original unsorted list of articles currently loaded into the **NewsFilter** application. The *currentArt* member is a reference to the **NewsArticle** instance selected in the top right **List** box. The *keywords* **String** array holds the words the user has defined for use in evaluating the relevance of articles.

The *filterAgent* member holds a reference to the **FilterAgent**, a **CIAgent**-based intelligent agent that handles the user profile maintenance and the scoring and building of models used by the Cluster and Feedback filters. The *scored* flag indicates whether the entire set of news group articles has been scored or not. The *filterType* member indicates which filter technique the user wants to use, if any, to determine the order that articles are displayed. The value 0 corresponds to Keyword filter, 1 means Cluster filter, and 2 means Feedback filter. These values are set in the *FilterType_Action()* method based on the settings of the three corresponding **CheckboxMenuItems** in the **Filter** menu.

```
String newsHost ;         // name of the news server
Socket news ;            // socket to news server
PrintStream newsOut ;    // stream to write to server
DataInputStream newsIn ; // stream to read from server
Vector articles = new Vector() ; // articles in news group
Vector originalArticleList ;     //unfiltered list of articles
String[] keywords = new String[0] ;  // user specified keywords
FilterAgent filterAgent  = new FilterAgent() ;
NewsArticle currentArt ;       // currently selected article
boolean scored = false ;       // true if articles were scored
int  filterType = 0 ;  // keyword filter as default
```

The *connectToNewsHost()* method makes the initial connection to the NNTP news server. The *newsHost* **String** contains the address of the news server supplied by the user (the default is news-s01.ny.us.ibm.net). The Network News Transfer Protocol specification by Kantor and Lapsley (Internet RFC #977) says that NNTP servers should listen on well-known port number 119, so an instance of a **Socket** to the *newsHost* is created using that port. The *newsIn* and *newsOut* data streams are instantiated. A **DataInputStream** is used for reading, and a

PrintStream for writing. After making the initial socket connection, the news server responds with its status, which is read into the *reply* **String** and displayed in the main **TextArea**. If anything goes wrong, such as an incorrect news host address or a network communication error, any **Exceptions** generated by the IO operations are caught and an error message is displayed.

```
public void connectToNewsHost() {

   try {
        news = new Socket(newsHost, 119) ;
        newsIn = new DataInputStream(news.getInputStream()) ;
        newsOut = new PrintStream(news.getOutputStream()) ;
        String reply = newsIn.readLine();
        textArea1.appendText( reply + "\n") ;
   }
   catch (Exception e) {
        textArea1.appendText("Exception:" + e) ;
   }
}
```

The *closeNewsHost()* method shuts down the **Socket** connection to the news server. The NNTP protocol requires the client to send the "QUIT" string as the closing message. That string is sent to the server and is echoed to the display. The news server does not send any reply message to a "QUIT."

```
public void closeNewsHost() {

   try {
         String cmd = "QUIT \n" ;
         newsOut.println(cmd) ;
         textArea1.appendText( cmd + " \n" ) ;
         newsIn.close() ;
   }
   catch (Exception e) {
         textArea1.appendText("Exception:" + e) ;
   }

}
```

The *readNewsGroup()* method takes a single parameter, the name of the news group to be read. NNTP requires the "GROUP" command, followed by the name of the news group using standard string representation (such as *comp.ai.neural-nets*). The news server responds with a return code, the number of articles in the news group that the server holds, the first and last article id, and an acknowledgment string stating that the news group has been selected. A

StringTokenizer is used to parse these parameters. Once a news group is selected on the server, an individual article must be selected. The first article id returned by the “GROUP” command is used with the “STAT” command to set an internal cursor on the news server, so that subsequent articles can be retrieved using the “NEXT” command, rather than requiring that each article id be known beforehand.

The majority of the method is a *do* loop, where the “NEXT” command is used to walk through the news group, one article at a time. First the “HEAD” command is used to download the header for each article. The *parseHeader()* method takes the article id as a single input parameter. It instantiates a **NewsArticle** instance, reads each line of the header and parses it, looking for the “Subject:” line. It stores the subject in the subject member of the **NewsArticle** object and continues reading the header lines until the news server signals it is done by sending a line containing only a period character. Back in the *readNewsGroupHeaders()* method, the subject is displayed in the news item list, and the new **NewsArticle** object is added to the *articles* **Vector**.

In a similar manner, the “BODY” command is sent to the news server and the lines of the news articles are downloaded and displayed in the **TextArea**. When the period sentinel character is received, the article body member is set by doing a *getText()* on the **TextArea** control.

The “NEXT” command steps to the next article in the news group. After each NNTP command string is sent, the return codes are verified. During testing we encountered cases where the server returned the header for an article in response to the “HEAD” command, but then balked when the “BODY” command was sent for the same article id.

The *do* loop exits when the *maxArticles* limit is reached, or if an error return code is received from the server. If any articles are read, the first one is selected in the news item list, and its body is displayed in the **TextArea**. Before any filtering is performed on the new group, the original set of articles in the original order is copied to the *originalArticles* member. If filtering is turned on the articles are immediately filtered using the specified filter type.

Please note, the **NewsFilter** class is not a commercial news reader. There is much more error-checking logic that should be added for a general-purpose news reader. However, this does provide the basic mechanism to browse an Internet news group and allows us to obtain news articles to build the user profile.

```
public void readNewsGroup(String newsGroup) {

   int maxArticles = 20 ;
   boolean exit = false ;

   scored = false ; // articles were not scored yet

   try {
         String cmd = “GROUP ” + newsGroup + “ \n” ;
         newsOut.println(cmd) ;
         String reply = newsIn.readLine();
         textArea1.appendText( cmd + “ \n” + reply + “\n”) ;

         StringTokenizer st = new StringTokenizer(reply) ;
         String s1 = st.nextToken() ;  // response code
         String s2 = st.nextToken() ;  // number of appends
         String s3 = st.nextToken() ;  // first id
         String s4 = st.nextToken() ;  // last id
         String s5 = st.nextToken() ;  // newsgroup

         cmd = “STAT ” + s3  + “\n” ;
         newsOut.println(cmd) ;
         reply = newsIn.readLine();
         textArea1.appendText( cmd + “ \n” + reply + “\n”) ;

         String retCode ;
         do {
             textArea1.setText(“”);
             cmd = “HEAD \n” ;
             newsOut.println(cmd) ;
             reply = newsIn.readLine();
             textArea1.appendText( cmd + “ \n” + reply + “\n”) ;
             StringTokenizer tok = new StringTokenizer(reply, “ ”) ;
             retCode = tok.nextToken() ;
             String id = tok.nextToken() ;
             String msgId = tok.nextToken() ;

             if (!retCode.equals(“221”)) continue ;

             // now read all header records for this article and par
```

```
            NewsArticle art = parseHeader(id) ;
            list2.addItem( art.subject ) ;  // display subject only
            articles.addElement(art) ;  // add to Vector

            cmd = "BODY \n" ;
            newsOut.println(cmd) ;
            reply = newsIn.readLine();
            StringTokenizer stok = new StringTokenizer(reply) ;
            retCode = stok.nextToken() ;  // response code
            if (!retCode.equals("222")) {  // error?
              articles.removeElement(art) ; // bad article
              continue ;
            }

            textArea1.appendText( cmd + " \n" + reply + "\n") ;
            do {
              reply = newsIn.readLine();
              textArea1.appendText( reply + "\n") ;

            } while(!reply.equals(".")) ;
            art.body = textArea1.getText() ;
            cmd = "\n NEXT \n" ;
            newsOut.println(cmd) ;
            reply = newsIn.readLine();
            textArea1.appendText( cmd + " \n" + reply + "\n") ;
            StringTokenizer st2 = new StringTokenizer(reply) ;
            retCode = st2.nextToken() ;  // response code

        }  while (retCode.equals("223") &&
                  (articles.size() < maxArticles)) ;
        list2.select(0) ;
        currentArt = (NewsArticle)articles.elementAt(0) ;
        textArea1.setText(currentArt.body) ;
      }
      catch (Exception e) {
            textArea1.appendText("Exception:" + e) ;
      }
       originalArticleList = (Vector)articles.clone() ; // save copy
       if (filterArt.getState() == true) filterArticles();

}
```

The *filterArticles()* method first checks to see if the **Filter articles CheckboxMenuItem** in the **Filter** menu is selected. If all of the articles in the news group haven't been scored yet, they are scored. Note that the articles, the current set of keywords, and the current type of filter are all passed to the **FilterAgent** for scoring. The result of this call is that the score member of each **NewsArticle** is set to the appropriate value based on the current *filterType*. The

insertionSort() method is called to sort the articles in decreasing order by score. The article **List** is then cleared and refilled with the articles in the new order.

```
// the user wants us to filter articles
// clear the news Item list and redisplay
// using the article scores to determine the order
public void filterArticles() {
    if (filterArt.getState() == true) {
      if (!scored) filterAgent.score(articles, keywords,
                                     filterType) ;
      Vector sortedByScore = insertionSort(articles) ;
      articles = sortedByScore;
    } else {
      articles = originalArticleList ;
    }

    list2.clear() ; // remove all articles from list
    Enumeration enum = articles.elements() ;
    while (enum.hasMoreElements()) {
        NewsArticle art = (NewsArticle)enum.nextElement();
        list2.addItem(art.subject) ;
    }
    list2.select(0) ;
    currentArt = (NewsArticle)articles.elementAt(0) ;
}
```

This *insertionSort()* method is a standard implementation of the insertion sort algorithm (Sedgewick 1984) adapted for Java. First, the elements in the **Vector** are copied into an array for convenience. Next, starting at the second element in the array, we walk through the array, sorting as we go. Once the sort algorithm is complete, the array elements are copied back into a return **Vector**.

```
// sort articles by decreasing order of score
Vector insertionSort(Vector articles) {

   int i, j ;
   int size = articles.size() ;

   NewsArticle sortedList[] = new NewsArticle[articles.size()] ;
   articles.copyInto( sortedList ) ;
   NewsArticle temp ;

   for (i=1 ; i < size ; i++) {
     temp = sortedList[i] ;
     textArea1.appendText(temp.score + " ") ;
     j = i ;
     while ((j > 0) && (sortedList[j-1].score < temp.score)) {
```

```
          sortedList[j] = sortedList[j-1];
          j = j - 1 ;
        }
        sortedList[j] = temp;
      }

        Vector outList = new Vector() ;
        textArea1.appendText("\n Sorted list = ") ;
        for (i=0 ; i < size ; i++) {
          temp = sortedList[i] ;
          textArea1.appendText(temp.score + " ") ;
          outList.addElement(temp) ;
        }
      return outList ;
}

}
```

Previous | Table of Contents | Next

NewsArticle Class

The **NewsArticle** class defines all the information about a single news article. The primary piece of information is the NNTP *id*, which is returned by the News Server in response to the "NEXT" command as all the articles in the news group are read. The *subject* member contains the subject line parsed out of the header information. The *body* member is the entire text returned by the server in response to the "BODY" command. The remainder of the members are used by the **NewsFilter** functions. The *counts* array holds the raw keyword match scores. The *sum* is the sum of counts. *Feedback* is used to describe the usefulness of the article to the user (0.0 is useless, 1.0 is interesting). The *score* is the article's ranking based on the type of filter being applied. The *clusterId* holds the segment into which this article falls in the Cluster filter.

The *readArticle()* and *writeArticle()* methods are provided to allow test postings to be added to the user profile. For example, the user could load some articles of particular interest in order to score them, give them a high value using the Feedback option, and then add the profile record to the *newsfilter.dat* file. The *subject* and *body* of the text are concatenated and written out and read as a single block of bytes.

The *getProfileString()* method is used to format the contents of the *counts[]* array, along with the *feedback* and *score* values. This string is a space-delimited text representation that is returned to *addArticleToProfile()* and appended to the *newsfilter.dat* file.

```
public class NewsArticle {
    String id ;  // numeric article ID -- valid only for this server
    String subject ;   // subject line from header
    String body ;      // body or text of news posting
    int    counts[] ;  // raw counts of keys in body
    int    sum ;       // sum of raw counts
    double feedback=0.5 ;   // 0=useless 0.5=neutral, 1.0=interestin
    double score ;   // match score or ranking
    int clusterId;   // cluster this article falls into
    NewsArticle(String id) {  this.id = id ; }

  void readArticle(String fileName) {
     File f = new File(fileName) ;
```

```
   int size = (int) f.length();
   int bytesRead = 0 ;
   try {
     FileInputStream in = new FileInputStream(f) ;

     byte[] data = new byte[size] ;
     in.read(data, 0, size);
     subject = “Subject: ” + fileName  ;
     body = new String(data) ;
     id = fileName ;
     in.close() ;
   }
   catch (IOException e) {
       System.out.println(“Error: couldn’t read news article from
                           + fileName + “\n”) ;
   }
}

void writeArticle(String fileName) {
   File f = new File(fileName) ;
   String dataOut = subject + “ ” + body ;
   int size = (int) dataOut.length();
   int bytesOut = 0 ;
   byte data[] = new byte[size] ;
   body.getBytes(0, body.length(), data, 0) ;
   try {
     FileOutputStream out = new FileOutputStream(f) ;
     out.write(data, 0, size);
     out.flush() ;
     out.close() ;
   }
   catch (IOException e) {
      System.out.println(“Error: couldn’t write news article to ”+
                         fileName + “\n”);
   }
}

  // return the profile data as a string for writing out
String getProfileString() {

   StringBuffer outString = new StringBuffer(“”) ;
   for (int i = 0 ; i < counts.length ; i++) {
      outString.append(counts[i]) ;
      outString.append(“ ”) ;
   }
   outString.append(feedback) ;
   outString.append(“ ”) ;
   outString.append(score) ;

   return outString.toString() ;
```

```
  }
}
```

FilterAgent Class

The **FilterAgent** class is a subclass of **CIAgent**. It provides all of the functions related to management of the user profile data, the keyword scoring, and the construction of the neural network models used for the Cluster and Feedback filters. The **FilterAgent** class is tightly coupled to the **NewsFilter** application. As such, it could be used simply as an intelligent helper class, with all of its methods simply called directly by the **NewsFilter**. However, to illustrate how an intelligent agent can be tightly coupled but still run autonomously, we add a simple signaling protocol for the **NewsFilter** application to initiate long-running **FilterAgent** functions.

The **FilterAgent** defines five additional data members over CIAgent class, a reference to the owning **NewsFilter** application instance, the *clusterNet* which holds the **KMapNet** instance for clustering, and the *scoreNet* member which holds the **BackProp** instance for predictive scoring. The two boolean flags are set by the **NewsFilter** application when the user selects the **Build Cluster Filter** and **Build Feedback Filter** menu options. When either of these flags are set, the **FilterAgent** will notice that fact when it wakes up from its periodic *sleep()* in the *run()* method. This approach allows the agent to autonomously and asynchronously train the neural networks. When the training is complete, the **FilterAgent** signals the completion using the **NewsFilter** *clusterNetBuilt()* and *scoreNetBuilt()* methods.

Remember that the **CIAgent** class implements the **CIAgentEventListener** interface which requires the *ciaEventFired()* method. This method has been enabled to allow other **CIAgents** to initiate the training of the neural networks through the **CIAgentEvent** mechanism.

```
public class FilterAgent extends CIAgent {

 NewsFilter newsFilter ;
 protected KMapNet clusterNet ;
 protected BackProp scoreNet ;
 protected boolean buildClusterNet = false;
 protected boolean buildScoreNet = false ;

 public FilterAgent() { name = “Filter”; }
```

```
 public FilterAgent(String Name) { super(Name); } ;

public void process() {
   stopped = false ;
   // start a thread running
   trace("Starting " + name + "\n") ;
   runnit = new Thread(this) ;
   runnit.start() ;
}

public void stop() {
   stopped = true;
   trace(name + " stopped \n") ;
   runnit.stop() ;
}

// method of Runnable (Thread) interface
public void run() {

     while(stopped == false){     // interval timer
       try {
        runnit.sleep((long) 5 * 1000) ;  // in milliseconds
        if (buildClusterNet) {
          trace(name + ": Starting to build Cluster network\n");
          trainClusterNet() ;  // train the network
          buildClusterNet = false ;
          newsFilter.clusterNetBuilt() ; // signal application
          trace(\n name + ": Completed build of Cluster network\n");
        } else if (buildScoreNet) {
          trace(name + ": Starting to build Score network \n");
          trainScoreNet() ;  // train the network
          buildScoreNet = false ;
          newsFilter.scoreNetBuilt();
          trace(\n name + ": Completed build of Score network\n");
        }
      }
      catch (InterruptedException e)
      {
         // interrupted
      }

   }

}

 public  void ciaEventFired(CIAgentEvent e) {
     trace(name + ":  CIAgentEvent received by " + name +
          " from " + e.getSource() + " with arg " +
            e.getArgObject()) ;
     // set flag for autonomous thread to see
```

```
    String arg = (String)e.getArgObject() ;
    if (arg.equals("buildClusterNet")) {
        buildClusterNet = true ;
    } else if (arg.equals("buildScoreNet"))
        buildScoreNet = true ;
 }

// used to trigger autonomous build of Kohonen Map
public void buildClusterNet() {
   buildClusterNet = true ;
}

// used to trigger autonomous build of BackProp network
public void buildScoreNet() {
   buildScoreNet = true ;
}
```

Previous | Table of Contents | Next

The *scoreArticle()* method takes a single **NewsArticle**, the current list of *keywords*, and the currently selected *filterType* as inputs. The first task is to count the occurrence of each keyword in the article, and then compute the *sum*. Processing continues, based on the filter type. For the keyword filter, the *score* is simply the sum of keyword matches. If the Cluster filter has been created, the article profile data is passed through the Kohonen map neural network to determine the cluster identifier, which is stored in the article object. The actual Cluster filter score cannot be computed in this method, because it relies on the computation of the average score of all articles with the same *clusterId*. These scores are set only in the second *score()* method that takes the entire articles **Vector** as a parameter.

A similar approach is taken when Feedback filtering is enabled and the back propagation model has been built. But in the Feedback case, the neural network output *is* the Feedback score. Thus it is immediately assigned to the *article.score*.

