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 the first part 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 was made difficult because our readers could not provide this kind of feedback until we had already designed and developed the product (this book). In the first edition, we made some obvious decisions concerning the audience and intended purpose of this book, as indicated by the title. In this second edition, we have taken reader feedback into account. We are going to develop intelligent agents using Java. We are also going to provide the ability to add intelligence to new or existing applications written in Java. An additional fundamental requirement is that we should reuse the artificial intelligence code we developed in the first part of this book.

Another requirement is that our intelligent agent framework must 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 are 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 1.2 development environment.

Having just said all this, we 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 Borland JBuilder 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 Borland 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 agents to plug into a simple agent platform, to handle information filtering over the Internet, and to perform 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 that we can use to guide the technical decisions that 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 that 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 that 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 monitor and drive the applications. Here our agent manager itself is an application and must interface with other applications that 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 agent framework we are developing 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 generally *protected* and accessor functions will be used in the majority of cases. *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 inter-agent communication is supported. It must also be flexible so that support for new applications, AI techniques, and other features can be easily added.

Functional Specifications

In this section, we take the requirements and our design goals, and turn them into a list of functions that 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 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 or from 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 agents must be persistent; 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 rather than direct method invocations to communicate between the application and agent. Because this gives us both autonomy and flexibility, this is the design we will pursue.

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 Borland's JBuilder, WebGain's Visual Café, and IBM's VisualAge for Java that allow you to construct applications using a "construction from parts" metaphor. However, 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 that allows a visual builder tool to manipulate that object. It also has a defined runtime interface that allows applications comprised of JavaBeans to run.

   Another nice feature of Beans is that they can be nested, meeting our requirement for the ability to compose agents out of other agents. This allows us to develop special-purpose agents that can be reused in other higher-level agents. For example, we can have low-level agents that use neural networks for learning and high-level agents that 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).

3. The agents must support a relatively sophisticated event-processing capability. Our agent will need to handle events from the outside world or from other agents, and signal events to outside applications.

   The JDK 1.1 release featured a powerful event-processing model called the Delegation Event Model. This event framework design 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. The agent can be an **EventListener**, and whenever the agent is notified that an event occurred, it can process the event. We can describe our own event type by subclassing the **EventObject** class.

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 **BooleanRuleBase**, **Rule**, and **RuleVariable** classes we developed in Chapter 4, "Reasoning Systems," 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 way for users to specify conditions and actions without programming. If we build a JavaBeans property editor for our **BooleanRuleBase** class, a user could easily construct a set of **RuleVariable**s and **Rule**s 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 JavaBean 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, "Learning Systems," we designed and developed decision tree classifiers, 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 drawing board 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 Java 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 agents must be persistent; 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 *java.io* package in the Java run-time supports serialization of Java objects. Simply marking the class as implementing the **Serializable** interface makes saving and loading objects very easy to do. Any data members that are not explicitly declared **transient** will be saved out to a file. This is another advantage of using Java to implement our agent framework.

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 One 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. Figure 7.1 shows all of the classes in the *ciagent* package. Before we get to the **CIAgent** abstract base class that defines the common interface used by the elements of our architecture, we introduce several utility classes.

The CIAgent Base Classes

CIAgent is the base class that 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 the agents into groups, and still treat the group as if it were 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.

Figure 7.2 shows the relationship between the classes that make up a functional **CIAgent** instance. In the following paragraphs we describe each class in detail.

The **CIAgent** class uses several helper classes including **CIAgentState**, **CIAgentTimer**, and **CIAgentEventQueue**. **CIAgentState**, shown in Figure 7.3, contains a

