CHAPTER 8

Personal Agent Manager Application

In this chapter, we illustrate how we can use the **CIAgent** architecture to construct a personal assistant application that uses several intelligent agents to assist a user with tasks. The application provides an agent platform that serves as the environment in which the agents live and work with one another. One of the agents developed in this chapter watches a file on the local system and alerts the user, executes a command, or signals another agent when a trigger condition is met. This agent is then decomposed into smaller helper agents that handle the simple tasks of scheduling events and notifying the user when events occur. The final agent is another task-based agent that checks airline flights and fares on a Web site and notifies the user about flights matching the user's criteria.

Introduction

The first intelligent agent application we develop is a basic agent platform that allows us to create, configure, and control a set of personal assistant agents that will do tasks on our behalf. Our goal is to provide a flexible base that illustrates the general concepts of a multiagent platform and that can be easily extended by plugging in additional agents. The *PAManager* application provides the graphical user interface for managing the agents and makes use of the agent customizer dialogs to allow interaction with the other **CIAgent**s running on the platform. The user can start, suspend, or resume agent execution from within the *PAManager* application or can stop and remove an agent from the platform.

The *PAManager* application classes are **PAManagerApp**, which contains the *main()*, and **PAManagerFrame**, which implements our main window. They were constructed

using the Borland/Inprise JBuilder 3.0 Java interactive development environment. The Visual Builder allows us to create the complete GUI using Java Swing components in a drag-and-drop style. JBuilder automatically generates the Java code for creating the GUI controls and action event handlers in the **PAManagerFrame** class. The **PAManagerApp** code that invokes the **PAManagerFrame** is not presented here. It simply instantiates the **PAManagerFrame** class and displays it. Likewise, a substantial amount of the code in the **PAManagerFrame** class deals with GUI control logic, and we will not present that material here. All of the classes discussed in this chapter are in the *pamanager* package, shown in Figure 8.1.

The FileAgent

The first agent we will create to run on the *PAManager* platform is the **FileAgent**, listed in Figure 8.2. A **FileAgent** monitors a file or directory on the current system. When the **FileAgent** detects that a specific file has been changed or deleted, or has grown to a size larger than a given threshold, the **FileAgent** will take some action. The user can specify that the **FileAgent** displays an alert message, executes a command, or sends an event to another agent.

The three conditions that can be checked by the **FileAgent** are: whether the file has been MODIFIED, whether the file has been DELETED (it does not exist in the file system), or whether the file exceeds a THRESHOLD size. The *action* taken by the **FileAgent** when the *condition* occurs can be: ALERT, if the user is to be notified; EXECUTE, if a command should be executed; or EVENT, if a **CIAgentEvent** should be sent to another agent. 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 *threshold* contains the size, in bytes, that is being compared against the actual file size.

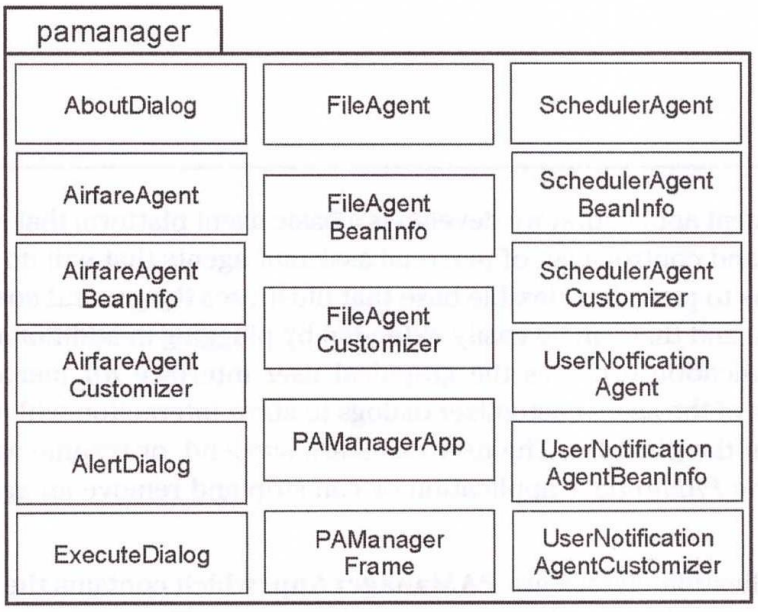


Figure 8.1 The *pamanager* package UML diagram.

```
package pamanager;

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

public class FileAgent extends CIAgent {
  public static final int MODIFIED = 0;
  public static final int DELETED = 1;
  public static final int THRESHOLD = 2;
  public static final int ALERT = 0;
  public static final int EXECUTE = 1;
  public static final int EVENT = 2;
  protected int condition;
  protected int action = EVENT;
  protected String fileName;
  protected File file;
  protected long lastChanged;
  protected int threshold;
  protected JDialog actionDialog;
  protected String actionString;
  protected String parms;

  public FileAgent() {
    name = "Watch";
  }

  public FileAgent(String name) {
    super(name);
  }

  public String getTaskDescription() {
    return "Watching filename=" + fileName + "condition=" + condition;
  }

  public void setFileName(String fileName) {
    this.fileName = fileName;
    file = new File(fileName);
    lastChanged = file.lastModified();
  }

  public String getFileName() {
    return fileName;
  }
```