One last function performed in this method is the automatic assignment of *feedback* values. A simple threshold ladder determines this value based on the raw match score. If user-specific values were set before *score()* was called, that information would be lost.

```
// score a single article
void score(NewsArticle article, String[] keywords, int filterType) {

    article.counts = countWordMultiKeys(keywords, article.body);
    int size = article.counts.length ;
    int sum = 0 ;
    for (int i = 0 ; i < size ; i++) sum += article.counts[i];
    article.sum = sum ;  // for convenience

    // based on the type of filter the user wants
    // fill in the score slot of each article
    switch(filterType) {
     case 0:   // keyword
       article.score = (double)sum ;
       break;
     case 1:   // cluster
       if (clusterNet != null) {
          //  pass through network --- get clusterID
         float[] inputRec = new float[size+2] ;
```

```
          for (int i=0 ; i < size ; i++) inputRec[i] =
                                (float)article.counts[i];
          inputRec[size] =   (float)article.sum ;
          inputRec[size+1] = (float)article.feedback ;
          article.score = clusterNet.getCluster(inputRec) ;
        }
        break ;
      case 2:   // feedback
        if (scoreNet != null) {
          //  pass through network --- get score
          float[] inputRec = new float[size+2] ;
          for (int i=0 ; i < size ; i++) inputRec[i] =
                                    (float)article.counts[i];
          inputRec[size] =   (float)article.sum ;
          inputRec[size+1] = (float)article.feedback ;
          article.score = scoreNet.getPrediction(inputRec) ;
        }
        break ;
    }
    // OK, now do an automatic feedback pass
    // so user doesn't have to do it for each article
    // User can override via Feedback menu option
    if (sum == 0) {
      article.feedback = 0.0 ; // negative feedback
    } else if (sum < 2) {
      article.feedback = 0.25 ;
    } else if (sum < 4) {
      article.feedback = 0.50 ;
    } else if (sum < 6) {
      article.feedback = 0.75 ;
    } else article.feedback = 1.0 ;
}
```

The *second score()* method takes a vector of **NewsArticle** objects, the current list of *keywords*, and the currently selected *filterType* as inputs. It calls the first *score()* method for each **NewsArticle** in the **Vector**. Because the single *article score()* method takes care of the unique aspects of each filter type, no filter-specific processing is required in this loop. However, the Cluster filter does require some special processing after the *clusterIds* have been set. This is why there is a call to *computeClusterAverages()* at the end of *score()*.

As indicated by its name, *computeClusterAverages()* walks through the **NewsArticles** and computes the raw match score sum and the number of articles in each cluster. The average match score for each cluster is computed by dividing the sum by the number of articles in the cluster. Once these average scores are computed, another pass over the **NewsArticles** is required to set the

article score to the corresponding average value.

```
  // score all loaded articles using the designated filter
  void score(Vector articles, String[] keywords, int filterType) {

      try {
          String id ;

          Enumeration enum = articles.elements() ;
          while (enum.hasMoreElements()) {
            trace("") ;
            NewsArticle article = (NewsArticle)enum.nextElement() ;
            score(article, keywords, filterType) ;
          }
       }
        catch (Exception e) {
            trace("Exception:" + e) ;
        }
      if (filterType == 1) computeClusterAverages(articles) ;
  }

// compute the average score for each cluster
// and set the score of each article in each cluster
// to that average value
void computeClusterAverages(Vector articles) {

   int numClusters = 4 ;          // we are using 4 for now
   int sum[] = new int[numClusters] ;
   int numArticles[] = new int[numClusters] ;
   double avgs[] = new double[numClusters] ;

   Enumeration enum = articles.elements() ;
   while (enum.hasMoreElements()) {
        NewsArticle article = (NewsArticle)enum.nextElement() ;
        int cluster = article.clusterId ;
        sum[cluster] += article.sum; // sum of counts
        numArticles[cluster]++ ; // bump counter
   }

   // now compute the average score for each cluster
   for (int i=0 ; i < numClusters ; i++) {
      if (numArticles[i] > 0) {
       avgs[i] = (double)sum[i] / (double)numArticles[i];
      } else {
        avgs[i] = 0.0 ;
      }
      trace(" cluster " + i + " avg = " + avgs[i] + "\n") ;
   }
```

```
   enum = articles.elements() ;
   while (enum.hasMoreElements()) {
      NewsArticle article = (NewsArticle)enum.nextElement() ;
      article.score = avgs[article.clusterId] ;
   }

}
```

The *countMultiWordKeys()* method is a workhorse in the **FilterAgent** class. It takes an array of keyword **Strings** and the article text as parameters. It performs a crude but effective search for complete keyword matches. First, an array of **Vectors** is populated with each key, with each keyword of length N stored at the *N*th index of the array. So, for example, the **Vector** at *table[5]* would contain every key of length 5. During the same pass over the keywords, the *keyHash* **Hashtable** is populated with the keyword as the **Hashtable** key, and the length of the keyword (and therefore its index in the table[]) as the **Hashtable** value. This provides a mapping from the keyword to its index in the keys array. Each element in the *counts[]* array is also initialized to 0.

Previous | Table of Contents | Next

The text is searched one word at a time, using a **StringTokenizer** to do the parsing. The length of each word is used to limit the search to keywords of the same length. A linear search is performed at this point. When a case-insensitive match is found, the *keyHash* is used to find the index of the keyword. This index is used to increment the corresponding element in the *counts[]* array. The result of the *countMultWordKeys* is an array of integers, where each element represents the total number of hits for each keyword in the text.

```
// count the number of occurrences of the specified keys
// in the text
int[] countWordMultiKeys(String[] keys, String text) {
     StringTokenizer tok = new StringTokenizer(text) ;
     int keyLen = 1 ;  // for now
     int counts[] = new int[keys.length] ;
     Vector table[] = new Vector[50] ; // up to length 50
     Hashtable keyHash = new Hashtable() ;

     trace("Searching for keywords ... \n") ;

     for (int i=0 ; i < keys.length ; i++) {
        int len = keys[i].length() ;
        if (table[len] == null) table[len] = new Vector() ;
        table[len].addElement(keys[i]) ;
        keyHash.put(keys[i], new Integer(i)) ;
        counts[i] = 0 ;
     }

      while (tok.hasMoreTokens()) {
        String token = tok.nextToken() ;
        int len = token.length();
        if ((len < 50) && (table[len] != null)) {
         Vector searchList = table[len] ;
         Enumeration enum = searchList.elements() ;
         while (enum.hasMoreElements()) {
           String key = (String)enum.nextElement() ;
           if (token.equalsIgnoreCase(key)) {
               Integer index = (Integer)keyHash.get(key);
               counts[index.intValue()]++ ; // found another one
               continue ;
           }
         }
        }
       }

      for (int i=0 ; i < keys.length ; i++) {
```

```
        trace("key = " + keys[i] +
              " count = " + counts[i] + "\n") ;
      }

      return counts ;
    }
```

The *writeProfileDataDefinition()* method generates the *newsfilter.dfn* text file, which defines the layout of user profile data file. A *.dfn* file is read by the **DataSet** class when it loads a file for training neural networks or decision trees. The file contains pairs of data types and field names. Every keyword has an entry for the number of times it appears in the article text. The article feedback value and the current score are also written out. Note that this operation is dependent on what filter is in effect, because the article score will vary, based on the filter selection.

```
// used by neural networks to read newsfilter.dat
void writeProfileDataDefinition(String[] keywords) {

  try {
    FileWriter writer = new FileWriter("newsfilter.dfn") ;
    BufferedWriter out = new BufferedWriter(writer) ;
    for (int i=0 ; i < keywords.length ; i++) {
     out.write("continuous ") ;
     out.write(keywords[i]) ;
     out.newLine() ;
    }
    out.write("continuous ClassField") ;  // feedback
    out.newLine() ;
    out.write("continuous score") ;
    out.newLine() ;
    out.flush() ;
    out.close() ;
  }
  catch (IOException e) {
  System.out.println("Error: couldn't create 'newsfilter.dfn' \n");
  }
 }
```

The *addArticleToProfile()* method takes a single **NewsArticle** as a parameter and appends its filter profile record to the *newsfilter.dat* file. This information is formatted by the *NewsArticle.getProfileString()* method. The *addAllArticlesToProfile()* method calls the *addArticleToProfile()* method for each **NewsArticle** in the articles **Vector**.

```
void addArticleToProfile(NewsArticle currentArt) {

 try {
   FileWriter writer = new FileWriter("newsfilter.dat", true) ;
   BufferedWriter out = new BufferedWriter(writer) ;

   out.write(currentArt.getProfileString()) ;
   out.newLine() ;
   out.flush() ;   out.close() ;
 }
 catch (IOException e) {
System.out.println("Error: couldn't append article to profile \n");
 }

}
void addAllArticlesToProfile(Vector articles) {
try {
   FileWriter writer = new FileWriter("newsfilter.dat", true) ;
   BufferedWriter out = new BufferedWriter(writer) ;

   Enumeration enum = articles.elements() ;
   while (enum.hasMoreElements()) {
     NewsArticle art = (NewsArticle)enum.nextElement() ;
     out.write(art.getProfileString()) ;
     out.newLine() ;

   }
     out.flush() ;
     out.close() ;
 }
 catch (IOException e) {
System.out.println("Error: couldn't append article to profile \n");
 }

}
```

The *trainClusterNet()* method builds the Kohonen feature map used by the Cluster filter. First, a **DataSet** object is instantiated over the *newsfilter.dat* file. Then a **KMapNet** object is created and configured to have as many inputs as are defined in the *newsfilter.dfn* file and to have 4 outputs or clusters. The **KMapNet** object is trained for 20 passes over the data, the network weights are locked by setting the mode to 1, and a single pass is used to check the results. Notice that after each training pass, the **FilterAgent** thread yields*()*. This allows the **NewsFilter** article to continue its processing.

```
void trainClusterNet() {
```

```
    DataSet dataSet = new DataSet(“ProfileData”, “newsfilter”) ;
    dataSet.setDisplay(textArea) ;
    dataSet.loadDataFile() ; // load the data set

    // create a KMap neural network
    // specify the newsfilter.dat as the training data
    // cluster it into 4 clusters

    clusterNet = new KMapNet(“NewsFilter Cluster Profile”);
    clusterNet.textArea1 = textArea ;
    clusterNet.ds = dataSet ;
    clusterNet.numRecs = dataSet.numRecords ;
    clusterNet.fieldsPerRec = dataSet.fieldsPerRec ;
    clusterNet.data = dataSet.normalizedData; // get vector of dat

    // create network, all fields are inputs
    clusterNet.createNetwork(clusterNet.fieldsPerRec, 2, 2) ;
    int maxNumPasses = 20 ;
    int numRecs = clusterNet.numRecs ;

    // train the network
    for (int i= 0 ; i < maxNumPasses ; i++) {
       for (int j=0 ; j < numRecs; j++) {
         clusterNet.cluster() ;   // train
       }
       runnit.yield() ;  // after each pass
    }
     clusterNet.mode = 1 ;  // lock the network weights
     for (int i=0 ; i < clusterNet.numRecs ; i++) {

        clusterNet.cluster() ;   // test
       //  clusterNet.display_network() ;
    }
}
```

The *trainScoreNet()* method is used to build the back propagation regression model used by the Feedback filter. The *newsfilter.dat* file is used as the training data. A neural network is created and trained for 2500 passes over the data (this value may be excessive for large training sets). After each training pass the thread *yields()* to the **NewsFilter**. When training completes, the network is locked and a single test pass is taken over the data. The trained network uses the article profile (the keyword matches and the score) to predict expected usefulness as represented by the feedback value assigned to the article. The output value ranges from 0.0 to 1.0, so articles are sorted in descending order with the score closest to 1.0 at the top of the list and the score closest to 0.0 at the bottom.

```
void trainScoreNet() {
    DataSet dataSet = new DataSet(“ProfileData”, “newsfilter”) ;
    dataSet.setDisplay(textArea) ;
    dataSet.loadDataFile() ; // load the data set

    scoreNet = new BackProp(“NewsFilter Score Model”);
     scoreNet.textArea1 = textArea ;
     scoreNet.ds = dataSet ;
    scoreNet.numRecs = dataSet.numRecords ;
    scoreNet.fieldsPerRec = dataSet.normFieldsPerRec ;
    scoreNet.data = dataSet.normalizedData ;  // get vector of data
    int numOutputs = dataSet.getClassFieldSize() ;
     int numInputs = scoreNet.fieldsPerRec - numOutputs;
     scoreNet.createNetwork(numInputs,2 * numInputs,numOutputs) ;
     int maxNumPasses = 2500 ;  // default -- could be on applet
     int numRecs = scoreNet.numRecs ;
     for (int i = 0 ; i < maxNumPasses ; i++) {
    for (int j=0 ; j < numRecs; j++) {
          scoreNet.process() ;   // train
        }
        runnit.yield() ;  // after each pass
    }

    scoreNet.textArea1.appendText(“\n Passes Completed: ” +
                        maxNumPasses + “  RMS Error = ” +
                        scoreNet.aveRMSError + “ \n”) ;

    scoreNet.mode = 1 ;  // lock the network
    // do a final pass and display the results
    for (int i=0 ; i < scoreNet.numRecs ; i++) {
        scoreNet.process() ;   // test
```

```
    //  scoreNet.display_network() ;
    }
}
```

Discussion

Perhaps the main issue in this application is the relationship between the application code and intelligent agent. One could easily imagine an implementation where the **FilterAgent** did not exist, and its functions were performed by the **NewsFilter** class itself. However, by maintaining the distinct nature of the **FilterAgent** functions, we are in a position to reuse it in a composite agent or multiagent system. While the **FilterAgent** is not as autonomous as the **TimerAgents** or **FileAgents**, it asynchronously trains the neural network models. With just a little additional work, the **CIAgent**-based **FilterAgent** could be modified to build neural network models against any data set, rather than the hardcoded newsfilter data.

In this application, we have developed a modest data mining capability (Bigus 1996). By providing other **CIAgents** that can access databases or other data sources (the **NewsFilter** class provides Internet news sources), an autonomous data mining system could be built. The **TimerAgent** and **FileAgent** developed in Chapter 8 could be used to automatically mine data at set times or only when specific files are modified.

Another design issue for this application is the way keyword matching is done. In the **NewsFilter** class, only complete keyword matches were counted. There are algorithms for doing partial matches so that both singular and plural would be counted for each term. The simple whole-word approach used here also misses cases where punctuation marks are adjacent to the word in the text. Obviously, the better the match information, the more accurately we can filter the articles for the user.

The last issue is the compilation of the user profile data. The Cluster filter and Feedback filter will only perform as well as the profile data used to train the underlying neural network models. With many of the new groups we tested, only a few of the 20 articles had raw keyword matches greater than 0. One solution to this problem is to generate a few canned profile records, which are hand constructed with a representative number of keyword matches and feedback values. This training set could be used as the base data, and selected article profiles could be added.

Summary

In this chapter, we developed an information filter for Internet news groups using a tightly coupled **CIAgent**-based intelligent agent. The main points include:

- The underlying protocol for accessing an Internet news server is the *Network News Transport Protocol* (NNTP). News servers listen at port 119. The basic command strings include **GROUP** for selecting a news group, **STAT** to set the cursor on a specific article, **NEXT** to walk through the news group articles, and **HEAD** and **BODY** to retrieve data concerning the individual articles.
- The **NewsFilter** application class provides three types of filtering of news group articles: *Keyword filtering* based on the number of keyword matches, *Cluster filtering* based on grouping similar articles using neural clustering, and *Feedback filtering* using a neural prediction model to score articles based on relevance.
- The **FilterAgent** extends the CIAgent class by providing methods for keyword scoring and for autonomously training Kohonen map and back propagation neural networks. The FilterAgent also maintains the user profile text files, *newsfilter.dat* and *newsfilter.dfn*, used for training the neural networks.

Exercises

1. How would you modify the **NewsFilter** application to be more flexible? Add the capability for the user to set the number of news articles to read through the GUI.

2. While browsing a set of news groups, build your own user profile, selecting representative articles across the Useless to Interesting spectrum. Be sure to set the feedback levels appropriately. Build the Feedback filter, and see how it performs. Add 20 hand-constructed profile records to the *newsfilter.dat* file, and rebuild the Feedback filter. Did the filtering improve?

3. Enhance or replace the *countWordMultiKeys()* method to improve the keyword match accuracy of the **NewsFilter**. Will the *int counts[]* array still work for your new approach? What if you add partial word match capabilities?

Chapter 10
MarketPlace Application

This chapter focuses on the issues involved when multiple autonomous agents interact in multiagent systems. The application is an intelligent agent marketplace, where buyer and seller agents cooperate and compete to process sales transactions for their owners. A facilitator agent is developed to act as a manager for the marketplace. The buyer and seller agents range from using hardcoded logic to rule-based inferencing in their negotiation strategies.

Introduction

Multiagent systems provide a complex and interesting environment for evaluating intelligent agent behavior. While such systems have been used in computer-aided design systems, the domain knowledge required to understand those applications is quite specialized. In this chapter, we will build a multiagent system for an electronic marketplace because everyone is familiar with buying and selling things. We develop a **FacilitatorAgent** which manages the marketplace, and include several kinds of **BuyerAgents** and **SellerAgents** to interact there. All of these intelligent agents are derived from the **CIAgent** base class presented in Chapter 7. The Buyer and Seller agents are differentiated primarily by the sophistication of their negotiation strategies, ranging from simple, hardcoded logic to forward-chaining rule inferencing.

The **FacilitatorAgent** is the go-between or matchmaker between the Buyers and Sellers. All agents must register with the Facilitator before they can have any interactions with other agents in the marketplace. Sellers advertise their desire to sell products or services with the **FacilitatorAgent**, while Buyers ask the Facilitator to recommend a prospective Seller. Once the Buyer and Seller agents have been introduced by the Facilitator, they still communicate indirectly through the FacilitatorAgent.

Figure 10.1 CIAgent MarketPlace application.