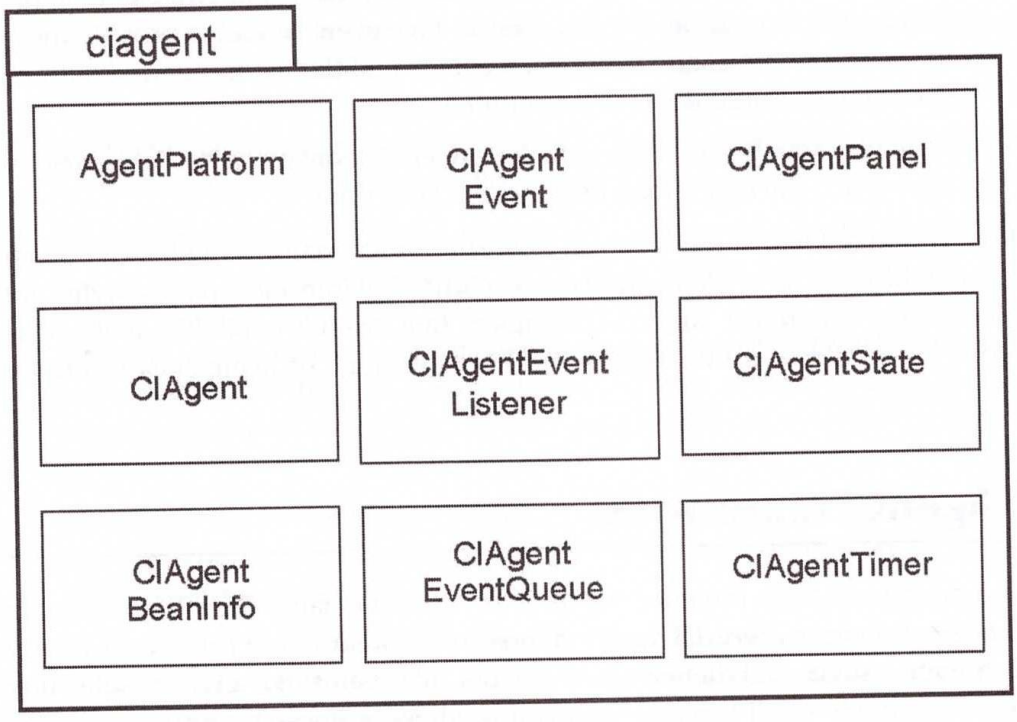

Figure 7.1 The *ciagent* package UML diagram.

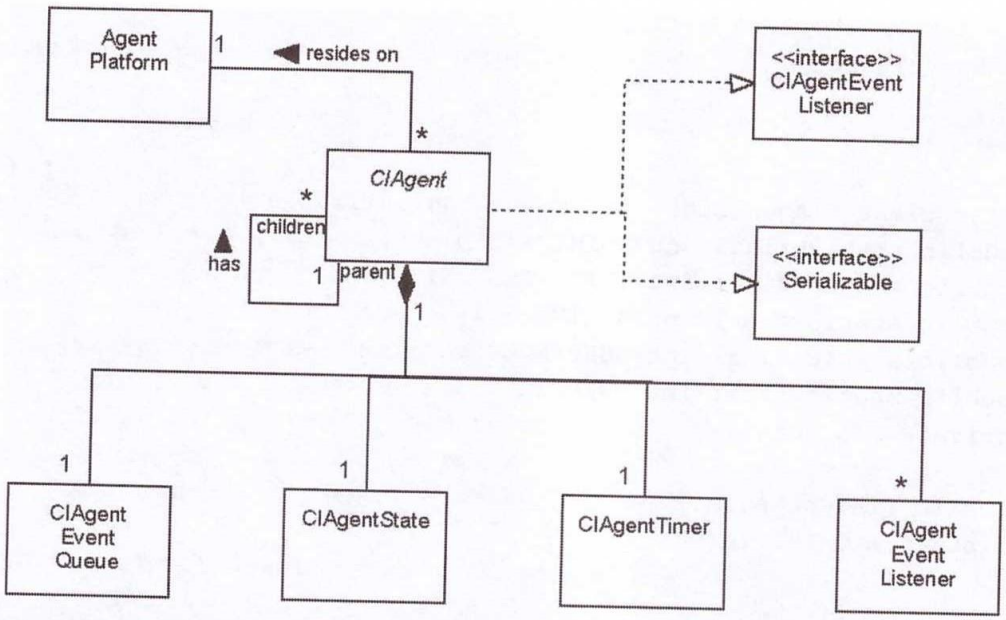

Figure 7.2 The CIAgent UML class diagram.

single data member, the **int** *state*. When a **CIAgentState** object is constructed, its *state* is set to UNINITIATED. Changes can be made using the *setState()* method.

The **CIAgentTimer** class, shown in Figure 7.4, implements the **Runnable** interface, which requires that it implement the *run()* method, which is called by the *runnit* **Thread**. This is the mechanism we use to give our agents autonomy. The **CIAgentTimer** provides two basic functions to our **CIAgents**. The first is the autonomous behavior where the agent's *processTimerPop()* method gets called every *sleepTime* milliseconds. The other is the ability to process events in an asynchronous manner by queueing them up and then processing them every *asyncTime* milliseconds. Both of these behaviors are supported by a single thread. The timer is controlled by the *startTimer()*, *stopTimer()*, and *quitTimer()* methods. The *startTimer()* method creates the thread the first time and starts it if it isn't already running. The *stopTimer()* method simply sets a *timerEnabled* **boolean** to false where the *run()* method will see it and will avoid calling the *processTimerPop()* method. The *quitTimer()* method sets the *quit* flag to true, causing the *run()* method to exit and the *runnit* thread to end. Note that the **CIAgentTimer** class is coded to have a single lifecycle.