(continues)

Figure 8.2 The FileAgent class listing.

```
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 action) {
  this.action = action;
}

public int getAction() {
  return action;
}

public void setActionString(String actionString) {
  this.actionString = actionString;
}

public String getActionString() {
  return actionString;
}

public void setParms(String params) {
  parms = params;
}

public String getParms() {
  return parms;
}

public void setDialog(JDialog dlg) {
  actionDialog = dlg;
}

public void initialize() {
  if(actionDialog != null) {
    actionDialog.dispose();
```

Figure 8.2 The FileAgent class listing (Continued).

```
    }
    JFrame frame = new JFrame();

    if(action == ALERT) {
      actionDialog = new AlertDialog(frame, name + ": Alert", false);
    }
    if(action == EXECUTE) {
      actionDialog = new ExecuteDialog(frame, name + ": Execute",
        false);
    }
    setSleepTime(15 * 1000);
    setState(CIAgentState.INITIATED);
  }

  public void process() {
    if(checkCondition()) {
      performAction();
    }
  }

  public void processCIAgentEvent(CIAgentEvent e) {}

  public void processTimerPop() {
    process();
  }

  private boolean checkCondition() {
    boolean truth = false;

    switch(condition) {
      case MODIFIED:
        truth = changed();
        break;
      case DELETED:
        truth = !exists();
        break;
      case THRESHOLD:
        truth = threshold > length();
        break;
    }
    return truth;
  }

  void performAction() {
    Date time = Calendar.getInstance().getTime();
    String timeStamp = time.toString();
```

(continues)

Figure 8.2 Continued.

```
    switch(action) {
      case ALERT:
        trace(timeStamp + " " + name + ": Alert fired \n");
        ((AlertDialog) actionDialog).appendMsgText(timeStamp + " - "
            + parms);
        actionDialog.show();
        break;
      case EXECUTE:
        trace(name + ": Executing command \n");
        executeCmd(parms);
        break;
      case EVENT:
        notifyCIAgentEventListeners(new CIAgentEvent(this,
          actionString, "Watch condition on " + fileName + " was
          triggered!"));
        break;
    }
  }

  public int executeCmd(String cmd) {
    Process process = null;
    String line;
    String osType = (System.getProperty("os.name")).toUpperCase();

    trace(cmd);
    actionDialog.show();
    try {
      if(osType.equals("WINDOWS 95")) {
        process = Runtime.getRuntime().exec("command.com /c " + cmd
            + "\n");
        BufferedReader data = new BufferedReader(
          new InputStreamReader(process.getInputStream()));

        while((line = data.readLine()) != null) {
          trace(line);
        }
        data.close();
      } else if(osType.equals("AIX") || osType.equals("UNIX")) {
        process = Runtime.getRuntime().exec(cmd + "\n");
        BufferedReader data = new BufferedReader(
          new InputStreamReader(process.getInputStream()));

        while((line = data.readLine()) != null) {
          trace(line);
        }
        data.close();
      } else if(osType.equals("WINDOWS NT")) {
```

Figure 8.2 The FileAgent class listing (Continued).

```
        process = Runtime.getRuntime().exec("cmd /C " + cmd + "\n");
        BufferedReader data = new BufferedReader(
          new InputStreamReader(process.getInputStream()));

        while((line = data.readLine()) != null) {
          trace(line);
        }
        data.close();
      } else if(osType.equals("OS/2")) {
        process = Runtime.getRuntime().exec("cmd.exe /c " + cmd +
          "\n");
        BufferedReader data = new BufferedReader(
          new InputStreamReader(process.getInputStream()));

        while((line = data.readLine()) != null) {
          trace(line);
        }
        data.close();
      } else {
        trace("FileAgent Error -- unsupported OS or run-time
          environment");
      }
    } catch(IOException err) {
      trace("Error: EXEC failed, " + err.toString());
      err.printStackTrace();
      return -1;
    }
    if(((ExecuteDialog) actionDialog).getCancel() == true) {
      stopAgentProcessing();
    }
    if(process != null) {
      return process.exitValue();
    } else {
      return -1;
    }
  }

  protected boolean exists() {
    return file.exists();
  }

  protected boolean changed() {
    long changeTime = lastChanged;

    lastChanged = file.lastModified();
    return !(lastChanged == changeTime);
```

(continues)

Figure 8.2 Continued.

```
  }

  protected long length() {
    return file.length();
  }

  protected long lastModified() {
    return file.lastModified();
  }
}
```

Figure 8.2 The FileAgent class listing (Continued).