Figure 10.1 shows the main panel of the **CIAgent** MarketPlace application. The two **TextArea** controls are used to display messages from the Facilitator and the **BuyerAgents** and **SellerAgents** in the marketplace. The **MenuBar** options include **Start** and **Stop**, a selection of the **Detail** or **Summary** message content, and the choice of three types of Buyers and Sellers to place in the marketplace. These agents can be selected in any combination of Basic, Intermediate, and Advanced Buyers and Sellers. The default setting is for a single basic **BuyerAgent** and basic **SellerAgent** to be in the marketplace. Up to six independent and autonomous agents can be placed in the marketplace at one time.

When the user selects two to six agents and selects the **Start** option from the **Actions** menu, a single **FacilitatorAgent** is created along with the selected Buyer and Seller agents. The **SellerAgents** advertise the items they have to sell by sending messages to the Facilitator and initialize their internal inventory of items. The **BuyerAgents** initialize their own shopping lists. At this point, the marketplace is open for business.

All of the agents are running their own threads which awaken at specified intervals. The first sales negotiation takes place when one of the **BuyerAgents** wakes up and takes an item from its wishList. It then asks the Facilitator to recommend a **SellerAgent** who has advertised its ability to sell that item. If more than one Seller has advertised an item, the Facilitator randomly selects one and returns the name of this **SellerAgent** to the Buyer. This starts the communication between Buyer and Seller as they try to agree on a price and close the deal. The Facilitator acts as an intermediary between all Buyer and Seller communications. This allows us to watch the exchange of messages between Buyer and Seller agents and display a trace of these interactions in the top **TextArea**.

All of the communications between Buyers, Sellers, and the Facilitator use the **CIAgentEvent** and **CIAgentEventListener** interface described in Chapter 7. The argument object that is passed with the **CIAgentEvents** is a new object called a **CIAgentMessage**, which is modeled after a standard KQML message

packet. Although we are not parsing KQML messages in this application, the use of the **CIAgentMessage** class should give you a good feel for what a KQML agent application would look like (see Chapter 6 for a description of KQML). Remember that KQML uses performatives to indicate the action it wants another agent to take on its behalf. While KQML specifies the format and some of the content of these interactions, the nitty-gritty details of the Buyer-Seller negotiation is up to the application to define. The sales negotiation convention used in the MarketPlace application is described in the following section.

After all agents are registered with the Facilitator and the Sellers advertise their wares, the following **CIAgentMessage**s are exchanged:

1. Buyer asks the Facilitator to **recommend-one** seller for an item, P.
2. Facilitator **tells** the Buyer the name of the Seller.
3. Buyer **asks** the Seller (through the facilitator) if the Seller has an item P for sale.
4. Seller either **makes-offer** to the Buyer, passing the item P, a unique item id, and an initial asking price, or will **deny** that he has item P for sale.
5. Buyer then can either **accept** the offer by echoing the offer back to the Seller, or make a **counter offer** (with a different price) to the Seller.
6. The Seller can either **accept** the offer, make a **counter offer**, or **reject** the offer.
7. If the Seller **accepts** he sends a **tell** message to the Buyer, and the sales transaction is complete.
8. If the Seller **rejects** the offer, the negotiation is over.

Note that the Buyer and Seller never communicate directly, but always use the Facilitator as a go-between in a sales negotiation.

Figure 10.2 MarketPlace example 1.

An Example

The following example illustrates the interactions between a single basic **BuyerAgent** and a single basic **SellerAgent**. The application is started by

entering the following command:

```
> java MarketPlace
```

When the MarketPlace main window comes up, the application is started by selecting the **Start** option in the **Actions** menu. Figure 10.2 shows the main window right after start is selected, but before any sales transactions begin.

Previous | Table of Contents | Next

The **BuyerAgent** goes to sleep for approximately 5 seconds before it wakes up and picks the first item from its shopping list. This pause between negotiations is used to make it easier to follow the messages exchanged between agents. The Buyer has a short list of items to buy, a guitar for $100, a set of drums for $200, and another guitar for $100 (must be starting a power trio?). It turns out that the **SellerAgent** just happens to have two guitars and one set of drums in his inventory. As described earlier, the **BuyerAgent** asks the Facilitator to recommend the name of a **SellerAgent** who has a guitar to sell. Because there is only one **SellerAgent**, the Facilitator returns the name of the **SellerAgent** to the **BuyerAgent** and things progress from there. When the first item is sold, the **BuyerAgent** goes back to sleep for another 5 seconds. When the **BuyerAgent** wakes up, it pulls the next item off the shopping list and another set of communications occurs. This sequence continues until the **BuyerAgent** has exhausted its shopping list. When the marketplace has quiesced, and no activity other than the Buyer proudly proclaiming its purchases remains, the MarketPlace application run can be halted by selecting the **Stop** menu option under the **Actions** menu.

When a **SellerAgent** sells an item, it sends an unadvertise message to the Facilitator, and so the **SellerAgent** is removed from the list. Note, if a Seller has multiple items of the same type, it advertises multiple times, so for each sale and each corresponding unadvertise message, the Facilitator always has an up-to-date list of Sellers and items available.

Figure 10.3 contains the trace messages from the Facilitator that appear in the top **TextArea**. You can see the **FacilitatorAgent** is started when the *process()* method is called and halted when the *stop()* method is called. The first transaction between the Buyer and Seller is reflected in lines 6 through 12. Figure 10.4 contains the trace messages from the agents in the MarketPlace. Both agents are started by the *process()* method. When the **BuyerAgent** wakes up, it announces that it wants to buy a guitar. The exchange of offers follows, and the transaction completes. Note that this is the default **Summary** level of trace messages. If the **Details** option was selected, additional information would be displayed, including the contents of the **CIAgentMessages**.

Figure 10.3 Facilitator trace log from example 1.

Figure 10.4 MarketPlace trace log from example 1.

As you can see, the basic **BuyerAgent** and basic **SellerAgent** came to a price of $175 for the guitars, and $325 for the drums. These prices are much higher than the Buyer's goal prices. Similar runs can be done for the Intermediate Buyer and Intermediate Seller, or the Best Buyer can be matched up against the Basic Seller. In this case, the Buyer does much better; as shown in Figure 10.5, it purchases the three musical instruments for $500.

Figure 10.5 MarketPlace trace log from example 2.

The last example presents the negotiations between the **BestBuyerAgent** and the **BestSellerAgent**. Figure 10.6 shows this exchange. There is much more haggling going on as the strategies implemented in the rule-bases of these agents are more complex. The **Best-BuyerAgent** gets the better of the **BestSellerAgent**, completing its three purchases for only $470.

Figure 10.6 MarketPlace trace log from example 3.

These examples show the basic operation and interactions between agents in the MarketPlace. In the following sections, the design and implementation details of the **FacilitatorAgent**, **BuyerAgent**, **SellerAgent**, the enhanced Buyer and Seller agents, and the MarketPlace main application itself are described.

FacilitatorAgent

The **FacilitatorAgent** uses the Singleton design pattern (Gamma et al. 1995), which insures that only a single instance of **FacilitatorAgent** exists at one time. This prevents an impostor Facilitator from redirecting sales traffic to its own favorite agents, and ensures that all Buyers and Sellers can find the marketplace through the global **FacilitatorAgent** instance. While we favor a model-view-controller design, where the agents in the marketplace represent the model and the **java.awt. Frame** and GUI represents the view and controller, we mix these elements in this application for simplicity.

The Facilitator contains two **Hashtables**, one which is a registry for all agents in the marketplace, and one which contains the communities of interest in the marketplace. For example, there may be a set of **SellerAgents** which sells musical instruments, and another set which sells airplane tickets. These are grouped into communities or domains. **SellerAgents** are added to these communities by advertising their willingness or ability to sell a product in that domain. Just because an agent advertises something, however, doesn't mean that it can really deliver that product or service. Besides the obvious case where deviousness is involved, a sincere Seller may advertise a one-of-a-kind item. A Buyer may then ask the Facilitator to recommend a Seller for this item. Negotiations could proceed between the Seller and the Buyer, resulting in a sale. Meanwhile, another Buyer could ask the Facilitator to recommend a seller and when it contacts the Seller asking for a price, the Seller will deny that it has such an item. This scenario might be prevented if the Seller always retracts its advertisements. This application provides this capability. However, if multiple transactions are going on at the same time, one Buyer will have to be disappointed (or the Seller could sell the same item to two Buyers, which is unethical but very profitable in the short term).

As a subclass of **CIAgent**, the **FacilitatorAgent** is **Runnable**. The *process()* method is used to start its thread. It sleeps for 10 seconds and then wakes up to see if anything exciting is happening. Note that this is not strictly necessary for this application. The **FacilitatorAgent** runs in the main application thread by synchronously handling **CIAgentEvents**. An alternative design for long-running transactions would be to spin a new thread whenever an event needed to be processed. But this gets a little involved and is getting far afield from the exploration of the fundamentals of multiagent systems.

```
public class FacilitatorAgent extends CIAgent {

  private static FacilitatorAgent instance = null ;  // Singleton

  // in Singleton design pattern, used to get single instance
  static public FacilitatorAgent Instance() {
        if (instance == null) {
            instance = new FacilitatorAgent("Facilitator");
        }
        return instance ;
  }

  // Note: can't be used as a Java Bean without public constructor

  protected FacilitatorAgent() { super() ; name = "Facilitator"; }
  protected FacilitatorAgent(String Name) { super(Name); } ;

  public java.awt.TextArea textArea ;
  public synchronized void trace(String msg) {
       textArea.appendText(msg) ;
  }

  Random random = new Random() ;  // used to select agents

  Hashtable allAgents = new Hashtable() ;
  Hashtable communities = new Hashtable() ;
  CIAgentMessage msg ;    // current message being processed

  public void reset() {
   allAgents = new Hashtable() ;  // clear all agents
   communities = new Hashtable() ; // clear all communities
  }

  public void process() {
    stopped = false ;
```

```
    // start a thread and send an update message every interval secs
    trace("Facilitator: process() \n") ;
    runnit = new Thread(this) ;
    runnit.start() ;
 }

 public void stop() {
    stopped = true;
    trace("Facilitator: stopped \n") ;
    runnit.stop() ;
}

 // method of Runnable (Thread) interface
 public void run() {
    while(stopped == false){
        try {
           Thread.sleep((long)10*1000);  // come alive every 10 secs
           if (traceLevel > 0) trace("Facilitator: active \n") ;
        }
        catch (InterruptedException e)
        {
          // interrupted
        }
       // do any housekeeping here !!!
    }
 }
```

The *ciaEventFired()* method is required by the **CIAgentEventListener** interface. The **CIAgentEvent** object contains the source object that fired the event (retrieved using the *getSource()* method on the **CIAgentEvent** object), and the argument object which is retrieved using the *getArgObject()* method. In all cases in the MarketPlace application, this argument object is a **CIAgentMessage** object, roughly equivalent to a KQML message packet. Once the **CIAgentMessage** is extracted from the **CIAgentEvent**, the *route()* method is called to process the message.

```
public  void ciaEventFired(CIAgentEvent e) {
     if (traceLevel > 0) {
       trace("Facilitator:  CIAgentEvent received by " + name +
             " from " + e.getSource() + " with args " +
               e.getArgObject() + "\n") ;
     }
     Object arg = e.getArgObject() ;
     msg = (CIAgentMessage)arg ;
     if (traceLevel > 0) msg.display() ;
     route(msg) ;
 }
```

```
 public static synchronized void register(CIAgentEvent e) {
     if (instance == null) instance = new FacilitatorAgent() ;
     instance.allAgents.put(((CIAgent)e.getSource()).getName(),
                          e.getSource()) ;
     ((CIAgent)e.getSource()).addCIAgentEventListener(instance) ;
     instance.addCIAgentEventListener((CIAgent)e.getSource()) ;
 }
```

The *route()* method takes a **CIAgentMessage** as a parameter. The method gets a reference to the sending **CIAgent** from the **Hashtable** of registered agents. The Facilitator handles three types of performatives in this method, **advertise**, **unadvertise**, **recommend-one**, in addition to its message-routing function. The advertise and unadvertise performatives either add or remove the sending agent from a community. Special logic is needed to handle the cases where the agent is the first to be added, or last to be removed from a community.

The recommend-one performative first checks to see if there are any agents in the community of interest. If so, the Facilitator determines how many agents have advertised and then rolls a random number to select one to recommend to the **BuyerAgent**. The case where a **BuyerAgent** asks for a **SellerAgent**, and none have advertised, results in an undefined state in the sample application. The **BuyerAgent** is expecting a **tell** response from the Facilitator which never comes. A reasonable solution here would be to send a **deny** message back to the **BuyerAgent**. However, the **BuyerAgent** state logic would have to be updated to deal with this correctly.

```
public synchronized void route(CIAgentMessage msg) {
  CIAgent sender = (CIAgent)allAgents.get(msg.sender) ;
// agent wants to say they can handle questions concerning content
if (msg.performative.equals("advertise")) {
    trace("Facilitator: adding " + msg.sender +
                        " to " + msg.content + " community \n") ;
   if (communities.containsKey(msg.content)) {
       Vector agents = (Vector)communities.get(msg.content) ;
       agents.addElement(sender) ;
   } else {
       Vector agents = new Vector() ;
       communities.put(msg.content, agents) ;
       agents.addElement(sender) ;
   }
   return ;
 }

// agent wants to remove themselves from the community
```

```
if (msg.performative.equals(“unadvertise”)) {
    trace(“Facilitator: removing ” + msg.sender +
                        “ from ” + msg.content + “ community \n”) ;

   if (communities.containsKey(msg.content)) {
       Vector agents = (Vector)communities.get(msg.content) ;
       agents.removeElement(sender) ;
       if (agents.size() == 0) communities.remove(msg.content) ;
   }
   return ;
 }

 // agent wants facilitator to recommend an agent
 // randomly pick one from the list of advertising seller agents
 if (msg.performative.equals(“recommend-one”)) {
   String item = msg.content ;
   if (communities.containsKey(msg.content)) {
     Vector agents = (Vector)communities.get(msg.content) ;
     int num = agents.size() ;
     int index;
     if (num > 1) {
      double rand = random.nextDouble() ;
       index = (int)(rand * num) ;
      } else {
       index = 0 ;
     }
     CIAgent agent = (CIAgent)agents.elementAt(index) ;
     msg.performative = “tell” ;
     msg.content = agent.name ;
     msg.receiver = msg.sender ;
     msg.sender = name ; // show facilitator is sender of msg
     // construct an anwser message and send it to sender agent
     trace(“Facilitator: Recommended ” + agent.name +
                         “ to ” + msg.receiver +
                         “ for ” + item + “\n”) ;
     sender.ciaEventFired(new CIAgentEvent(this,msg)) ;

   } else {
      trace(“Facilitator: there are no agents advertising ” +
                          msg.content + “\n”) ;
   }
   return ;
 }
 trace(“Facilitator: routing ” + msg.performative +
       “ message from ” + msg.sender + “ to ” +
       msg.receiver + “\n”) ;
 // route the message to the receiver agent
 CIAgent receiver = (CIAgent)allAgents.get(msg.receiver) ;
 if (receiver != null) {
    receiver.ciaEventFired(new CIAgentEvent(this,msg)) ;
```

```
 } else {
    // should tell sender that receiver could not be found?
    trace("Facilitator: receiver " +
           msg.receiver + " is unknown! \n") ;
 }

}
```

Previous | Table of Contents | Next

CIAgentMessage

The **CIAgentMessage** class is used as the argument object in a **CIAgentEvent**. It is basically a collection of data that corresponds to the major slots of a KQML message. The class has two constructors, with the only difference being that one allows the specification of the language and ontology, and the other does not. Each **CIAgentMessage** must specify a *performative*, a *content* string, the *sender* agent's name, and the *receiver* agent's name. In a buyer/seller transaction these would be the names of the Buyer and Seller agents, even though the messages are routed through the **FacilitatorAgent**. The *replyWith* parameter tells the *receiver* object to use that **String** in the *inReplyTo* slot in the reply message. All of the parameters are publicly visible, although they could have been shielded by making them private or protected and providing accessor methods. **EventObjects** are usually immutable, so this would be the desired approach in a real application.

```
public class CIAgentMessage {

   String performative ;
   String content ;
   String inReplyTo ;
   String language ;
   String ontology ;
   String receiver ;
   String replyWith ;
   String sender ;

   CIAgentMessage(String Performative, String Content,
                  String InReplyTo, String Language,
                  String Ontology, String Receiver,
                  String ReplyWith, String Sender) {
         performative = Performative ;
         content = Content ;
         inReplyTo = InReplyTo ;
         language = Language ;
         ontology = Ontology ;
         receiver = Receiver ;
         replyWith = ReplyWith ;
         sender = Sender ;
   }

   CIAgentMessage(String Performative, String Content,
```

```
                String InReplyTo, String Receiver,
                String ReplyWith, String Sender) {
            performative = Performative ;
            content = Content ;
            inReplyTo = InReplyTo ;
            receiver = Receiver ;
            replyWith = ReplyWith ;
            sender = Sender ;
    }

    public void display() {
      System.out.println("performative: " + performative + "\n" +
      "content: " + content + "\n" +
      "inReplyTo: " + inReplyTo + "\n" +
      "language: " + language + "\n" +
      "ontology: " + ontology + "\n" +
      "receiver: " + receiver + "\n" +
      "replyWith: " + replyWith + "\n" +
      "sender: " + sender + "\n");
    }
}
```

BuyerAgent

The **BuyerAgent** class contains the base functionality for all of the Buyers. The major data members include the *wishList*, a **Vector** of items the agent wants to buy and desired purchase prices; the *inventory*, which contains all of the items the **BuyerAgent** has purchased; and the *negotiations*, a **Hashtable** of negotiations in progress. The *process()* method is used to register the **BuyerAgent** with the global Facilitator, initialize the *wishList* with instances of **BasicNegotiation** objects, and start the agent's **Thread** running. The *run()* method provides the body of the agent's thread. It goes to sleep for 5 seconds, and then if it still has items on its *wishList* and no negotiation is in progress, it takes the first item off the *wishList* and kicks off a negotiation by asking the Facilitator to recommend a Seller for the item. It does this by first instantiating a **CIAgentMessage** object and setting the slots appropriately, and then using it as the argument for a new **CIAgentEvent** object. When the **BuyerAgent** registered with the Facilitator, the Facilitator added itself to the BuyerAgent's **CIAgentEventListener** list. So, calling the *notifyCIAgentEventListener()* method results in the Facilitator receiving a copy of the **CIAgentMessage**.