To communicate with other **CIAgents** and other JavaBeans, we implement the **CIAgentEventListener** interface, as shown in Figure 7.5. This interface extends the standard Java **EventListener** interface used by all *java.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 *CIAgent()* constructor. A **Vector** of listeners holds all Java objects which implement the **CIAgentEventListener** interface and which have registered themselves using the *addCIAgentEventListener()* method. Any Java **Object** can be the event source for **CIAgentEvents**, and any object that implements the **CIAgentEventListener** interface can be a registered listener for those events. The **CIAgent** class provides the *addCIAgentEventListener()* and

```
package ciagent;

import java.io.*;

public class CIAgentState implements Serializable {
  public static final int UNINITIATED = 0;
  public static final int INITIATED = 1;
  public static final int ACTIVE = 2;
  public static final int SUSPENDED = 3;
  public static final int UNKNOWN = 4;
  private int state;

  public CIAgentState() {
    state = UNINITIATED;
  }

  public synchronized void setState(int state) {
    this.state = state;
  }

  public int getState() {
    return state;
  }

  public String toString() {
    switch(state) {
      case UNINITIATED: {
        return "Uninitiated";
      }
      case INITIATED: {
        return "Initiated";
      }
      case ACTIVE: {
        return "Active";
      }
      case SUSPENDED: {
        return "Suspended";
      }
      case UNKNOWN: {
        return "Unknown";
      }
    }
    return "Unknown";
  }
}
```

Figure 7.3 The CIAgentState class listing.

```
package ciagent;

import java.util.*;
import java.io.*;

public class CIAgentTimer implements Runnable, Serializable {
  private CIAgent agent;
  private int sleepTime = 1000;
  private boolean timerEnabled = true;
  private int asyncTime = 500;
  transient private Thread runnit = new Thread(this);
  private boolean quit = false;
  private boolean debug = false;

  public CIAgentTimer(CIAgent agent) {
    this.agent = agent;
  }

  public void setSleepTime(int sleepTime) {
    this.sleepTime = sleepTime;
  }

  public int getSleepTime() {
    return sleepTime;
  }

  public void setAsyncTime(int asyncTime) {
    this.asyncTime = asyncTime;
  }

  public int getAsyncTime() {
    return asyncTime;
  }

  public void startTimer() {
    timerEnabled = true;
    if(!runnit.isAlive()) {
      runnit.start();
    }
  }

  public void stopTimer() {
    timerEnabled = false;
  }

  public void quitTimer() {
```

(continues)

Figure 7.4 The CIAgentTimer class listing.

```
    quit = true;
  }

  public void run() {
    long startTime = 0;
    long curTime = 0;

    if(debug) {
      startTime = new Date().getTime();
      curTime = startTime;
    }
    if(sleepTime < asyncTime) {
      asyncTime = sleepTime;
    }
    int numEventChecks = sleepTime / asyncTime;

    if(debug) {
      System.out.println("sleepTime= " + sleepTime + " asyncTime= "
                         + asyncTime + "numEventChecks= "
                         + numEventChecks);
    }
    while(quit == false) {
      try {
        for(int i = 0; i < numEventChecks; i++) {
          Thread.sleep(asyncTime);
          if(debug) {
            curTime = new Date().getTime();
            System.out.println("async events timer at "
                               + (curTime - startTime));
          }
          if(quit) {
            break;
          }
            agent.processAsynchronousEvents();
        }
        if(timerEnabled && (quit == false)) {
          if(debug) {
            curTime = new Date().getTime();
            System.out.println("timer event at " + (curTime -
              startTime));
          }
          agent.processTimerPop();
        }
      } catch(InterruptedException e) {}
    }
  }
```

Figure 7.4 The CIAgentTimer class listing (Continued).

```
    private void readObject(ObjectInputStream theObjectInputStream)
        throws ClassNotFoundException, IOException {
      runnit = new Thread(this);
      theObjectInputStream.defaultReadObject();
    }
}
```

Figure 7.4 Continued.

removeCIAgentEventListener() methods so that other listeners can be added to the multicast event notification list. These methods fully support the JavaBeans event API, so **CIAgents** can be wired up using any visual builder tool that supports JavaBeans.