The *actionDialog* is used to display information for the user when the ALERT or EXECUTE action has been chosen. The *parms* member is also used when executing a command. The *actionString* is used as the action parameter on the **CIAgentEvent** created if the EVENT action was chosen.

The *initialize()* method sets up the dialogs for the ALERT or EXECUTE actions. It also sets the sleep timer for the agent to 15 seconds and sets the agent's state to INITIATED. Whenever the sleep timer pops or the agent is called synchronously, the *process()* method gets called and the file *condition* is tested. If the condition is met, then the specified *action* is performed.

In addition to the **FileAgent** class, **FileAgentCustomizer** and **FileAgentBeanInfo** classes are provided for the **FileAgent**. The **FileAgentCustomizer** is shown in Figure 8.3. These classes will be used by the **PAManagerFrame** when instantiating a **FileAgent** bean. Before running the **FileAgent**, let's take a closer look at the *PAManager* platform, where the **FileAgent** will run.

The PAManagerFrame

The **PAManagerApp** instantiates a **PAManagerFrame** object that does the majority of the work in the application. When the **PAManagerFrame** is instantiated, the constructor initializes the GUI controls and then calls the *readPropertiesFile()* method to get the class names of all the **CIAgents** that can run on the PAManager platform. The properties file is called *pamanager.properties* and must be located in the directory from which the **pamanager.PAManagerApp** is run.

```
private void readPropertiesFile() {
  Properties properties = new Properties();

  try {
    FileInputStream in = new FileInputStream("pamanager.properties");
```