The Facilitator will reply to the **BuyerAgent** by sending a **CIAgentMessage**. All such messages are received through the *ciaEventFired()* method.

```
public class BuyerAgent extends CIAgent {

  CIAgentMessage msg ;    // current message being processed
  BasicNegotiation current;

  Vector wishList = new Vector() ;

  BasicNegotiation pending = null ; // item waiting for seller

  Hashtable inventory = new Hashtable() ; // items we have purchased
  long totalSpent = 0 ;                   // total money spent

  Hashtable negotiations = new Hashtable() ; // transaction history
                                              // key is the item id

  public BuyerAgent() { name = “Buyer”; }

  public BuyerAgent(String Name) { super(Name); } ;

  public void initialize() {} ;

  public void process() {
    initialize() ;  // call any specific initialization code
    stopped = false ;
    // start a thread running and send message every interval secs
    trace(name + “: process() \n”) ;
    msg = new CIAgentMessage(“register:”, name , null , null ,
                             null , null) ;
    CIAgentEvent e = new CIAgentEvent(this, msg) ;
    FacilitatorAgent.register(e) ;  // add this agent to Facilitator

    // add items to buy on wish list
    wishList.addElement(new BasicNegotiation(“guitar”, 100)) ;
    wishList.addElement(new BasicNegotiation(“drums”, 200)) ;
    wishList.addElement(new BasicNegotiation(“guitar”, 100)) ;
    runnit = new Thread(this) ;
    runnit.start() ;
 }
 public void stop() {
    trace(name + “: stopped \n”) ;
    stopped = true ;
    runnit.stop() ;
}

 // method of Runnable (Thread) interface
 public void run() {
    while(stopped == false){
      try {
         Thread.sleep((long)5 * 1000);  // come alive every 5 secs
         if (traceLevel > 0) trace(name + “:  active \n”) ;
```

```
      }
      catch (InterruptedException e)
      {
        // interrupted
      }
      if ((wishList.size() > 0) && (pending == null)) {
        current = (BasicNegotiation)wishList.firstElement() ;
        pending = current ; // we have a pending negotiation
        wishList.removeElementAt(0);
        trace(name + " is looking to buy " + current.offer.item +
              "\n");
        msg = new CIAgentMessage("recommend-one", current.offer.item
                       null , "Seller" , current.offer.item , name)
        CIAgentEvent e = new CIAgentEvent(this, msg) ;
        notifyCIAgentEventListeners(e);  // signal listeners
      }
      trace(name + " has purchased " + inventory.size() +
             " items for " + totalSpent + "\n");
  }

 }
...

};
```

Previous | Table of Contents | Next

The *ciaEventFired* method is the **BuyerAgent's** method for receiving messages from other agents in the marketplace. Two levels of tracing information are supported in the MarketPlace application, a **Summary** level of 0, and a **Details** level of 1. If **Details** are desired, a trace message is displayed. Next the **CIAgentMessage** is extracted from the **CIAgentEvent** and the *processMessage()* method is called, passing the message as a parameter.

```
public  void ciaEventFired(CIAgentEvent e) {
    if (traceLevel > 0) {
      trace(name + ":  CIAgentEvent received by " + name +
            " from " + e.getSource() + " with args " +
            e.getArgObject()) ;
    }
    Object arg = e.getArgObject() ;
    msg = (CIAgentMessage)arg ;
    if (traceLevel > 0) msg.display() ; // show message contents
    processMessage(msg) ;
 }
```

The *processMessage()* method is the workhorse of the **BuyerAgent**. There are several conditions which it must detect and handle. The first is when the Facilitator responds to the recommend request. This message is a tell performative and the content is the name of a **SellerAgent** that has advertised the desired item. In this case, the Seller is asked (through the Facilitator) how much it wants for the item.

Otherwise, the message must contain an offer or counter-offer from a Seller. First an instance of an **Offer** is created. An **Offer** contains a triplet of item name, item identifier, and offer price, along with the name of the Seller making the offer. The **Offer** constructor takes the **CIAgentMessage** as a parameter and extracts the information from the sender and content slots of the message. Now that an **Offer** exists, we need to determine if this is the first offer of a new negotiation or whether this is a counter-offer in an ongoing negotiation. If a **BasicNegotiation** object exists in the *negotiation* **Hashtable** with the same item identifier contained in the **offer**, the **BasicNegotiation** object is updated by passing the latest **offer** using the *newOffer()* method. If the item id isn't in the **Hashtable**, it must be the first offer from the **SellerAgent**. The **BuyerAgent** takes the pending negotiation, makes it the current one, sets pending to null, and places the negotiation on the active list.

The **Offer** can contain one of three performatives: make-offer, accept-offer, or reject-offer. If the message is an accept-offer, the sale is acknowledged by sending a tell message back to the seller, the item is put in the inventory of purchases, the *negotiation* is removed from the active list, the money spent is added to the tab, and the message is sent. The *pending* negotiation is also set to null, so that the next time the **BuyerAgent** wakes up, it can start a new negotiation if there are still items on its *wishList*. This behavior is an arbitrary design decision. We could just as easily have decided to immediately pull another negotiation off the *wishList*, but we chose this design to allow the transactions to be more spread out in time.

Another alternative is that the Seller rejects the last offer. In our design, this ends the negotiations. Once again, this is an arbitrary decision, but we wanted to make explicit when the Seller decides to end a negotiation and not have the **BuyerAgent** try to make counter-offers to extend the negotiation. The **BuyerAgent** cleans up the negotiation from the active list and places it back on the *wishList* so it can try again by asking the Facilitator to recommend another Seller for the item.

The third alternative is that the Seller makes an offer and the **BuyerAgent** needs to negotiate using the (what else?) *negotiate()* method. The sophistication of the processing logic in this method is the primary point of differentiation between our basic **BuyerAgent**, the **BetterBuyerAgent**, and the **BestBuyerAgent** classes.

```
public void processMessage(CIAgentMessage msg) {

 // facilitator has found a seller for item
 if (msg.sender.equals("Facilitator")) {
   if (msg.performative.equals("tell" )) {
     // OK, ask seller what he wants for the item
     CIAgentMessage answer =
       new CIAgentMessage("ask", pending.offer.item, msg.replyWith,
                          msg.content , pending.offer.item , name);
     CIAgentEvent e = new CIAgentEvent(this, answer) ;
     notifyCIAgentEventListeners(e); // respond through Facilitator
     return ;
   }
 }

   // seller is denying he has any items to sell
   // this is a response to an "ask"
   if (msg.performative.equals("deny" )) {
```

```
      trace(name + ": Seller denied our 'ask' about " +
            msg.content + "\n") ;
      wishList.addElement(pending); // put back on wish list
      pending = null ;  // try again
      return;
   }

  Offer offer = new Offer(msg) ;  //
  trace(name + ": Offer from " + offer.sender + ": content is " +
        offer.item + " " + offer.id + " " + offer.price + "\n") ;

  if (negotiations.containsKey(offer.id)) {
     current = (BasicNegotiation)negotiations.get(offer.id) ;

  } else {
     current = pending ; // get pending negotiation
     pending = null ;  // no items pending
     negotiations.put(offer.id, current) ; // place on active list
}
   current.newOffer(offer) ; // update the negotiation object

   // seller has agreed and sales transaction is complete
   // transfer item to buyer, seller accepts our last offer
   if (msg.performative.equals("accept-offer" )) {
     trace(name + ": OK -- sale of item " + offer.item +
           " with id " + offer.id +
           " at price of " + offer.price + " is complete \n") ;
     CIAgentMessage answer =
           new CIAgentMessage("tell", msg.content , msg.replyWith ,
                              msg.sender , offer.item , name) ;
     trace(name + ": " + offer.item + " purchased! \n") ;
     inventory.put(offer.id, current) ;
     negotiations.remove(offer.id) ;
     totalSpent += offer.price ;
     CIAgentEvent e = new CIAgentEvent(this, answer) ;
     notifyCIAgentEventListeners(e); // respond through Facilitator
   }

   // seller is making a counter offer
   // we can either, make a counter offer, accept, or reject
   if (msg.performative.equals("make-offer" )) {
       negotiate(offer, msg) ;
   }

   // seller is rejecting our last offer - no counter offer
   if (msg.performative.equals("reject-offer" )) {
     trace(name + ": " + msg.sender + " rejected our last offer ");
     negotiations.remove(offer.id) ;  // remove from active list
     wishList.addElement(current); // put back on wish list
   }
```

```
}
```

The basic **BuyerAgent** checks if the Seller's offer price is lower than the *strikePrice*, which is the desired maximum price. If it is, it accepts the offer by echoing the message back to the Seller, assuming that the **SellerAgent** will accept the offer it just made. If the offer price is higher than the buyer is willing to pay, the **BuyerAgent** must make a counter offer (remember, in this design, only the Seller can reject an offer). The basic agent simply lops $25 off the Seller's price and makes an offer in the hopes that the Seller will accept. Later in this chapter, we will explore the more sophisticated strategies used by our BetterBuyer and BestBuyer agents.

```
// this method must be overridden to provide alternate
// negotiation strategies
void negotiate(Offer offer, CIAgentMessage msg) {

   if (offer.price < current.strikePrice) {
     // accept the offer -- by repeating it to the seller
     CIAgentMessage answer =
          new CIAgentMessage("make-offer", msg.content,
              msg.replyWith , msg.sender , offer.item , name) ;
     CIAgentEvent e = new CIAgentEvent(this, answer) ;
     notifyCIAgentEventListeners(e); // respond through Facilitator
   } else {
     // make a counter offer
     current.lastOffer = offer.price - 25 ;  //
     CIAgentMessage answer =
          new CIAgentMessage("make-offer", offer.item + " " +
              offer.id + " " + current.lastOffer, offer.item,
              msg.sender , offer.item , name) ;
     CIAgentEvent e = new CIAgentEvent(this, answer) ;
     notifyCIAgentEventListeners(e); // respond through Facilitator
  }
}
```

SellerAgent

The **SellerAgent** class contains the base functionality for all Sellers. The major data members include the *inventory*, a **Hashtable** of items the agent wants to sell and desired sales prices, and the *negotiations*, a **Hashtable** of negotiations in progress. The *process()* method is used to register the **SellerAgent** with the global Facilitator, initialize the *inventory* with instances of **BasicNegotiation** objects, advertise the items to the Facilitator, and to start the agent's **Thread**

running. The *run()* method provides the body of the agent's thread. It goes to sleep for 15 seconds and then reports the status of its inventory.

After the **SellerAgent** advertises its wares to the Facilitator, it must wait for a prospective **BuyerAgent** to contact the Facilitator looking for a recommendation of a likely Seller. When the Facilitator passes the **SellerAgent**'s name to the **BuyerAgent**, the Buyer sends an ask message to start the negotiations. Like the **BuyerAgent** described earlier, the **SellerAgent** receives any message as an argument from a **CIAgentEvent** through the *ciaEventFired()* method. The *processMessage()* method interprets the message and determines the appropriate response.

```
public class SellerAgent extends CIAgent {

 private long seed = 0 ;

 CIAgentMessage msg ;    // current message being processed
 BasicNegotiation current;
 long income = 0 ;   // total money earned
 Vector inventory = new Vector() ; // items we have for sale

 Hashtable negotiations = new Hashtable() ; // transaction history
                                             // key is the item id

 public SellerAgent() { name = "Seller"; }

 public SellerAgent(String Name) { super(Name); } ;

 public void process() {
   initialize() ;  // call any specific initialization code
   stopped = false ;
   // start a thread running and send a message every interval secs
   trace( name + ": process() \n") ;
   msg = new CIAgentMessage("register:", name , null ,
                            null, null , null) ;
   CIAgentEvent e = new CIAgentEvent(this, msg) ;
   FacilitatorAgent.register(e) ;  // add agent to Facilitator

   inventory.addElement(new BasicNegotiation("guitar",100));

   inventory.addElement(new BasicNegotiation("drums", 225));

   inventory.addElement(new BasicNegotiation("guitar",100));

   // advertise all items for sale
   Enumeration enum = inventory.elements() ;
   while (enum.hasMoreElements()) {
```

```
      current = (BasicNegotiation)enum.nextElement() ;
      msg = new CIAgentMessage("advertise", current.offer.item,
                           null , null , current.offer.item , name) ;
      e = new CIAgentEvent(this, msg) ;
      notifyCIAgentEventListeners(e);   // signal interested observe
    }

    runnit = new Thread(this) ;
    runnit.start() ;
}

public void stop() {
    stopped = true;
    trace(name + " stopped\n") ;
    runnit.stop() ;
}

// method of Runnable (Thread) interface
public void run() {
    while(stopped == false){
       try {
          Thread.sleep((long)15 * 1000); // come alive every 15 secs
          if (traceLevel > 0) trace(name + ": active \n") ;
       }
       catch (InterruptedException e)
       {
         // interrupted
       }
       trace(name + " has " + inventory.size() +
             " items in inventory. \n");
       trace(name + " has earned " + income + ".\n ");
    }

}

 public  void ciaEventFired(CIAgentEvent e) {
     if (traceLevel > 0 ) {
        trace(name + ":  CIAgentEvent received by " + name +
              " from " + e.getSource() + " with args " +
              e.getArgObject() + "\n") ;
     }
     Object arg = e.getArgObject() ;
     msg = (CIAgentMessage)arg ;
     if (traceLevel > 0) msg.display() ;
     processMessage(msg) ;
 }

 // generate a unique id for an item in inventory
 String genId() {
    seed++ ;
```

```
    return name+seed;
  }

 ...

};
```

The *processMessage()* method must take care of two basic cases. First is to respond to an initial ask from a **BuyerAgent**, and the other is to respond to an offer. In the first case, there is a lot of housekeeping to do when a **BuyerAgent** asks about an item. First, it must check to see if it still has any of the desired items in inventory (as described earlier, it could have just sold the last one). Assuming it has the item, it then generates a unique item identifier by calling the *genID()* method. The item identifier is the **SellerAgent**'s name concatenated with a nonrepeating integer (e.g., Seller1, or BetterSeller3). This item identifier is then used as the **Hashtable** key for both the Buyer and Seller negotiation objects. The item's **BasicNegotiation** object is removed from the inventory list, the item id member is set, and the lastOffer field is initialized to the minimum selling price plus $100. The **SellerAgent** then places the **BasicNegotiation** on the active negotiations list, and sends a **make-offer** message back to the **BuyerAgent** through the Facilitator that is a registered **CIAgentEventListener**.

Previous | Table of Contents | Next

In the second case, there is a negotiation in progress. The **SellerAgent** instantiates an **Offer** from the **CIAgentMessage** and retrieves the negotiation from the active negotiations list, using the item id as the **Hashtable** key. If there is no **BasicNegotiation** object with that item id, it must have just been sold, so the SellerAgent sends a **reject-offer** message to the **BuyerAgent**. Otherwise, it updates the **BasicNegotioation** with the current offer. Next, it determines whether the Buyer is making a counter offer or simply acknowledging a previous **accept-offer** message. If it is a **tell**, it must be the latter case, an item was just sold. The **BasicNegotiation** is removed from the active negotiations list, and the purchase price is added to the sales total.