The *notifyCIAgentEventListeners()* method is used to send events to registered listeners. Note that the *addCIAgentEventListener()*, *removeCIAgentEventListener()*, and *notifyCIAgentEventListener()* methods 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 bean's **Customizer** or property editor. We use the JavaBean **PropertyChangeSupport** class to make the *name* a bound property. When the *name* is changed, other property change listeners will be notified. **PropertyChange** events are used in JavaBeans to signal configuration or state changes, while the **CIAgentEvents** are used for agent communication while processing an application.

The other methods we define include the *initialize()* and *reset()* methods for getting the agent to a known state, the *startAgentProcessing()* and *stopAgentProcessing()* methods for starting the agent-processing thread or stopping it. The *suspendAgentProcessing()* and *resumeAgentProcessing()* methods can be used to temporarily stop the autonomous behavior and invocation of the *processTimerPop()* method.

The CIAgents provided in this book could be easily turned into **Applets**, or with subclassing of an **awt.Component**, they could be made into visible JavaBeans. As implemented here, our CIAgents are invisible JavaBeans, meaning that they can be used in the Visual Builder environment, but they do not represent graphical components in the application GUI.

We provide the *addAgent()* and *removeAgent()* methods to build composite **CIAgents** and the *getChildren()* method to access any contained agents. The **CIAgent** class implements the **Serializable** interface so that we can save and load **CIAgent** objects to files. We have marked several data members as **transient**, meaning that they will not be saved when the object is serialized out to a file. Consequently, we must implement the *readObject()* method, which recreates the transient members when the **CIAgent** is being deserialized. Note that if we didn't do this, serialization would also serialize all objects registered as *listeners* and all events on the *eventQueue*.

```
package ciagent;

import java.util.*;
import java.awt.*;
import javax.swing.*;
import java.beans.*;
import java.io.*;

public abstract class CIAgent
    implements CIAgentEventListener, Serializable {
  public static final int DEFAULT_SLEEPTIME = 15000;
  public static final int DEFAULT_ASYNCTIME = 1000;
  protected String name;
  private CIAgentState state = new CIAgentState();
  private CIAgentTimer timer = new CIAgentTimer(this);
  transient private Vector listeners = new Vector();
  transient private CIAgentEventQueue eventQueue = new
    CIAgentEventQueue();
  transient private PropertyChangeSupport changes =
    new PropertyChangeSupport(this);
  private boolean traceOn = false;
  protected int traceLevel = 0;
  protected AgentPlatform agentPlatform = null;
  protected Vector children = new Vector();
  protected CIAgent parent = null;

  public CIAgent() {
    this("CIAgent");
  }

  public CIAgent(String name) {
    this.name = name;
    timer.setAsyncTime(DEFAULT_ASYNCTIME);
    timer.setSleepTime(DEFAULT_SLEEPTIME);
    state.setState(CIAgentState.UNINITIATED);
  }

  public void setName(String newName) {
    String oldName = name;

    name = newName;
    changes.firePropertyChange("name", oldName, name);
  }

  public String getName() {
    return name;
  }
```

Figure 7.5 The CIAgent class listing.

```
  protected void setState(int newState) {
    int oldState = state.getState();

    changes.firePropertyChange("state", oldState, newState);
    this.state.setState(newState);
  }

  public CIAgentState getState() {
    return state;
  }

  public void setSleepTime(int sleepTime) {
    timer.setSleepTime(sleepTime);
  }

  public int getSleepTime() {
    return timer.getSleepTime();
  }

  public void setAsyncTime(int asyncTime) {
    timer.setAsyncTime(asyncTime);
  }

  public int getAsyncTime() {
    return timer.getAsyncTime();
  }

  public void setTraceLevel(int traceLevel) {
    this.traceLevel = traceLevel;
  }

  public int getTraceLevel() {
    return traceLevel;
  }

  public void setAgentPlatform(AgentPlatform agentPlatform) {
    this.agentPlatform = agentPlatform;
  }

  public AgentPlatform getAgentPlatform() {
    return agentPlatform;
  }

  public Vector getAgents() {
    if(agentPlatform == null) {
      return null;
```

(continues)