If the performative is **make-offer**, the *negotiate()* method is called to come up with an appropriate response. As in our **BuyerAgent**, this method is overridden by subclasses to introduce increasing levels of sophistication.

```
public void processMessage(CIAgentMessage msg) {

   if (msg.performative.equals("ask" )) {
     String item = msg.content ;
     // see if we have any items left
     if (itemInInventory(item)) {
       // start a new negotiation
       String id = genId() ;
       current = removeItemFromInventory(item) ;
       current.offer  = new Offer(msg.sender, item, id, 0);
       current.lastOffer = current.strikePrice + 100 ;
       negotiations.put(id, current) ;
       CIAgentMessage answer =
          new CIAgentMessage("make-offer", item + " " + id  + " " +
          current.lastOffer , msg.replyWith , msg.sender , item,
          name) ;
       CIAgentEvent e = new CIAgentEvent(this, answer) ;
       notifyCIAgentEventListeners(e); // respond through Facil.
     } else {
        // deny we have any to sell
       CIAgentMessage answer =
             new CIAgentMessage("deny", item , msg.replyWith ,
                                msg.sender , item , name) ;
       CIAgentEvent e = new CIAgentEvent(this, answer) ;
       trace(name + ": deny we have any " + item +
             " to sell to " + msg.sender + "\n");
       notifyCIAgentEventListeners(e); // respond through Facil
     }
```

```
      return ;
    }

    Offer offer = new Offer(msg) ;  //
    trace(name + ": Offer from " + offer.sender + ": content is " +
          offer.item + " " + offer.id + " " + offer.price + "\n") ;

    current = (BasicNegotiation)negotiations.get(offer.id) ;
    if (current == null) {
       // we must have sold this item --- reject offer
       CIAgentMessage answer =
             new CIAgentMessage("reject-offer", msg.content ,
                   msg.replyWith , msg.sender , offer.item , name) ;
       CIAgentEvent e = new CIAgentEvent(this, answer) ;
       notifyCIAgentEventListeners(e);   // respond through Facil
       return;
    }
    current.newOffer(offer) ;

    // buyer has made an offer, we can accept,
    // make a counter offer, or reject
    if (msg.performative.equals("make-offer" )) {
         negotiate(offer, msg) ;
    }

    // buyer has received and acknowledged our accept-offer
    // remove item from active list, add $ to income
    // unadvertise this item with the Facilitator
    if (msg.performative.equals("tell" )) {
        negotiations.remove(offer.id) ;
        income += offer.price ;
        CIAgentMessage msg2 =
               new CIAgentMessage("unadvertise", current.offer.item ,
                           null , null , current.offer.item , name) ;
        CIAgentEvent e = new CIAgentEvent(this, msg2) ;
        notifyCIAgentEventListeners(e);  // signal listeners
      }

 }
```

In the basic **SellerAgent**, if the Buyer's offer price is above the desired minimum selling price, the offer is immediately accepted and a message is sent back to the **BuyerAgent**. The **BuyerAgent** must acknowledge the offer with a tell message before the item can be removed from inventory. If the **BuyerAgent** is offering less than the **SellerAgent** is willing to take for the item, the **Offer** is rejected and the negotiations are closed by placing the item back in inventory (remember, this is the basic agent).

```
void negotiate(Offer offer, CIAgentMessage msg) {
   if (offer.price > current.strikePrice) {
          // accept
          CIAgentMessage answer =
             new CIAgentMessage(“accept-offer”, msg.content,
                  msg.replyWith , msg.sender , offer.item , name) ;
          CIAgentEvent e = new CIAgentEvent(this, answer) ;
          notifyCIAgentEventListeners(e);
          return ;
      }
      if (offer.price < current.strikePrice) {
          // reject
          rejectOffer(offer) ;
          return;
      }
    CIAgentMessage answer =
            new CIAgentMessage(“make-offer”, offer.item + “ ” +
                             offer.id + “ ” + current.lastOffer ,
                             msg.replyWith , msg.sender ,
                             offer.item , name) ;
    CIAgentEvent e = new CIAgentEvent(this, answer) ;
    notifyCIAgentEventListeners(e);
}

// break off a negotiation (for whatever reason)
// return item to inventory, remove negotiations from active list
void rejectOffer(Offer offer) {
    CIAgentMessage answer =
            new CIAgentMessage(“reject-offer”, msg.content ,
                 msg.replyWith , msg.sender , offer.item , name) ;
    CIAgentEvent e = new CIAgentEvent(this, answer) ;
    notifyCIAgentEventListeners(e);
    negotiations.remove(offer.id) ; // no further negotiations
    current.offer.id = null ;       // offer id has expired
    inventory.addElement(current); // place item back in inventory
}
```

The *itemInInventory()* method is called by *processMessage()* in response to an “ask” from a Buyer to see if any items of that type are available to sell. An **Enumeration** is used to walk through the **Vector** to examine each item in *inventory*. A boolean true is returned when a match is found. The *removeItemFromInventory()* method has a similar structure to *itemInInventory()*. In this case, though, the found item is removed from the *inventory* **Vector**, and the **BasicNegotiation** object is returned to the caller. The **BasicNegotiation** is used as a convenient holder for *inventory* items.

```
// returns true if the item is in stock
```

```
boolean itemInInventory(String item) {
  Enumeration enum = inventory.elements() ;
  boolean haveItem = false ;
  while(enum.hasMoreElements()) {
    BasicNegotiation stockItem =
                (BasicNegotiation)enum.nextElement() ;
    if (stockItem.getItem().equals(item)) {
      haveItem = true ;
      break ;
    } else {
      continue ;
    }
  }
  return haveItem ;
}

// returns the specified item  from inventory
// if we have no items, return null
BasicNegotiation removeItemFromInventory(String item) {
  Enumeration enum = inventory.elements() ;
  BasicNegotiation stockItem = null ;
  while(enum.hasMoreElements()) {
    stockItem = (BasicNegotiation)enum.nextElement() ;
    if (stockItem.getItem().equals(item)) {
      inventory.removeElement(stockItem) ;
      break ;
    } else {
      stockItem = null ; // this isn't it
      continue ;
    }
  }
  return stockItem ;
}
```

Previous | Table of Contents | Next

Enhanced Buyers and Sellers

In this section, we describe improvements to our basic Buyer and SellerAgents. These enhanced agents extend the **BuyerAgent** and **SellerAgent** classes, primarily by overriding the *negotiate()* methods, and by utilizing additional information about the negotiation process. The **BetterBuyerAgent** and **BetterSellerAgent** use hardcoded logic with a slightly more aggressive negotiation strategy and warrant no further discussion. The **BestBuyerAgent** and **BestSellerAgent** make use of the **RuleBase** class and if-then rules to determine their prices and actions during the negotiations.

We first focus on the **BestBuyerAgent** class. As a subclass of **BuyerAgent**, it inherits all of the basic sales transaction behavior. **BestBuyerAgents** have the same *wishList* items and prices as the **BuyerAgents**. However, they make use of additional information in the **BasicNegotiation** object and also use a set of rules to determine what action to take in the negotiation process.

Several additional data members are added to the **BestBuyerAgent** class. These include *rb*, the **RuleBase** instance that is initialized when the overridden *initialize()* method is invoked from the **BuyerAgent** *process()* startup method, and several **RuleVariables** used in the **RuleBase**. The *initializeBestBuyerRuleBase()* method defines the *offerDelta*, an intermediate output variable, and *spread* and *firstOffer*, which are antecedent variables. The *spread* variable is the difference between the current offered price and the asking or strikePrice. The *firstOffer* variable is a boolean used to signal when a negotiation is just starting.

In addition to rules that compute the value of *offerDelta*, there are a couple of action rules. These rules have **EffectorClauses** as their consequent. When they fire, they call the *effector()* method defined in **BestBuyerAgent**. As described earlier, the Buyer agents can only make offers to the SellerAgents. If the forward-chaining inferencing decides to make a counter offer or accept, the *effector()* method is called by the action rule, and the message is sent directly from the rule firing. An alternative design, if the **RuleBase** did not support effector or action rules, would be to compute the *offerDelta* and have hardcoded logic examine the value and determine what action to perform.

The negotiating strategy is more complex than in the **BuyerAgent** and **BetterBuyerAgent**, but it is not a masterpiece of haggling by any stretch of the imagination. If the Seller's offer is within $25 of the *strikePrice*, the offer is accepted. Otherwise, depending on the size of the spread, increasing amounts are deducted from the Seller's offer and a counter offer is made. The intent was to show how domain knowledge in the form of rules could be integrated into an intelligent agent marketplace. In a commercial application, a more sophisticated rule base would be used.

```
public class BestBuyerAgent extends BuyerAgent implements Effector {

 RuleBase rb = new RuleBase("BestBuyer");
 RuleVariable offerDelta;
 RuleVariable spread ;
 RuleVariable firstOffer ;
 Offer offer ;   // the current offer

 public void initialize() {
   initBestBuyerRuleBase() ;
   rb.setDisplay(textArea) ;  // give rulebase a display area
 }

 public BestBuyerAgent() { name = "BestBuyer"; }

 public BestBuyerAgent(String Name) { super(Name); }

 // used by action rules in rule base to make or accept offers
 public long effector(Object obj, String eName, String args) {

   if (eName.equals("make-offer")) {
       // rule base decided to counter-offer
       long delta = (new Long(offerDelta.getValue())).longValue() ;
       current.lastOffer = offer.price - delta ;  //
       CIAgentMessage answer =
            new CIAgentMessage("make-offer", offer.item + " " +
                               offer.id + " " + current.lastOffer,
                               offer.item , msg.sender , offer.item,
                               name) ;
       CIAgentEvent e = new CIAgentEvent(this, answer) ;
       notifyCIAgentEventListeners(e);
       return 0 ;
   }

   if (eName.equals("accept-offer")) {
       // rule base decided to accept the offer
       // just resend it to the seller
       CIAgentMessage answer =
```

```
            new CIAgentMessage("make-offer", msg.content ,
                 msg.replyWith , msg.sender , offer.item , name) ;
       CIAgentEvent e = new CIAgentEvent(this, answer) ;
       notifyCIAgentEventListeners(e);
       return 0 ;
    }
    return 1 ; // unknown effector method
 }

 // this method must be overridden to provide alternate
 // negotiation strategies
 void negotiate(Offer offer, CIAgentMessage msg) {

    // figure out if 1st time or second time or other
    // compute spread
    // let rule base figure out our response
    // if deltaoffer is 0 then we should accept

    rb.reset() ;  // allow all rules to fire, clear vars
    this.offer = offer;  // make visible to rule base effectors

    if (offer.price >= current.strikePrice) {

       long delta = offer.price - current.strikePrice;
       if (delta < 25) {
         spread.setValue("<25;") ;
       } else if (delta < 50) {
         spread.setValue("25-50") ;
       } else if (delta >= 50) {
         spread.setValue(">50;") ;
       }
    } else {
       spread.setValue("0") ; // meets our price, accept

    }
    if (current.prevOffer.id.equals("")) {
        firstOffer.setValue("yes") ;
      } else {
        firstOffer.setValue("no") ;
    }

    offerDelta.setValue(null) ;
    rb.forwardChain() ;  // inference
    if (offerDelta.getValue() == null) {
       trace(name + " rule base couldn't decide what to do.\n") ;
    }
 }

// initialize the BestBuyer rule base
public void initBestBuyerRuleBase() {
```

```
 rb.goalClauseStack = new Stack() ; // goals and subgoals

 rb.variableList = new Hashtable() ;
 offerDelta = new RuleVariable("offerDelta") ;
 offerDelta.setLabels("0 25 50 100") ;
 rb.variableList.put(offerDelta.name, offerDelta) ;

 firstOffer = new RuleVariable("firstOffer") ;
 firstOffer.setLabels("yes no") ;
 rb.variableList.put(firstOffer.name, firstOffer) ;

 spread = new RuleVariable("spread") ;
 spread.setLabels("<25; 25-50 >50;") ;
 rb.variableList.put(spread.name, spread) ;

 // Note: at this point all variables values are NULL

Condition cEquals = new Condition("=") ;
Condition cNotEquals = new Condition("!=") ;

// define rules
rb.ruleList = new Vector() ;
Rule first = new Rule(rb, "first",
     new Clause(firstOffer,cEquals, "yes") ,
     new Clause(offerDelta,cEquals, "75")) ;  // counter

Rule second1 = new Rule(rb, "second1",
     new Clause(firstOffer, cEquals, "no"),
     new Clause(spread, cEquals, ">50;"),
     new Clause(offerDelta,cEquals, "50")) ; // counter

Rule second2 = new Rule(rb, "second2",
     new Clause(firstOffer, cEquals, "no"),
     new Clause(spread, cEquals, "25-50"),
     new Clause(offerDelta,cEquals, "25")) ;  // counter

Rule second3 = new Rule(rb, "second3",
     new Clause(firstOffer, cEquals, "no"),
     new Clause(spread, cEquals, "<25;"),
     new Clause(offerDelta,cEquals, "0")) ;  // accept

// action rule
Rule accept = new Rule(rb, "accept",
     new Clause(offerDelta, cEquals, "0"),
     new EffectorClause("accept-offer", "0")) ;

// action rule
Rule counter = new Rule(rb, "counter",
     new Clause(offerDelta, cNotEquals, "0"),
     new EffectorClause("make-offer", null)) ; // use offerDelta
```

```
// define this object as effector implementor
rb.addEffector(this, “make-offer”);
rb.addEffector(this, “accept-offer”);
}

};
```

Previous | Table of Contents | Next

Next, we focus on the **BestSellerAgent** class. Much of its structure mirrors the changes to **BestBuyerAgent**. As a subclass of **SellerAgent**, it inherits all of the basic sales transaction behavior. **BestSellerAgents** have the same inventory items and prices as the **SellerAgents**.

Additional data members are added to the **BestSellerAgent** class. These include *rb*, a **RuleBase** instance, and several **RuleVariables** used in the **RuleBase**. The *initializeBestSellerRuleBase()* method defines the *offerDelta*, an intermediate output variable, and *spread* and *firstOffer*, which are antecedent variables. *Spread* is the difference between the current offered price and the asking or strikePrice. *FirstOffer* is a boolean used to signal when a negotiation is just starting.

Like the **BestBuyerAgent**, the **BestSellerAgent** uses action rules to directly send responses. As described earlier, the SellerAgents control the negotiations, and can decide to accept an offer, make a counter offer, or reject the offer and break off negotiations with the **BuyerAgents**. The **RuleBase** in the **BestSellerAgent** is more complex than the **BestBuyerAgent** to handle these additional cases and associated actions.

The *negotiate()* method first resets the rule base and then computes and sets the value of the *spread*. A special test is added to see if the previous offer and the current offer are identical. If so, that means the buyer has accepted the offer, and the value of spread is set to >50; to force an accept-offer action. Obviously, accept could have been called directly, but in this case, we want to demonstrate the **RuleBase** action mechanism.

In testing with the rule-based **BestBuyerAgent** and **BestSellerAgent**, we found that the agents could get into infinite loops bargaining back and forth. The iteration member was added to the **BasicNegotiation** class to keep track of this and short-circuit these loops when they occur. However, in real multiagent systems there is always the possibility of producing message storms or deadlocks between agents. Failsafe mechanisms to detect and avoid these conditions must be included in commercial implementations.

A forward-chaining inference cycle is performed. The amount of the counter offer (if any) is computed, and the action or *effector()* methods are called during

the inferencing process. The *effector()* method contains logic for interpreting the value of the *offerDelta* **RuleVariable**, depending on which effector was invoked.

```
public class BestSellerAgent extends SellerAgent implements Effector

  RuleBase rb = new RuleBase(“BestSeller”);
  RuleVariable offerDelta;
  RuleVariable spread ;
  RuleVariable firstOffer ;
  Offer offer ;   // the current offer

  public void initialize() {
    initBestSellerRuleBase() ;
    rb.setDisplay(textArea) ;  // give rulebase a display area
  }

  public BestSellerAgent() { name = “BestSeller”; }

  public BestSellerAgent(String Name) { super(Name); } ;

  // used by action rules in rule base to make or accept offers
  public long effector(Object obj, String eName, String args) {

    if (eName.equals(“make-offer”)) {
        // rule base decided to counter-offer
        long delta = (new Long(offerDelta.getValue())).longValue() ;
        if (delta == -100) {
           current.lastOffer = current.strikePrice ;
        } else {
           current.lastOffer = offer.price + delta ;  //
        }
        CIAgentMessage answer =
                new CIAgentMessage(“make-offer”, offer.item + “ ” +
                                   offer.id + “ ” + current.lastOffer,
                                   offer.item , msg.sender , offer.item ,
                                   name) ;
        CIAgentEvent e = new CIAgentEvent(this, answer) ;
        notifyCIAgentEventListeners(e);
        return 0 ;

    }
   if (eName.equals(“reject-offer”)) {
        // rule base decided to reject the offer
         rejectOffer(offer) ;
         return 0 ;
    }

    if (eName.equals(“accept-offer”)) {
        // rule base decided to accept the offer
```

```
        CIAgentMessage answer =
                  new CIAgentMessage("accept-offer", msg.content ,
                    msg.replyWith , msg.sender , offer.item , name)
        CIAgentEvent e = new CIAgentEvent(this, answer) ;
        notifyCIAgentEventListeners(e);
        return 0 ;
    }
    return 1 ; // unknown effector method
  }

void negotiate(Offer offer, CIAgentMessage msg) {

    // figure out if 1st time or second time or other
    // compute spread
    // let rule base figure out our response
    // if deltaoffer is 0 then we should accept
     rb.reset() ;  // allow all rules to fire, clear vars
     this.offer = offer;  // make visible to rule base effectors

    if (offer.price < current.strikePrice) {
        spread.setValue("<0;") ; // below asking price
    } else {                     // above asking price
       long delta = offer.price - current.strikePrice ;
       if (delta < 25) {
         spread.setValue("0-25") ;
       } else if (delta < 50) {
         spread.setValue("25-50") ;
       } else if (delta >= 50) {
         spread.setValue(">50;") ;
       }
    }

    // if buyer echos the lastOffer, then he is agreeing
    if (offer.price == current.lastOffer) {
        spread.setValue(">50;") ; // accept via rule action
    }

     if (current.iteration == 0) {
         firstOffer.setValue("yes") ;
     } else {
         firstOffer.setValue("no") ;
     }
     current.iteration++ ; // increment iteration count

     if (current.iteration > 10) {  // could be randomized
         rejectOffer(offer) ;
         return; // break off negotiations
      }

     rb.forwardChain() ;  // inference
```

```
    if (offerDelta.getValue() == null) {
       trace(name + " rule base couldn't decide what to do -- so re
       rejectOffer(offer) ;
     }

 }

// initialize the BestSeler rule base
public void initBestSellerRuleBase() {
 rb.goalClauseStack = new Stack() ; // goals and subgoals

 rb.variableList = new Hashtable() ;

 offerDelta = new RuleVariable("offerDelta") ;
 offerDelta.setLabels("-100 0 30 60 100") ;
 rb.variableList.put(offerDelta.name, offerDelta) ;

 firstOffer = new RuleVariable("firstOffer") ;
 firstOffer.setLabels("yes no") ;
 rb.variableList.put(firstOffer.name, firstOffer) ;

 spread = new RuleVariable("spread") ;
 spread.setLabels("<0; 0-25 25-50 >50;") ;
 rb.variableList.put(spread.name, spread) ;

 // Note: at this point all variables values are NULL
Condition cEquals = new Condition("=") ;
Condition cNotEquals = new Condition("!=") ;

// define rules
rb.ruleList = new Vector() ;
Rule first = new Rule(rb, "first",
     new Clause(firstOffer,cEquals, "yes") ,
     new Clause(offerDelta,cEquals, "50")) ;  // counter

Rule second1 = new Rule(rb, "second1",
     new Clause(firstOffer,cEquals, "no")  ,
     new Clause(spread, cEquals, "25-50"),
     new Clause(offerDelta,cEquals, "40")) ;  // counter

Rule second2 = new Rule(rb, "second2",
     new Clause(firstOffer,cEquals, "no")  ,
     new Clause(spread, cEquals, "0-25"),
     new Clause(offerDelta,cEquals, "0")) ;  // accept

Rule second3 = new Rule(rb, "second3",
     new Clause(firstOffer,cEquals, "no")  ,
     new Clause(spread, cEquals, ">50;"),
     new Clause(offerDelta,cEquals, "0")) ; // accept
```

```
 Rule second4 = new Rule(rb, “second4”,
      new Clause(firstOffer,cEquals, “no”)  ,
      new Clause(spread, cEquals, “<0;”),
      new Clause(offerDelta,cEquals, “-100”)) ;  // reject

// action rule
 Rule accept = new Rule(rb, “accept”,
      new Clause(offerDelta, cEquals, “0”),
      new EffectorClause(“accept-offer”, “0”)) ;

// action rule
 Rule reject = new Rule(rb, “reject”,
      new Clause(firstOffer, cEquals, “no”),
      new Clause(offerDelta, cEquals, “-100”),
      new EffectorClause(“reject-offer”, “0”)) ;

 // action rule
 Rule counter = new Rule(rb, “counter”,
      new Clause(offerDelta, cNotEquals, “0”),
      new EffectorClause(“make-offer”, null)) ; // use offerDelta

 // define this object as effector implementor
 rb.addEffector(this, “make-offer”);
 rb.addEffector(this, “reject-offer”);
 rb.addEffector(this, “accept-offer”);
 }

};
```

Previous | Table of Contents | Next

MarketPlace Application

The **CIAgent** MarketPlace application GUI was developed using the Symantec Visual Café development tool, version 1.0d. This code generates Java 1.0.2-level code and uses the 1.0.2 event model. However, this code was tested and run under the Java 1.1.1 virtual machine. As mentioned in the earlier applications, this code is not central to the intelligent agent application developed in this chapter.

We point out some of the unique application logic required to implement the **CIAgent** MarketPlace. The seven **CIAgents**, the **FacilitatorAgent**, and the three types of Buyer and SellerAgents are data members of the MarketPlace application class.

```
public class Marketplace extends Frame {

   // used by agents to display messages in market place window
   synchronized void trace(String msg) { textArea2.appendText(msg)

    void About_Action(Event event) {
         //{{CONNECTION
         // Action from About Create and show as modal
         (new AboutMarketplaceDialog(this, true)).show();
         //}}
    }

    void Exit_Action(Event event) {
         //{{CONNECTION
         // Action from Exit Create and show as modal
         (new QuitDialog(this, true)).show();
         //}}
    }

   void Clear_Action(Event event) {
       textArea1.setText("") ;
       textArea2.setText("") ;
   }

    void Start_Action(Event event) {

        textArea1.setText("") ;
        textArea2.setText("") ;
```

```
    int traceLevel=0 ;  // default is summary
    if (viewDetails.getState() == true) {
        traceLevel = 1 ;
    }

   Object display = this ;    // or textArea2 ???
   facilitator = FacilitatorAgent.Instance() ; // singleton
   facilitator.reset() ; // clear it out
   facilitator.textArea = textArea1 ; // give view port for msg
   facilitator.traceLevel = traceLevel ;
   facilitator.process() ;

    // need to check Buyer and Seller menu items
    // to see if we need to instantiate any agents
    if (basicSeller.getState()==true) {
     basicSellerAgent = new SellerAgent() ;
     basicSellerAgent.setDisplay(display,traceLevel);
     basicSellerAgent.process() ;
    }
    if (intermedSeller.getState()==true) {
     intermedSellerAgent = new BetterSellerAgent() ;
     intermedSellerAgent.setDisplay(display,traceLevel);
     intermedSellerAgent.process() ;
    }
    if (advancedSeller.getState()==true) {
     advancedSellerAgent = new BestSellerAgent() ;
     advancedSellerAgent.setDisplay(display,traceLevel);
     advancedSellerAgent.process() ;
    }
    if (basicBuyer.getState()==true) {
     basicBuyerAgent = new BuyerAgent() ;
     basicBuyerAgent.setDisplay(display, traceLevel);
     basicBuyerAgent.process() ;
    }
    if (intermedBuyer.getState()==true) {
     intermedBuyerAgent = new BetterBuyerAgent() ;
     intermedBuyerAgent.setDisplay(display, traceLevel);
     intermedBuyerAgent.process() ;
    }
    if (advancedBuyer.getState()==true) {
      advancedBuyerAgent = new BestBuyerAgent() ;
     advancedBuyerAgent.setDisplay(display, traceLevel);
     advancedBuyerAgent.process() ;
    }
   textArea1.appendText("Starting Marketplace \n") ;
}

void Stop_Action() {
  if (facilitator != null) facilitator.stop() ;
  if (basicSellerAgent != null) basicSellerAgent.stop() ;
```

```
  if (intermedSellerAgent != null) intermedSellerAgent.stop() ;
  if (advancedSellerAgent != null) advancedSellerAgent.stop() ;
  if (basicBuyerAgent != null) basicBuyerAgent.stop() ;
  if (intermedBuyerAgent != null) intermedBuyerAgent.stop() ;
  if (advancedBuyerAgent != null) advancedBuyerAgent.stop() ;
  textArea1.appendText("Ending Marketplace \n") ;
}
public Marketplace() {

    //{{INIT_CONTROLS
    setLayout(null);
    addNotify();
    resize(insets().left + insets().right + 537,insets().top +
           insets().bottom + 449);
    setBackground(new Color(16777215));
    openFileDialog1 =
       new java.awt.FileDialog(this, "Open",FileDialog.LOAD);
    //$$ openFileDialog1.move(48,384);
    textArea1 = new java.awt.TextArea();
    textArea1.reshape(insets().left + 12,insets().top +
                      36,516,120);
    textArea1.setBackground(new Color(16777215));
    add(textArea1);
    textArea2 = new java.awt.TextArea();
    textArea2.reshape(insets().left + 12,insets().top +
                      180,516,216);
    textArea2.setBackground(new Color(12632256));
    add(textArea2);
    label1 = new java.awt.Label("Facilitator");
    label1.reshape(insets().left + 24,insets().top +
                   12,221,21);
    add(label1);
    label2 = new java.awt.Label("Marketplace");
    label2.reshape(insets().left + 24,insets().top +
                    156,224,26);
    add(label2);
    setTitle("CIAgent Marketplace Application");
    setResizable(false);
    //}}

    //{{INIT_MENUS
    mainMenuBar = new java.awt.MenuBar();
    menu1 = new java.awt.Menu("File");
    menu1.add("Clear");
    menu1.addSeparator();
    menu1.add("Exit");
    mainMenuBar.add(menu1);

    menu5 = new java.awt.Menu("Actions");
```

```
        menu5.add(“Start”);
        menu5.add(“Stop”);
        mainMenuBar.add(menu5);

        menu4 = new java.awt.Menu(“View”);
        viewDetails = new java.awt.CheckboxMenuItem(“Details”);
        viewDetails.setState(false);
        menu4.add(viewDetails);
        viewSummary = new java.awt.CheckboxMenuItem(“Summary”);
        viewSummary.setState(true);
        menu4.add(viewSummary);
        mainMenuBar.add(menu4);

        menu6 = new java.awt.Menu(“Buyers”);
        basicBuyer = new java.awt.CheckboxMenuItem(“Basic”);
        basicBuyer.setState(true);
        menu6.add(basicBuyer);
        intermedBuyer =
            new java.awt.CheckboxMenuItem(“Intermediate”);
        intermedBuyer.setState(false);
        menu6.add(intermedBuyer);
        advancedBuyer = new java.awt.CheckboxMenuItem(“Advanced”);
        advancedBuyer.setState(false);
        menu6.add(advancedBuyer);
        mainMenuBar.add(menu6);

        menu7 = new java.awt.Menu(“Seller”);
        basicSeller = new java.awt.CheckboxMenuItem(“Basic”);
        basicSeller.setState(true);
        menu7.add(basicSeller);
        intermedSeller =
            new java.awt.CheckboxMenuItem(“Intermediate”);
        intermedSeller.setState(false);
        menu7.add(intermedSeller);
        advancedSeller =
            new java.awt.CheckboxMenuItem(“Advanced”);
        advancedSeller.setState(false);
        menu7.add(advancedSeller);
        mainMenuBar.add(menu7);

        menu3 = new java.awt.Menu(“Help”);
        mainMenuBar.setHelpMenu(menu3);
        menu3.add(“About”);
        mainMenuBar.add(menu3);
        setMenuBar(mainMenuBar);
        //$$ mainMenuBar.move(12,384);
        //}}
}
```

```
 public Marketplace(String title) {
     this();
     setTitle(title);
 }

public synchronized void show() {
    move(50, 50);
    super.show();
}

 public boolean handleEvent(Event event) {
 if (event.id == Event.WINDOW_DESTROY) {
       hide();         // hide the Frame
       dispose();      // free the system resources
       System.exit(0); // close the application
       return true;
 }
      return super.handleEvent(event);
 }
 public boolean action(Event event, Object arg) {

      if (event.target instanceof MenuItem) {
           String label = ((MenuItem)event.target).getLabel();
           if (label.equalsIgnoreCase("About")) {
                About_Action(event);
                return true;
           } else
                if (label.equalsIgnoreCase("Exit")) {
                     Exit_Action(event);
                     return true;
      } else
                if (label.equalsIgnoreCase("Clear")) {
                     Clear_Action(event);
                     return true;
           } else
                if (label.equalsIgnoreCase("Details")) {
                     if (viewDetails.getState()== true) {
                         viewSummary.setState(false) ;
                     } else {
                         viewSummary.setState(true) ;
                     }
                     return true;
           } else
                if (label.equalsIgnoreCase("Summary")){
                     if (viewSummary.getState()== true) {
                         viewDetails.setState(false) ;
                     } else {
                         viewDetails.setState(true) ;
                     }
                     return true;
```

```
            } else
                if (label.equalsIgnoreCase("Start")) {
                      Start_Action(event);
                      return true;
                } else
                    if (label.equalsIgnoreCase("Stop")) {
                      Stop_Action();
                      return true;
            }

        }
        return super.action(event, arg);
    }

    static public void main(String args[]) {
        (new Marketplace()).show();
    }

    //{{DECLARE_CONTROLS
    java.awt.FileDialog openFileDialog1;
    java.awt.TextArea textArea1;
    java.awt.TextArea textArea2;
    java.awt.Label label1;
    java.awt.Label label2;
    //}}

    //{{DECLARE_MENUS
    java.awt.MenuBar mainMenuBar;
    java.awt.Menu menu1;
    java.awt.Menu menu5;
    java.awt.Menu menu4;
    java.awt.CheckboxMenuItem viewDetails;
    java.awt.CheckboxMenuItem viewSummary;
    java.awt.Menu menu6;
    java.awt.CheckboxMenuItem basicBuyer;
    java.awt.CheckboxMenuItem intermedBuyer;
    java.awt.CheckboxMenuItem advancedBuyer;
    java.awt.Menu menu7;
    java.awt.CheckboxMenuItem basicSeller;
    java.awt.CheckboxMenuItem intermedSeller;
    java.awt.CheckboxMenuItem advancedSeller;
    java.awt.Menu menu3;
    //}}

FacilitatorAgent facilitator ;
BuyerAgent basicBuyerAgent ;
BetterBuyerAgent intermedBuyerAgent ;
BestBuyerAgent advancedBuyerAgent ;
SellerAgent basicSellerAgent ;
```

```
    BetterSellerAgent intermedSellerAgent ;
    BestSellerAgent advancedSellerAgent ;
}
```

Previous | Table of Contents | Next

Discussion

In this section, we describe the reasoning behind some of the design decisions made while developing the MarketPlace application.

Perhaps the first decision we made was the choice of using the KQML-like **CIAgentMessage** instead of parsing “real” KQML messages. Our feeling was that we could explore many of the issues of the agent communications using the **CIAgentMessage** class without getting bogged down in the intricacies of parsing KQML statements. We dealt with the issues related to the use of KQML performatives and the maintenance of the conversation states between the Facilitator and the Buyer and Seller agents.

Our next decision was to design our own matchmaker or Facilitator. There are several KQML facilitators in the literature and some in fielded applications and products. Our relatively simple Facilitator provided all of the function we needed for this application without getting too complex. Having the Facilitator stay “in the loop” between Buyer and Seller was another design decision which could easily have gone the other way. We felt the trace logs provided by a centralized Facilitator would be useful. However, in a system with many agents, the Facilitator as middleman would be a performance bottleneck. As a matchmaker who introduces the agents and then gets out of the way, the Facilitator would provide most of the function while not being a performance liability.

Another fundamental design choice was using the **BuyerAgent** and **SellerAgent** threads to kick off transactions, but not to handle event processing. In a commercial multiagent application, it would be better to have the message signaling happen synchronously but the message processing happen asynchronously. This is a more complicated design because you have to worry about signaling end-of-event-processing events. But it would allow higher throughput with a large number of agents and also would improve the robustness of the overall system, because a single pokey agent wouldn’t gum up the works.

Also, as may already have been apparent, we did not spend a lot of time developing optimal negotiation strategies for the **BuyerAgent** and **SellerAgents**. As we stated, our goal was to show the progress of agents from simple hardcoded to more complex rule-based, and to show the benefits of the increased

sophistication in the bottom-line performance of the algorithms. One could easily imagine hardcoding a better strategy than we used in the **BestBuyerAgent** and **BestSellerAgent**. But the point was to illustrate the power and flexibility of using a rule-based engine to control the negotiations and to directly take the actions using the *effector()* methods.

A simplification used in the negotiation strategy was that the **SellerAgent** removed the item from inventory as soon as a Buyer asked about it. This prevented two negotiations going on at the same time for the same item. While in a real marketplace this sort of thing goes on all of the time, the added complexity of this approach seems to outweigh the benefits.

Summary

In this chapter, we developed an electronic marketplace application using seven CIAgent-based intelligent agents. The main points include:

- The MarketPlace application consisted of a single *FacilitatorAgent*, one or more *BuyerAgents*, and one or more *SellerAgents*. There were three types of Buyer and SellerAgents using increasingly better negotiating strategies.
- All communications between the Buyers and Sellers went through the Facilitator using KQML-like objects called *CIAgentMessages*. These message include KQML-performatives such as *register*, *advertise*, *unadvertise*, *recommend-one*, *ask*, and *tell*. Application-specific performatives included *make-offer*, *accept-offer*, and *reject-offer*.
- The sales negotiation protocol was defined, giving the *SellerAgent* more control over the process. The *BasicNegotiation* class was introduced to encapsulate the details of each transaction. An *Offer* consists of the agent name, the item, a unique item id generated by the SellerAgent at the start of a negotiation, and the offer price.
- The *BestBuyerAgent* and *BestSellerAgent* used the *RuleBase*, *Rule*, and *Clause* classes from Chapter 4, along with enhancements such as *EffectorClauses* to represent the negotiation strategy in rule form, and directly send the reply messages using action rules.

Exercises

1. Write a simple KQML parser for the subset of the language supported by the **CIAgentMessage** class.

2. Modify or extend the **CIAgent FacilitatorAgent** behavior so that the Buyers and Sellers directly communicate with each other after a match is made. What are the benefits and drawbacks of this approach?
3. What domain knowledge would be needed to have a **BuyerAgent** you would trust with your money? What reasoning skills would be required? How difficult would it be to build this **BuyerAgent**?

Previous | Table of Contents | Next

Chapter 11
Java-Based Agent Environments

We conclude with an examination of other Java-based agent environments. While this is not an exhaustive list of Java-based agents, it provides an overview of what is happening in the industry, where the focus is, and what more needs to be done to bring Java intelligent agents to their full potential.

Aglets

Aglets are Java-based autonomous agents developed by IBM. They provide the basic capabilities required for mobility. Each aglet has a globally unique name. A travel itinerary is used to specify the destinations to which the agent must travel and what actions it must take at each location. In order for an aglet to run on a particular system, that system must be running an aglet host application. This provides a platform-neutral execution environment for the aglet. The aglet workbench includes a configurable Java security manager, which can restrict the activity of an aglet on the system in the same way the default security manager is used to restrict the activities of an applet.

Aglets can communicate using a whiteboard that allows agents to collaborate and share information asynchronously. Synchronous and asynchronous message passing is also supported for aglet communication. Aglets are streamed using standard Java serialization or externalization. A network agent class loader is supplied which allows an aglet's bytecode stream and state to travel across a network.

While aglets are not inherently intelligent, they are written in Java and can be extended with any of the intelligent Java code we have provided in the earlier chapters of this book. For more information on Aglets, see http://www.trl.ibm.co.jp/aglets/.

FTP Software Agent Technology

FTP Software Agent Technology is Java-based software designed to manage heterogeneous networks across the Internet using agent technology. The agents are autonomous and mobile, and can move to any system in the network which has an Agent Responder installed. As the agent moves from system to system, its tasks may change, depending on the environment of the system it is visiting.

The agents can interact with other agents or with the user, as needed. But FTP agents do not require any user interaction—based on "push" technology, they can move from system to system, respond to events, and perform tasks according to criteria predefined by the user. An Agent Manager is responsible for launching the agent.

Because of the business-oriented, network-centric focus of the FTP software agents, there is a strong emphasis on security. In addition to the standard Java security model, the software provides authentication by user password, RC2 (exportable RSA algorithms), and DES (Digital Encryption Standard) security.

Two applications are available using this agent technology: IP Auditor and IP Distributor. The IP Auditor application uses agents to collect hardware, software, and configuration information for systems in a network. The IP Distributor application will distribute software and optionally execute functions on target systems within a network.

Literature on the FTP Software Agent Technology describes the agents as "intelligent," but intelligence was defined in this context as having the ability to transport data and instructions, analyze the target environment, and respond to the conditions found at the target system. It does not seem to refer to rule-based inferencing or machine learning as described earlier in this book. Details on FTP software agents can be found at http:/www.ftp.com/.

Voyager

Voyager, from ObjectSpace, Inc., is an agent-enhanced Object Request Broker (ORB) written entirely in Java. An ORB provides the capability to create objects on a remote system and invoke methods on those objects. Voyager augments the traditional ORB with agent capabilities.

Voyager agents have mobility and autonomy which is provided in the base class, Agent. An Agent can move itself from one location to another and can leave

behind a forwarding address with a “secretary” so that future messages can be forwarded to its new location. Specialized agents, called Messengers, are used to deliver messages. Messages can be synchronous, one-way (similar to asynchronous), or future, which are asynchronous but return a placeholder that can be used to retrieve a return value at a later time.