Figure 7.5 Continued.

```
    } else {
      return agentPlatform.getAgents();
    }
  }

  public CIAgent getAgent(String name) {
    if(agentPlatform == null) {
      return null;
    } else {
       return agentPlatform.getAgent(name);
    }
  }

  public abstract String getTaskDescription();

  public Vector getChildren() {
    return (Vector) children.clone();
  }

  public void setParent(CIAgent parent) {
    this.parent = parent;
  }

  public CIAgent getParent() {
    return parent;
  }

  public Class getCustomizerClass() {
    Class customizerClass = null;

    try {
      BeanInfo beanInfo = Introspector.getBeanInfo(this.getClass());
      BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();

      customizerClass = beanDescriptor.getCustomizerClass();
    } catch(IntrospectionException exc) {
      System.out.println("Can't find customizer bean property " +
        exc);
    }
    return customizerClass;
  }

  public String getDisplayName() {
    String name = null;

    try {
      BeanInfo beanInfo = Introspector.getBeanInfo(this.getClass());
```

Figure 7.5 The CIAgent class listing (Continued).

```
        BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();

        name = (String) beanDescriptor.getValue("DisplayName");
      } catch(IntrospectionException exc) {
        System.out.println("Can't find display name bean property " +
          exc);
      }
      if(name == null) {
        name = this.getClass().getName();
      }
      return name;
    }

    public void reset() {}

    public abstract void initialize();

    public synchronized void startAgentProcessing() {
      timer.startTimer();
      setState(CIAgentState.ACTIVE);
    }

    public synchronized void stopAgentProcessing() {
      timer.quitTimer();
      setState(CIAgentState.UNKNOWN);
    }

    public void suspendAgentProcessing() {
      timer.stopTimer();
      setState(CIAgentState.SUSPENDED);
    }

    public void resumeAgentProcessing() {
      timer.startTimer();
      setState(CIAgentState.ACTIVE);
    }

    public abstract void process();

    public abstract void processTimerPop();

    public void processAsynchronousEvents() {
      CIAgentEvent event = null;

      while((event = eventQueue.getNextEvent()) != null) {
```

(continues)

Figure 7.5 Continued.

```
      processCIAgentEvent(event);
    }
  }

  public void processCIAgentEvent(CIAgentEvent event) {}

  public void postCIAgentEvent(CIAgentEvent event) {
    eventQueue.addEvent(event);
  }

  public synchronized void addCIAgentEventListener(
      CIAgentEventListener listener) {
    listeners.addElement(listener);
  }

  public synchronized void removeCIAgentEventListener(
      CIAgentEventListener listener) {
    listeners.removeElement(listener);
  }

  protected void notifyCIAgentEventListeners(CIAgentEvent e) {
    Vector l;

    synchronized(this) {
      l = (Vector) listeners.clone();
    }
    for(int i = 0; i < l.size(); i++) {
      ((CIAgentEventListener) l.elementAt(i)).processCIAgentEvent(e);
    }
  }

  public synchronized void addPropertyChangeListener(
      PropertyChangeListener listener) {
    changes.addPropertyChangeListener(listener);
  }

  public synchronized void removePropertyChangeListener(
      PropertyChangeListener listener) {
    changes.removePropertyChangeListener(listener);
  }

  public void trace(String msg) {
    CIAgentEvent event = new CIAgentEvent(this, "trace", msg);

    notifyCIAgentEventListeners(event);
  }
```

Figure 7.5 The CIAgent class listing (Continued).

```
    public void addAgent(CIAgent child) {
      children.addElement(child);
      child.setParent(this);
    }

    public void removeAgent(CIAgent child) {
      children.removeElement(child);
    }

    private void readObject(ObjectInputStream theObjectInputStream)
        throws ClassNotFoundException, IOException {
      changes = new PropertyChangeSupport(this);
      listeners = new Vector();
      eventQueue = new CIAgentEventQueue();
      theObjectInputStream.defaultReadObject();
    }
}
```

Figure 7.5 Continued.

CIAgentEvent

The **CIAgentEvent** class, shown in Figure 7.6, is derived from the Java **EventObject** class, as required by the JavaBeans specification. There are three *CIAgentEvent()* constructors defined. The first takes a single parameter, the *source*, which is a reference to the object sending the event. The second constructor creates a notification event, which takes the *source* and an event argument object. The third constructor is used to create an action event and requires the *source* object, an *action* **String** that can be a method name in the target object, and an event argument object. The argument is defined 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, "MarketPlace Application," we will use this flexibility to define a KQML-like message object. Getter methods are provided for the action and argument properties.