Like aglets, an agent’s itinerary instructs the agent as to what operations it needs to perform at each location. Also, like aglets, Voyager uses serialization to stream the agent’s state as the agent moves from location to location. Voyager also includes a security manager which can be used to restrict the operations an agent can perform. In the future, Voyager will be enhanced with a rule-based security model which supplements the standard Java security mechanism.

Voyager agents themselves are not inherently intelligent. They do not contain an inference engine, neural networks, or any other artificial intelligence technology. But, like aglets, they can be augmented with any of the artificial intelligence techniques described in this book. Additional information on Voyager can be found at http://www.objectspace.com/.

Odyssey

General Magic’s Odyssey agent system is a set of Java classes that support Odyssey agents, agent systems, and places. Agents are created by subclassing either the Agent class or the Worker class. Each Agent executes in its own Java thread, and can travel from place to place. At each new place, however, the thread must be restarted. An instance of the Place class defines the execution environment for an agent on each host system. A Worker is a specialization of the Agent class. A Worker contains an itinerary which is a set of tasks and destinations. The Worker executes one task at each destination host. The Odyssey agent system also provides support classes such as Ticket, which defines how and where an agent may travel, and Petition, which identifies other Agents with whom the Agent wishes to communicate.

Odyssey is written entirely in Java and requires JDK 1.1. It uses RMI as its transport, but can also be configured to use Microsoft’s Distributed Component Model (DCOM) or the Internet Inter-ORB Protocol (IIOP). An ORB is not included as part of the Odyssey system, but instead relies on the installation of Visigenic’s VisiBroker for Java. Even if DCOM or IIOP is used, the RMI registry is still needed by the Finder, which determines where a particular Place

is located. Unlike aglets or Voyager, Odyssey does not include its own SecurityManager, and relies only on the standard Java security mechanisms.

Like the other agents described in this chapter, Odyssey agents are not endowed with intelligence, but, because they are written in Java, can be enhanced by the artificial intelligence techniques demonstrated by the CIAgents in this book. For more information on Odyssey agents, see http://www.genmagic.com

Previous | Table of Contents | Next

JATLite

Java Agent Template Lite (JATLite) is a set of light-weight Java packages being developed at Stanford University that can be used to build multiagent systems. It is a layered architecture which provides a different communication protocol at each layer. The bottom-most layer provides only abstract classes which can be implemented to support any communication protocol. The base layer uses TCP/IP as the communication mechanism and allows the agent implementers to define the message protocol. The KQML layer provides support for agents that use KQML messages. The Router layer adds message routing which allows messages to be queued for agents that are not active when the message is received by the Router.

The JATLite framework is intended for developing typed-message, autonomous agents that communicate using a peer-to-peer protocol. Both synchronous and asynchronous message passing are supported. Messages can be delivered through polling or message queuing. The framework provides additional security which checks the agent name and password for a more secure connection.

The focus of JATLite is obviously on communication. Like many of the other agent environments discussed in this chapter, any intelligence would have to be provided by the agent implementers using techniques described earlier in this book. For more information on JATLite (and its predecessor, JAT), see http://java.stanford.edu/java_agent/html.

InfoSleuth

InfoSleuth is a research project at MCC that uses agents to find, retrieve, and integrate information from heterogeneous data sources. Its architecture is based on cooperating autonomous agents which advertise their services and process requests for those services. It is based on KQML, KIF, HTTP, and Java technologies.

KQML is used as the standard message format and protocol between agents. The InfoSleuth project includes a Java implementation of the KQML language and protocol. KIF is used for exchanging knowledge between agents. The agents

convert information from their internal formats into KIF and wrapper it in a KQML performative before sending the message to another agent.

At the bottom-most layer in the InfoSleuth architecture are rule-based agents written in a variety of languages including Lisp (a list-based AI programming language), LDL++ (a declarative database language that includes rule-based inferencing), CLIPS (a rule-based expert system), and Java. Because KQML and KIF are being used for the interchange of information, it is not necessary for all the agents to be implemented in a single language. On top of the cooperative agents is an ontologies layer which includes agents that use a common vocabulary and semantic model for each particular problem domain. Applications are written on top of the ontologies layer of the architecture.

A number of agents are used in the different layers of the architecture. User agents, written in Java, handle requests from users, route the requests to server agents, and return the results to the user. They are persistent and autonomous, which allows them to run tasks for users even when the user is no longer interacting with the agent. User agents also contain the intelligence to model the user's behavior. To do this, they interact with other Java agents called monitors. A monitor tracks a user's Web access and reports it to the user agent for use in inferencing and pattern detection.

The broker agent is another Java agent that is similar to the Facilitator in our MarketPlace application. It matches requests from agents with ontology server agents that advertise the ontologies for which they have responsibility. The ontology agents interact with execution agents that are responsible for executing the ontology-based queries. The execution agents work with the resource agents which are intelligent front ends to the underlying datastore. The data returned from a query may also be passed to data analysis agents which perform intelligent analysis, knowledge mining, and pattern recognition tasks on the data.

While Java was not the only language used to construct InfoSleuth, it was used throughout the implementation. Applets and applet security were used to safely distribute and view the data. The platform-independent nature of Java also made it possible to deploy the agents in the heterogeneous environment for which InfoSleuth was developed. While CLIPS and other rule-based languages were used to provide much of the intelligence, some of the agents relied on intelligence implemented solely in Java. More information on InfoSleuth can be found at http://www.mcc.com/projects/infosleuth.

Jess

Jess is not an agent environment, per se, but is a subsetted implementation of the CLIPS expert system shell written in Java. It can be used to give Java applications, applets, and agents the ability to reason using a CLIPS rule base.

CLIPS (*C L*anguage *I*ntegrated *P*roduction *S*ystem) was developed by NASA's Johnson Space Center and has been licensed by over 4,000 public and private institutions. It is a forward-chaining rule-based system, based on the Rete algorithm. Code written in the CLIPS language can be run by Jess.

Jess is a Java application that can be run from the command line like any other Java application. But the Jess package also contains classes that enable Java applications or applets to call the expert system engine (the Rete class) or the Jess parser (the Jesp class). This allows Java agents to be developed which can process preexisting CLIPS knowledge bases. For information on Jess, see http://herzberg.ca.sandia.gov/jess/.

ABE

The Agent Builder Environment (ABE) from IBM is a C++ class library and architecture for embedding intelligent agents into applications. We mention ABE in this chapter on Java-based environments because it comes with Java adapters, allowing ABE agents to communicate with Java applications. One of the authors (Joe) contributed to the design and specification of the ABE architecture. In some ways, the CIAgent framework presented in this book is a Java version of ABE-lite.

One of the major differentiators between ABE and most of the other Java-based agent environments is that it is focused on the "intelligent" part of intelligent agents. While IBM has aglets for mobile Java agents, ABE provides a powerful rule-based reasoning engine to control the agent's behavior. This rule-based engine, called RAISE, was developed at the IBM T.J. Watson Research Center and is the heart of ABE. It supports a flexible approach for configuring sensors and effectors for use by the agent as it reasons about the world.

The ABE is more than an agent, it is an architecture and toolkit for developing agents. ABE consists of several major components, including **Engines**, **Knowledge**, **Library**, **Views**, and **Adapters**. The engine in ABE is the RAISE

rule-based engine, but the architecture supports the concept of pluggable engines. Knowledge is represented in KIF, and a Java-based GUI editor is provided for rule authoring. ABE also provides a flexible Library component for storing named KIF rule sets and metadata. The View component defines how editors and dialogs can be attached through ABE adapters to provide clean model-view separation between the intelligent agent processing and agent interactions with users.

Previous | Table of Contents | Next

Adapters are used to interface ABE agents and the RAISE inference engine to applications. For example, ABE comes with adapters to the World Wide Web (HTTP), Internet news servers (NNTP), a Time adapter that triggers events based on time, and a File adapter that triggers events when files are changed. These adapters provide a much richer set of functions than those provided in the CIAgent-based implementation developed in this book. Sample adapters to interface with e-mail systems and to watch stock prices at a Web site are also included. These adapters can be written in Java and they can fully interoperate with other Java or C++ adapters through the ABE. For information on IBM's Agent Builder Environment, see http://www.networking.ibm.com/iag/iaghome.html.

Discussion

Most of the research for this chapter was done on the World Wide Web. During the course of our search for information on Java-based tools and products, it was clear that there is much activity in this area. At the same time, it is also apparent that many of the agent tools were focused more on agent mobility than on artificial intelligence. Without exception, agents were assumed to be autonomous, so it seems that at least that attribute is agreed upon.

We tried to select the set of what appeared to be the most complete Java agent environments. We did not download and try out all of the tools mentioned in this chapter. But from the systems we did examine, some clear patterns could be identified. KQML was cited in several as the interagent communication protocol. KIF was cited in several others as the knowledge representation. In the few systems where artificial intelligence was included, rule-based processing was a fairly standard feature. Learning, however, was not included in most systems.

As with other new technology areas, it seems that intelligent agents will not be a "market" by themselves. Instead, they will be embedded into business applications where they can provide the most immediate value, and slowly evolve into providing must-have features in commercial software. As mentioned earlier, autonomy is now a given for intelligent agents. Mobility clearly has appeal, but it is not clear whether this appeals to the developers or the potential users. The security and infrastructure required for mobile agents to easily move

around the Internet may simply be too much of a hurdle.

Intelligence will be used in agents where the application domain requires it. For many applications today, it seems that hardcoded logic will suffice. Where more flexibility is required, rule-based inferencing is included. Typically these are fairly lightweight inference engines, as far as traditional AI function is concerned. In many applications, such as e-mail automation, simple rule processing is all that is required.

Learning will probably follow this same pattern. The computational overhead of common learning algorithms makes their use a luxury for most applications. However, for intelligent agents based on Internet servers, where their ability to interpret data and patterns in transactions would add significant value, learning may quickly become a necessity. The Firefly Network's collaborative filtering tools utilize learning to provide intelligent personalization and adaptive on-line communities for companies providing high-traffic Web sites. Firefly learning technology can be used to develop a personalized top 10 list for users without requiring keywords or rule authoring (http://www.firefly.com).

As far as Java and intelligent agents go, it seems like a match made in heaven. Java's built-in threading support allows autonomy to be easily added to any Java class. Its serialization support and remote method invocation capabilities make mobility far less daunting than with other programming languages. Finally, its down-to-the-core object-oriented programming support allows artificial intelligence algorithms to be easily implemented, as we hope we have demonstrated in this book.

Our last comment relates to JavaBeans and how the Beans software component model will affect intelligent agents. As we mentioned in Chapter 7, the sophisticated event model, serialization, and visual construction tools make JavaBeans an idea way to develop intelligent agents in Java. Depending on how quickly good visual development tools for Beans get to market, as well as how quickly Beans are accepted in the marketplace, they may become "the" architecture for agents. A Java Bean is a standard encapsulated software component that every computer with a Java Virtual Machine can run. Add mobility to Beans, and you have an unbeatable environment for intelligent agent development and deployment.

Summary

In this chapter, we surveyed several Java intelligent agent environments and tools. These include:

- *Aglets* are IBM's autonomous and mobile Java agent framework. Aglets are free to roam between servers running the aglet host software and can follow a predefined travel itinerary. An aglet process can be suspended, shipped to another aglet host system, and resume as if it never was stopped.
- *FTP Software agents* are Java-based network system management and software distribution agents.
- The *Voyager* system, from ObjectSpace, Inc., is an agent-enhanced Object Request Broker written in Java. Voyager agents are autonomous and mobile, but not intelligent.
- *Odyssey* is General Magic's Java agent system. Written entirely in Java, Odyssey agents use RMI and are autonomous and mobile, but not intelligent.
- Java Agent Template Lite (**JATLite**) is a set of light-weight Java packages developed at Stanford University. JATLite provides a layered architecture for building multiagent systems.
- *InfoSleuth* is a research project at MCC that uses agents to find, retrieve, and integrate information from heterogeneous data sources. It uses KQML and KIF for communication and knowledge representation and provides rule-based agents in Java.
- *Java Expert System Shell (Jess)* is a Java implementation of the standard CLIPS rule base environment developed at Sandia National Lab. While not an agent environment, Jess provides rule-based inferencing support in Java.
- *Agent Builder Environment (ABE)* developed at IBM is an intelligent agent architecture and development toolkit for adding embedded agents to applications. The RAISE inference engine is used as the agent controller, with KIF rules and Library support for knowledge and metadata storage. ABE's architecture includes adapters which interface directly with applications and communicate with the RAISE engine through sensors and effectors.
- The *state-of-the-art of Java intelligent agents* is that they are autonomous and mobile, but not very intelligent. Where artificial intelligence is utilized, most often it is through rule-based inferencing, and seldom with learning. The future for Java and intelligent agents looks bright.

Appendix A
Bugs and Plants Rule Bases

This appendix contains the Java implementation of the bugs and plants rule bases provided with the Rule applet discussed in Chapter 4. The bugs rule base was the first expert system I ever encountered in graduate school at Lehigh University. It was provided with IBM's Expert System Environment (ESE) on their mainframes. The plants rule base first appeared in an article by Beverly and William Thompson in *Byte* magazine (1985).

The Bugs Rule Base Implementation

```
// initialize the Bugs rule base
public void initBugsRuleBase(RuleBase rb) {
  rb.goalClauseStack = new Stack() ; // goals and subgoals

  rb.variableList = new Hashtable() ;
  RuleVariable bugClass = new RuleVariable("bugClass") ;
  bugClass.setLabels("arachnid insect") ;
  rb.variableList.put(bugClass.name,bugClass) ;
  RuleVariable insectType = new RuleVariable("insectType") ;
  insectType.setLabels("beetle orthoptera") ;
  rb.variableList.put(insectType.name,insectType) ;

  RuleVariable species = new RuleVariable("species") ;
  species.setLabels("Spider Tick Ladybug Japanese_Beetle Cricket
Praying_Mantis") ;
  rb.variableList.put(species.name,species) ;

  RuleVariable color = new RuleVariable("color") ;
  color.setLabels("orange_and_black green_and_black black") ;
  rb.variableList.put(color.name,color) ;

  RuleVariable size = new RuleVariable("size") ;
  size.setLabels("small large") ;
  rb.variableList.put(size.name,size) ;

  RuleVariable leg_length = new RuleVariable("leg_length") ;
  leg_length.setLabels("short long") ;
  rb.variableList.put(leg_length.name,leg_length) ;
```

```
 RuleVariable antennae = new RuleVariable(“antennae”) ;
 antennae.setLabels(“0 2”) ;
 rb.variableList.put(antennae.name,antennae) ;

 RuleVariable shape = new RuleVariable(“shape”) ;
 shape.setLabels(“round elongated”) ;
 rb.variableList.put(shape.name,shape) ;

 RuleVariable legs = new RuleVariable(“legs”) ;
 legs.setLabels(“6 8”) ;
 rb.variableList.put(legs.name,legs) ;

 RuleVariable wings = new RuleVariable(“wings”) ;
 wings.setLabels(“0 2”) ;
 rb.variableList.put(wings.name,wings) ;

 // Note: at this point all variables values are NULL

Condition cEquals = new Condition(“=”) ;
Condition cNotEquals = new Condition(“!=”) ;

// define rules
rb.ruleList = new Vector() ;
Rule Spider = new Rule(rb, “spider”,
     new Clause(bugClass,cEquals, “arachnid”)  ,
     new Clause(leg_length,cEquals, “long”),
     new Clause(species,cEquals, “Spider”)) ;

Rule Tick = new Rule(rb, “tick”,
     new Clause(bugClass,cEquals, “arachnid”)  ,
     new Clause(leg_length,cEquals, “short”),
     new Clause(species,cEquals, “Tick”)) ;

Rule Ladybug = new Rule(rb, “ladybug”,
     new Clause(insectType,cEquals, “beetle”)  ,
     new Clause(color,cEquals, “orange_and_black”),
     new Clause(species,cEquals, “Ladybug”)) ;

Rule JapaneseBeetle = new Rule(rb, “Japanese_Beetle”,
     new Clause(insectType,cEquals, “beetle”)  ,
     new Clause(color,cEquals, “green_and_black”),
     new Clause(species,cEquals, “Japanese_Beetle”)) ;

Rule Cricket = new Rule(rb, “cricket”,
     new Clause(insectType,cEquals, “orthoptera”)  ,
     new Clause(color,cEquals, “black”),
     new Clause(species,cEquals, “Cricket”)) ;

Rule PrayingMantis = new Rule(rb, “praying_mantis”,
```

```
      new Clause(insectType,cEquals, “orthoptera”)  ,
      new Clause(color,cEquals, “green”),
      new Clause(size, cEquals, “large”),
      new Clause(species,cEquals, “Praying_Mantis”)) ;

 Rule ClassArachnid1 = new Rule(rb, “class_is_Arachnid1”,
      new Clause(antennae,cEquals, “0”)  ,
      new Clause(legs,cEquals, “8”),
      new Clause(bugClass,cEquals, “arachnid”)) ;

 Rule ClassArachnid2 = new Rule(rb, “class_is_Arachnid2”,
      new Clause(wings,cEquals, “0”)  ,
      new Clause(bugClass,cEquals, “arachnid”)) ;

 Rule ClassInsect1 = new Rule(rb, “class_is_Insect1”,
      new Clause(antennae,cEquals, “2”)  ,
      new Clause(legs,cEquals, “6”),
      new Clause(bugClass,cEquals, “insect”)) ;

 Rule ClassInsect2 = new Rule(rb, “class_is_Insect2”,
      new Clause(wings,cEquals, “2”)  ,
      new Clause(bugClass,cEquals, “insect”)) ;

 Rule TypeBeetle = new Rule(rb, “typeBeetle”,
      new Clause(bugClass,cEquals, “insect”)  ,
      new Clause(size,cEquals, “small”),
      new Clause(shape, cEquals, “round”),
      new Clause(insectType,cEquals, “beetle”)) ;

 Rule TypeOrthoptera = new Rule(rb, “typeOrthoptera”,
      new Clause(bugClass,cEquals, “insect”)  ,
      new Clause(size,cNotEquals, “small”),
      new Clause(shape, cEquals, “elongated”),
      new Clause(insectType,cEquals, “orthoptera”)) ;
}

public void demoBugsBC(RuleBase rb) {
 textArea2.appendText(“\n --- Starting Demo BackwardChain ---\n ”) ;
  // should be a insect, ladybug
((RuleVariable)rb.variableList.get(“wings”)).setValue(“2”) ;
((RuleVariable)rb.variableList.get(“legs”)).setValue(“6”) ;
((RuleVariable)rb.variableList.get(“shape”)).setValue(“round”) ;
((RuleVariable)rb.variableList.get(“antennae”)).setValue(“2”) ;
((RuleVariable)rb.variableList.get(“color”)).setValue(“orange_and_bl
((RuleVariable)rb.variableList.get(“leg_length”)).setValue(“long”) ;
((RuleVariable)rb.variableList.get(“size”)).setValue(“small”) ;
  rb.displayVariables(textArea2) ;
  rb.backwardChain(“species”) ;  // chain until quiescence...
  textArea2.appendText(“\n --- Stopping Demo BackwardChain! ---\n”)
```

```
  rb.displayVariables(textArea2) ;
}

public void demoBugsFC(RuleBase rb) {

 textArea2.appendText("\n --- Starting Demo ForwardChain ---\n") ;
  // should be a insect, ladybug
 ((RuleVariable)rb.variableList.get("bugClass")).setValue(null) ;
((RuleVariable)rb.variableList.get("insectType")).setValue(null) ;
((RuleVariable)rb.variableList.get("wings")).setValue("2") ;
((RuleVariable)rb.variableList.get("legs")).setValue("6") ;
((RuleVariable)rb.variableList.get("shape")).setValue("round") ;
((RuleVariable)rb.variableList.get("antennae")).setValue("2") ;
((RuleVariable)rb.variableList.get("color")).setValue("orange_and_bl
((RuleVariable)rb.variableList.get("leg_length")).setValue("long") ;
((RuleVariable)rb.variableList.get("size")).setValue("small") ;
  rb.displayVariables(textArea2) ;
  rb.forwardChain() ;  // chain until quiescence...
  textArea2.appendText("\n --- Stopping Demo ForwardChain! ---\n") ;

 // now display the results
 rb.displayVariables(textArea2);
 }
```

The Plants Rule Base Implementation

```
// initialize the Plants rule base
 public void initPlantsRuleBase(RuleBase rb) {
  rb.goalClauseStack = new Stack() ; // goals and subgoals
  rb.variableList = new Hashtable() ;

  RuleVariable family = new RuleVariable("family") ;
  family.setLabels("cypress pine bald_cypress ") ;
  rb.variableList.put(family.name,family) ;

  RuleVariable treeClass = new RuleVariable("treeClass") ;
  treeClass.setLabels("angiosperm gymnosperm") ;
  rb.variableList.put(treeClass.name, treeClass) ;

  RuleVariable plantType = new RuleVariable("plantType") ;
  plantType.setLabels("herb vine tree shrub") ;
  rb.variableList.put(plantType.name,plantType) ;

  RuleVariable stem = new RuleVariable("stem") ;
  stem.setLabels("woody green") ;
  rb.variableList.put(stem.name,stem) ;

  RuleVariable stemPosition = new RuleVariable("stemPosition") ;
```

```
 stemPosition.setLabels(“upright creeping”) ;
 rb.variableList.put(stemPosition.name,stemPosition) ;

 RuleVariable one_main_trunk = new RuleVariable(“one_main_trunk”) ;
 one_main_trunk.setLabels(“yes no”) ;
 rb.variableList.put(one_main_trunk.name,one_main_trunk) ;

 RuleVariable broad_and_flat_leaves =
       new RuleVariable(“broad_and_flat_leaves”) ;
 broad_and_flat_leaves.setLabels(“yes no”) ;
 rb.variableList.put(broad_and_flat_leaves.name, broad_and_flat_lea

 RuleVariable leaf_shape = new RuleVariable(“leaf_shape”) ;
 leaf_shape.setLabels(“needlelike scalelike”) ;
 rb.variableList.put(leaf_shape.name,leaf_shape) ;

 RuleVariable needle_pattern = new RuleVariable(“needle_pattern”) ;
 needle_pattern.setLabels(“random even”) ;
 rb.variableList.put(needle_pattern.name,needle_pattern) ;

 RuleVariable silver_bands = new RuleVariable(“silver_bands”) ;
 silver_bands.setLabels(“yes no”) ;
 rb.variableList.put(silver_bands.name,silver_bands) ;

 // Note: at this point all variables values are NULL

Condition cEquals = new Condition(“=”) ;
Condition cNotEquals = new Condition(“!=”) ;

// define rules
rb.ruleList = new Vector() ;
Rule Cypress = new Rule(rb, “cypress”,
     new Clause(treeClass,cEquals, “gymnosperm”)  ,
     new Clause(leaf_shape,cEquals, “scalelike”),
     new Clause(family,cEquals, “cypress”)) ;

Rule Pine1 = new Rule(rb, “pine1”,
     new Clause(treeClass,cEquals, “gymnosperm”)  ,
     new Clause(leaf_shape,cEquals, “needlelike”),
     new Clause(needle_pattern,cEquals, “random”),
     new Clause(family,cEquals, “pine”)) ;

Rule Pine2 = new Rule(rb, “pine2”,
     new Clause(treeClass,cEquals, “gymnosperm”)  ,
     new Clause(leaf_shape,cEquals, “needlelike”),
     new Clause(needle_pattern,cEquals, “even”),
     new Clause(silver_bands,cEquals, “yes”),
     new Clause(family,cEquals, “pine”)) ;

Rule BaldCypress = new Rule(rb, “baldCypress”,
```

```
      new Clause(treeClass,cEquals, “gymnosperm”)  ,
      new Clause(leaf_shape,cEquals, “needlelike”),
      new Clause(needle_pattern,cEquals, “even”),
      new Clause(silver_bands,cEquals, “no”),
      new Clause(family,cEquals, “bald_cypress”)) ;

 Rule Angiosperm = new Rule(rb, “angiosperm”,
      new Clause(plantType,cEquals, “tree”)  ,
      new Clause(broad_and_flat_leaves, cEquals, “yes”),
      new Clause(treeClass,cEquals, “angiosperm”)) ;

 Rule Gymnosperm = new Rule(rb, “gymnosperm”,
      new Clause(plantType,cEquals, “tree”)  ,
      new Clause(broad_and_flat_leaves,cEquals, “no”),
      new Clause(treeClass,cEquals, “gymnosperm”)) ;

 Rule Herb = new Rule(rb, “herb”,
      new Clause(stem,cEquals, “green”)  ,
      new Clause(plantType,cEquals, “herb”)) ;

 Rule Vine = new Rule(rb, “vine”,
      new Clause(stem,cEquals, “woody”)  ,
      new Clause(stemPosition,cEquals, “creeping”),
      new Clause(plantType,cEquals, “vine”)) ;

 Rule Tree = new Rule(rb, “tree”,
      new Clause(stem,cEquals, “woody”)  ,
      new Clause(stemPosition,cEquals, “upright”),
      new Clause(one_main_trunk,cEquals, “yes”),
      new Clause(plantType,cEquals, “tree”)) ;

 Rule Shrub = new Rule(rb, “shrub”,
      new Clause(stem,cEquals, “woody”)  ,
      new Clause(stemPosition,cEquals, “upright”),
      new Clause(one_main_trunk,cEquals, “no”),
      new Clause(plantType,cEquals, “shrub”)) ;
 }

public void demoPlantsBC(RuleBase rb) {
 textArea2.appendText(“\n --- Starting Demo BackwardChain ---\n ”) ;
  // should be a pine tree
((RuleVariable)rb.variableList.get(“stem”)).setValue(“woody”) ;
((RuleVariable)rb.variableList.get(“stemPosition”)).setValue(“uprigh
((RuleVariable)rb.variableList.get(“one_main_trunk”)).setValue(“yes”
((RuleVariable)rb.variableList.get(“broad_and_flat_leaves”)).setValu
((RuleVariable)rb.variableList.get(“leaf_shape”)).setValue(“needleli
((RuleVariable)rb.variableList.get(“needle_pattern”)).setValue(“rand
  rb.displayVariables(textArea2) ;
```

```
  rb.backwardChain("family") ;  // chain until quiescence...
  textArea2.appendText("\n --- Stopping Demo BackwardChain! ---\n")
  rb.displayVariables(textArea2) ;
}

public void demoPlantsFC(RuleBase rb) {

 textArea2.appendText("\n --- Starting Demo ForwardChain --- \n ") ;
 // should be a pine tree
 ((RuleVariable)rb.variableList.get("stem")).setValue("woody") ;
 ((RuleVariable)rb.variableList.get("stemPosition")).setValue("uprig
 ((RuleVariable)rb.variableList.get("one_main_trunk")).setValue("yes
 ((RuleVariable)rb.variableList.get("broad_and_flat_leaves")).setVal
((RuleVariable)rb.variableList.get("leaf_shape")).setValue("needleli
((RuleVariable)rb.variableList.get("needle_pattern")).setValue("rand
  rb.displayVariables(textArea2) ;

  rb.forwardChain() ;  // chain until quiescence...
  textArea2.appendText("\n --- Stopping Demo ForwardChain! --- \n")

  rb.displayVariables(textArea2);
 }
```

Previous | Table of Contents | Next

Appendix B
Training Data Sets

This appendix contains listings of the training data sets provided with the Learn applet discussed in Chapter 5. Each section shows the data definition file (.dfn) and the corresponding data (.dat) file.

Vehicles Data

The vehicles data set was used as the primary example in Chapter 4. It is included in Chapter 5 to show how neural networks and decision trees can classify this data.

```
Vehicles.dfn
discrete motor
discrete num_wheels
discrete num_doors
discrete size
discrete vehicleType
discrete ClassField
```

Vehicles.dat

```
no 2 0 small cycle bicycle
no 3 0 small cycle tricycle
yes 2 0 small cycle motorcycle
yes 4 2 small automobile sportsCar
yes 4 3 medium automobile miniVan
yes 4 4 medium automobile sedan
yes 4 4 large automobile SUV
```

Xor Data

This data set is the standard test for back propagation networks, because it is not a linearly separable classification problem.

Xor.dfn

```
continuous input1
```

```
continuous input2
continuous ClassField
```

Xor.dat

```
0.0 0.0 0.0
1.0 0.0 1.0
0.0 1.0 1.0
1.0 1.0 0.0
```

Animal Data

This data set was shipped with the IBM Neural Network Utility version 3. It is a simple classification problem with a mix of discrete and continuous input data.

Animal.dfn

```
discrete field1
discrete field2
discrete field3
discrete field4
discrete field5
continuous field6
continuous field7
discrete ClassField
```

Animal.dat

```
no  no no  mammal black       4  80000 panther
no  no no  mammal brown       4 100000 lion
no  no yes mammal silver      0  80000 dolphin
yes no no  mammal black/white 4 150000 zebra
yes no no  mammal brown       4  80000 deer
yes no no  bird   black/white 2  60000 ostrich
no  no yes bird   black/white 2  60000 penguin
```

Ramp2 Data

This data set is used to demonstrate the prediction capabilities of back propagation networks.

Ramp2.dfn

```
continuous input1
continuous input2
continuous ClassField
```

Ramp2.dat

```
0.0 0.1 0.1
0.0 0.2 0.2
0.0 0.3 0.3
0.0 0.4 0.4
0.0 0.5 0.5
0.0 0.6 0.6
0.0 0.7 0.7
0.0 0.8 0.8
0.0 0.9 0.9
```

Restaurant Data

This data set is used to test the Decision Tree algorithm. It first appeared in Russell and Norvig's artificial intelligence textbook (1995).

Resttree.dfn

```
discrete alternate
discrete bar
discrete FriSat
discrete hungry
discrete patrons
discrete price
discrete raining
discrete reservation
discrete rtype
discrete waitEstimate
discrete ClassField
```

Resttree.dat

```
yes no  no  yes some $$$ no  yes French  0-10  yes
yes no  no  yes full $   no  no  Thai    30-60 no
no  yes no  no  some $   no  no  Burger  0-10  yes
yes no  yes yes full $   no  no  Thai    10-30 yes
yes no  yes no  full $$$ no  yes French  >60;   no
no  yes no  yes some $$  yes yes Italian 0-10  yes
no  yes no  no  none $   yes no  Burger  0-10  no
```

```
no  no  no  yes some $$  yes yes Thai    0-10  yes
no  yes yes no  full $   yes no  Burger  >60;   no
yes yes yes yes full $$$ no  yes Italian 10-30 no
no  no  no  no  none $   no  no  Thai    0-10  no
yes yes yes yes full $   no  no  Burger  30-60 yes
```

Kmap1 Data

This simple data set is used to test the Kohonen map neural network. It contains points in four corners of a two-dimensional space.

Kmap1.dfn

```
continuous input1
continuous input2
continuous input3
```

Kmap1.dat

```
0.1 0.1 0.1
0.4 0.4 0.4
0.6 0.6 0.6
0.9 0.9 0.9
```

Bibliography

Bigus, J. (1996). *Data Mining with Neural Networks*. New York: McGraw-Hill.

Bigus, J. and Goolsbey, K. (1990). *Combining Neural Networks and Expert Systems in a Commercial Environment*. Proceedings International Joint Conference on Neural Networks, Washington D.C.

Bratko, I. (1986). *Prolog Programming for Artificial Intelligence*. Reading MA: Addison-Wesley.

Brooks, R.A. (1986) A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23.

Chavez, A. and Maes, P. (1996). *Kasbah, An Agent Marketplace for Buying and Selling Goods*. Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, London, UK.

Clocksin, W.F. and Mellish, C.S. (1981). *Programming in Prolog*. Berlin: Springer-Verlag.

Cornell G., and Hortsmann, C.S. (1996). *Core Java*. Upper Saddle River, NJ: Prentice Hall.

Duda, R., Hart, P.E., Nillson, N.J., Reboh, R., Slocum, J., and Sutherland, G. (1977). Development of a computer-based consultant for mineral exploration. *SRI Report*, Stanford Research Institute, Menlo Park, CA.

Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R. (1980). The HEARSAY-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253.

Finin, T., Labrou, Y., and Mayfield, J. (1995). KQML as an agent communication language. In *Software Agents*, J. Bradshaw, ed. Cambridge, MA: MIT Press.

Flanagan, D. (1996). *Java in a Nutshell*. Sebastopol, CA: O'Reilly & Associates.

Forgy, C.L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, vol. 19: 17–37.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Geary, D.M. and McClellan, A.L. (1996). *Graphic Java*. Upper Saddle River, NJ: Prentice Hall.

Gensereth, M.R. and Fikes, R.E. (1992). *Knowledge Interchange Format, version 3.0* Reference Manual.

Holland, J.H., Holyoak, K.J., Nisbett, R.E., and Thagard, P.R. (1986). *Induction: Processes of Inference, Learning, and Discovery*. Cambridge MA: MIT Press.

IBM (1996). Intelligent Agent Resource Manager, Open Blueprint, G325-6592-0.

Jagannathan, V., Dodhiawala, R., Baum, L.S. eds. (1989). *Blackboard Architectures and Applications*. San Diego: Academic Press.

JavaSoft (1996). Java Beans 1.0 API Specification, available at http://java.sun.com/beans.

Kantor, B. and Lapsley, P. (1986). *Network News Transport Protocol: A Proposed Standard for the Stream-Based Transmission of News*. RFC 997, Network Working Group.

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. (1983). Optmization by simulated annealing. *Science*, 220:671–680.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, 78, 1464–1479.

Kuokka, D. and Harada, L. (1995). On using KQML for matchmaking. *Proceedings of First International Conference on Multiagent Systems*. Menlo Park, CA: AAAI Press.

Lander, S.E. (1997). Issues in multiagent design systems. *IEEE Expert*, 12(2):18–26.

Maes, P. (1991). *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*. Cambridge, MA: MIT Press.

Maes, P. (1994). Agents that reduce work and information overload, *Communications of the ACM*, 7:31–40.

McDermott, J. (1982). R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88.

Merriam-Webster, (1988). *Webster's Ninth New Collegiate Dictionary*. Springfield, MA: Merriam-Webster.

Newel, A. (1980). Physical symbol systems, *Cognitive Science* 4:185–203.

Newel, A. and Simon, H.A. (1963) GPS, a program that simulates human thought. In *Computers and Thought*, ed. E.A. Feigenbaum and J. Feldman. New York: McGraw Hill.

Nwana, H.S. (1996). Software agents: An overview. *Knowledge Engineering Review*, 11 (3). 205–244.

Quillian, M.R. (1968). Semantic memory. Minsky, M.L., ed., In *Semantic Information Processing*, pp. 216–270. Cambridge, MA: MIT Press.

Quinlan, J.R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

Rich, E. and Knight, K. (1991). *Artificial Intelligence*. New York: McGraw-Hill.

Robinson, A. J. (1965). A machine-oriented logic based on the resolution principle. *JACM* 12: 23–41.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing*, Vol. 1, pp. 318–62. Cambridge, MA: MIT Press.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.

Sacerdoti, E.D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):155.

Sedgewick, R. (1984). *Algorithms*. Reading, MA: Addison-Wesley.

Shannon, C.E. and Weaver, W. (1949). *The Mathematical Theory of Communication*. University of Illinois Press, Urbana.

Shortliffe, E.H. (1976). *Computer-Based Medical Consultations: MYCIN*. Elsevier.

Stefik, M.J. (1981). Planning and meta-planning. *Artificial Intelligence*, 16:141–169.

Thompson, B. and Thompson, W. (1985). Inside an expert system. *Byte*, April.

Waterman, D.A. (1985). *A Guide to Expert Systems*. Reading, MA: Addison-Wesley.

Williams, J. (1996). *Bots and Other Internet Beasties*. Indianapolis: Sams Net.

Winston, P.H. (1993). *Artificial Intelligence*, third edition. Reading, MA: Addison-Wesley.

Zadeh, L.A. (1994) Fuzzy logic, neural networks and soft computing. *Communications of the ACM* 3: 78-84.

Previous | Table of Contents | Next

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X