CIAgentEventListener

The **CIAgentEventListener** interface, in Figure 7.7, extends the **EventListener** interface and requires that two methods be implemented. These are the *postCIAgentEvent()* method which usually would place the event on an **CIAgentEventQueue** for later asynchronous processing, and the *processCIAgentEvent()* method, which usually results in immediate processing of the event on the caller's thread.

```
package ciagent;

import java.util.*;

public class CIAgentEvent extends java.util.EventObject {
  private Object argObject = null;
  private String action = null;

  public CIAgentEvent(Object source) {
    super(source);
  }

  public CIAgentEvent(Object source, Object argObject) {
    this(source);
    this.argObject = argObject;
  }

  public CIAgentEvent(Object source, String action, Object argObject)
{
    this(source);
    this.action = action;
    this.argObject = argObject;
  }

  public Object getArgObject() {
    return argObject;
  }

  public String getAction() {
    return action;
  }

  public String toString() {
    StringBuffer buf = new StringBuffer();

    buf.append("CIAgent ");
    buf.append("source: " + source);
    buf.append("action: " + action);
    buf.append("argObject: " + argObject);
    return buf.toString();
  }
}
```

Figure 7.6 The CIAgentEvent class listing.

```
package ciagent;

import java.util.*;

public interface CIAgentEventListener extends java.util.EventListener
{
  public void processCIAgentEvent(CIAgentEvent e);

  public void postCIAgentEvent(CIAgentEvent e);
}
```

Figure 7.7 CIAgentEventListener interface listing.

CIAgentEventQueue

The **CIAgentEventQueue** class, listed in Figure 7.8, implements a queue of **CIAgentEvents**. Its only data member is the *eventQueue*, a **Vector** instance. The *addEvent()* method is used to add an event to the end of the queue, the *getNextEvent()* method is a non-blocking method that either returns null if the queue is empty, or removes the first event from the queue and then returns it. The *peekEvent()* method is used to test whether the queue is empty. It either returns null, meaning the queue is empty, or it returns the event at the front of the queue without removing it.

The methods provided by the **CIAgent** base class are intended to be used in a specific sequence so that the **CIAgentState** follows a predictable set of transitions. The lifecycle of a **CIAgent** should follow this sequence of states:

1. Construct the **CIAgent** object instance. The state is set to UNINITIATED.
2. Either programmatically or using a **Customizer**, set the JavaBean properties.
3. Call the *initialize()* method. The state is set to INITIATED.
4. Call the *startAgentProcessing()* method to start the *eventQueue* thread and any asynchronous event and timer processing. The state is set to ACTIVE.
5. Use the agent in an application by calling the *process()* method directly, or by sending action events to be processed asynchronously. If the *processTimerPop()* method is overridden and the timer *sleepTime* is set, the agent can also perform periodic autonomous processing.
6. Use the *suspendAgentProcessing()* and *resumeAgentProcessing()* methods to temporarily halt the autonomous behavior and the invocation of the *processTimerPop()* method. Note that this does not stop the asynchronous **CIAgentEvent** processing (if enabled). The agent state goes to SUSPENDED

```
package ciagent;

import java.util.*;
import java.io.*;

public class CIAgentEventQueue implements Serializable {
  private Vector eventQueue;

  public CIAgentEventQueue() {
    eventQueue = new Vector();
  }

  public synchronized void addEvent(CIAgentEvent event) {
    eventQueue.addElement(event);
  }

  public synchronized CIAgentEvent getNextEvent() {
    if(eventQueue.size() == 0) {
      return null;
    } else {
      CIAgentEvent event = (CIAgentEvent) eventQueue.elementAt(0);

      eventQueue.removeElementAt(0);
      return event;
    }
  }

  public synchronized CIAgentEvent peekEvent() {
    if(eventQueue.size() == 0) {
      return null;
    } else {
      return (CIAgentEvent) eventQueue.elementAt(0);
    }
  }
}
```

Figure 7.8 CIAgentEventQueue class listing.

after a *suspendAgentProcessing()* call and back to ACTIVE when *resumeAgentProcessing()* is called.

7. Call the *stopCIAgentProcessing()* method to stop the *eventTimer* thread. The state is set to UNKNOWN.

It is up to the implementers of the **CIAgent** subclass to ensure the agent *state* conforms to this expected behavior. The state transitions are illustrated in the UML diagrams in Figure 7.9 and Figure 7.10. Figure 7.9 shows the start-up sequence from creation to initialization. Figure 7.10 presents the interactions for synchronous and asynchronous event processing and autonomous agent behavior.

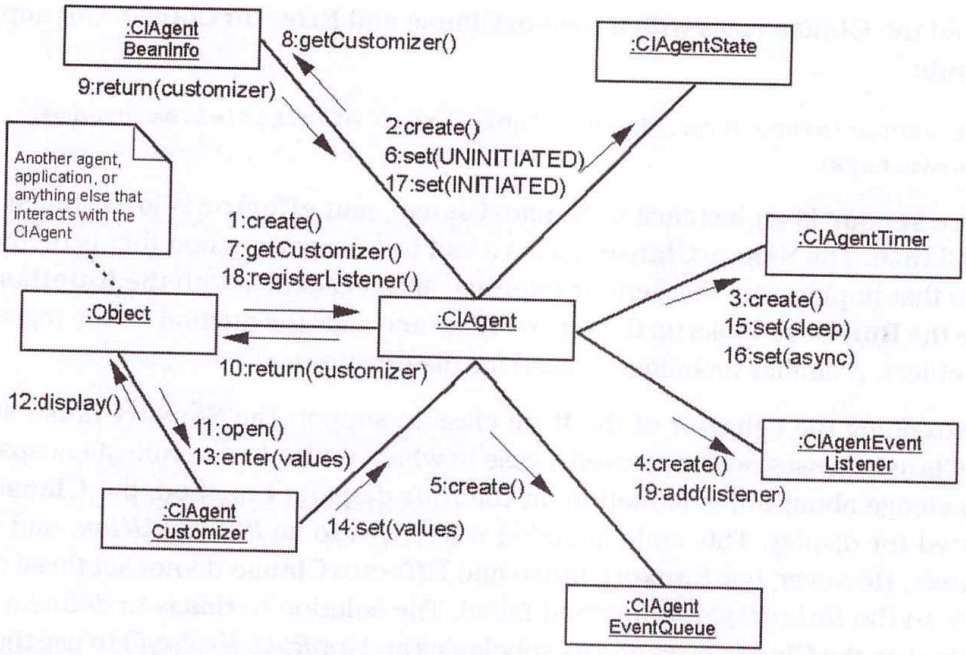

Figure 7.9 The CIAgent start-up collaboration diagram.

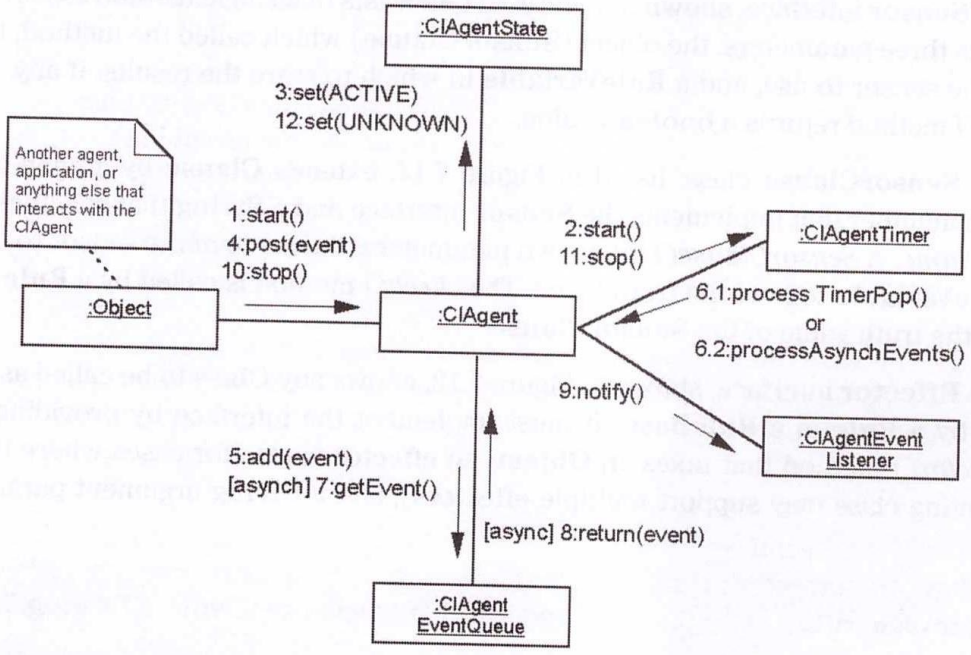

Figure 7.10 The CIAgent action collaboration diagram.

BooleanRuleBase Enhancements

In this section, we describe the enhancements to our **BooleanRuleBase** 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 that 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 in which 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 an *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.

The **Sensor** interface, shown in Figure 7.11, 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.

The **SensorClause** class, listed in Figure 7.12, 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**.

The **Effector** interface, shown in Figure 7.13, 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

```
package rule;

import java.util.*;
import java.io.*;

public abstract interface Sensor {
  public Boolean sensor(Object obj, String sName, RuleVariable lhs);
}
```

Figure 7.11 The Sensor interface listing.

```
package rule;

import java.util.*;
import java.io.*;

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 = false;
    sensorName = sName;
  }

  public String display() {
    return "sensor(" + sensorName + "," + rhs + ") ";
  }

  public Boolean check() {
    if(consequent == true) {
      return null;
    }
    if(lhs.value == null) {
      BooleanRuleBase rb = ((Rule) ruleRefs.firstElement()).rb;

      object = (Sensor) (rb.getSensorObject(sensorName));
      truth = object.sensor(this, sensorName, lhs);
    }
    return truth;
  }
}
```

Figure 7.12 The SensorClause class listing.

alternate implementation that may be useful would be to support an **Object** as the argument parameter.

The **EffectorClause** class, listed in Figure 7.14, 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

```
package rule;
import java.util.*;
import java.io.*;

public abstract interface Effector {
  public long effector(Object obj, String eName, String args);
}
```

Figure 7.13 The Effector interface listing.

argument **String**. The *perform()* method is provided to call the *effector()* method on the object that has registered to provide that effector function.

To support facts, we add a new class called **Fact**, shown in Figure 7.15, whose constructor takes a single clause as a parameter. A **Fact** can be an assignment of a value to

```
package rule;

import java.util.*;
import java.io.*;

public class EffectorClause extends Clause {
  Effector object;
  String effectorName;
  String arguments;

  public EffectorClause(String eName, String args) {
    ruleRefs = new Vector();
    truth = new Boolean(true);
    consequent = true;
    effectorName = eName;
    arguments = args;
  }

  public String display() {
    return "effector(" + effectorName + "," + arguments + ") ";
  }

  public Boolean perform(BooleanRuleBase rb) {
    object = (Effector) (rb.getEffectorObject(effectorName));
    object.effector(this, effectorName, arguments);
    return truth;
  }
}
```

Figure 7.14 The EffectorClause class listing.

```
package rule;

import java.util.*;
import java.io.*;
import java.awt.*;
import javax.swing.*;

public class Fact {
  BooleanRuleBase rb;
  String name;
  Clause fact;
  Boolean truth;
  boolean fired = false;

  Fact(BooleanRuleBase Rb, String Name, Clause f) {
    rb = Rb;
    name = Name;
    fact = f;
    rb.addFact(this);
    truth = null;
  }

  public void assert(BooleanRuleBase rb) {
    if(fired == true) {
      return;
    }
    rb.trace("\nAsserting fact " + name);
    truth = new Boolean(true);
    fired = true;
    if(fact.lhs == null) {
      ((EffectorClause) fact).perform(rb);
    } else {
      fact.lhs.setValue(fact.rhs);
    }
  }

  void display(JTextArea textArea) {
    textArea.append(name + ": ");
    textArea.append(fact.toString() + "\n");
  }
}
```

Figure 7.15 The Fact class listing.

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.

The main reason for adding sensors and effectors is to add procedural attachments to our rule-processing capability. Instead of being limited to the variables we have defined in the rule base, we can call outside methods to gather data using sensors, or to take actions when a rule fires using effectors. We use the effectors in the *Marketplace* application in Chapter 10, "MarketPlace Application."

Discussion

Now that we have defined the **CIAgent** framework, what can we do with it? In the next three chapters we explore this question by constructing ten different agents and using them in a variety of applications. As we go through those chapters, we make use of the flexibility in our **CIAgent** design; in some cases we tightly couple the agents to the application, while in others we create an agent platform where our agents exchange **CIAgentEvent**s to communicate.

While the **CIAgent** framework is not complex, it still has the basic attributes required of agents and agent applications. Our **CIAgent**s can be autonomous and they can handle asynchronous events, so they can act like daemons. This behavior is provided by our **CIAgentTimer** and **CIAgentEventQueue** classes. **CIAgent**s can be composed to build a complicated hierarchy of function by virtue of our use of the *Composite* design pattern.

Our agents are JavaBeans, so they can be mixed with other non-agent Java classes and components. With the addition of **BeanInfo** and **Customizer** classes, the **CIAgent**s can be used with most Visual Builder development tools. We make use of this flexibility in our first application, described in the next chapter.

Summary

In this chapter we described the **CIAgent** intelligent agent framework. The main points include the following:

- 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 compose agents out of other Java components and other agents.
  3. The agents must support a relatively sophisticated event-processing capability.
  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.