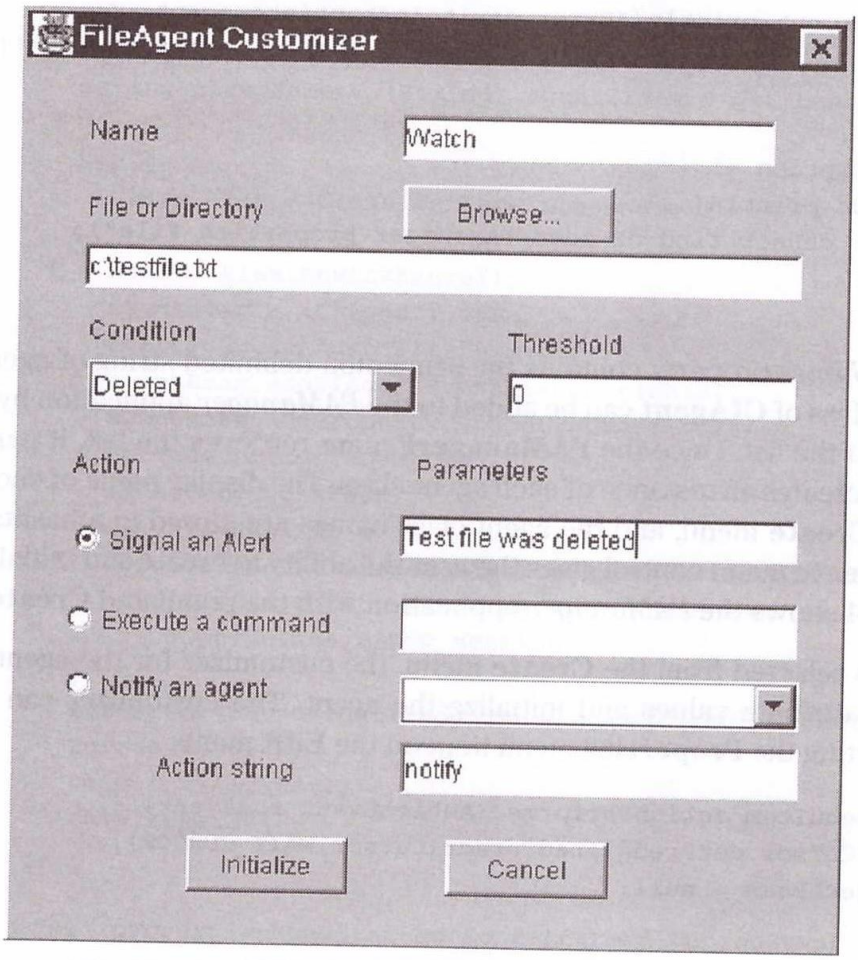


Figure 8.3 The FileAgentCustomizer window.

```
properties.load(new BufferedInputStream(in));
String property;

property = properties.getProperty("AgentClassNames");
StringTokenizer tok = new StringTokenizer(property, ";");
String displayName;

while(tok.hasMoreTokens()) {
  String agentClassName = tok.nextToken();
  CIAgent agent = null;

  try {
    Class klas = Class.forName(agentClassName);

    agent = (CIAgent) klas.newInstance();
    displayName = agent.getDisplayName();
    System.out.println("Adding agent class ... " + displayName);
    addAgentMenuItem(displayName);
    agentClasses.put(displayName, agentClassName);
  } catch(Exception exc) {
```

```
            System.out.println("Error can't instantiate agent: "
                               + agentClassName + " " + exc.toString());
          }
        }
      } catch(Exception e) {
        System.out.println(
          "Error: cannot find or load PAManager properties file");
      }
    }
```

The *AgentClassNames* property contains the semicolon-delimited string of agent class names. Any subclass of **CIAgent** can be added to the *PAManager* application by adding the class name to the list. Once the **PAManagerFrame** retrieves the list, it parses the class names and creates an instance of each agent class. The display name of each agent is added to the **Create** menu, and the agent class names are stored in a hashtable for later use. The **Create** menu control gives the user the ability to create and initialize new agents. Figure 8.4 shows the *PAManager* application with the populated **Create** menu.

Once an agent is selected from the **Create** menu, the customizer for the agent is then used to set the attribute values and initialize the agent. The customizer can also be invoked by selecting the **Properties** menu item on the **Edit** menu.

```
void CreateMenuItem_actionPerformed(ActionEvent theEvent) {
  setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
  CIAgent agentBean = null;
```

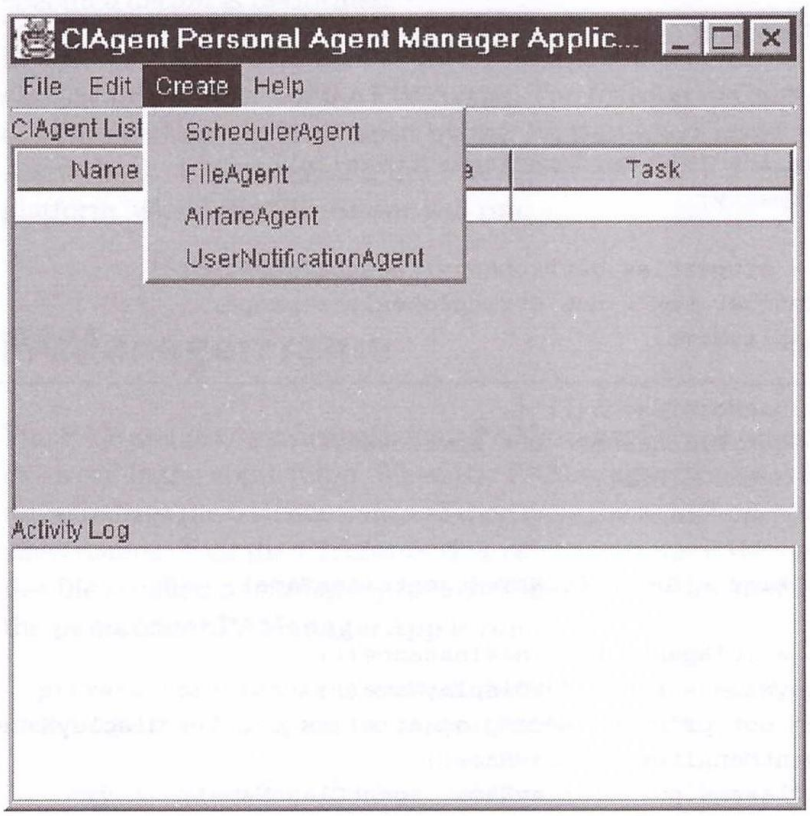


Figure 8.4 The PAManager Create menu.

```
    Object bean = null;
    String beanName = theEvent.getActionCommand();
    String className = (String) agentClasses.get(beanName);

    try {
      Class klas = Class.forName(className);

      bean = klas.newInstance();
      agentBean = (CIAgent) bean;
      agentBean.setAgentPlatform(this);
      agentBean.addCIAgentEventListener(this);
      agentBean.addPropertyChangeListener(this);
      openCustomizer(agentBean, true);
      addAgent(agentBean);
    } catch(Exception e) {
      JOptionPane.showMessageDialog(this, e.toString(),
          "Error: Can't create agent " + beanName,
          JOptionPane.ERROR_MESSAGE);
    }
    setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
    refreshTable();
    this.invalidate();
    this.repaint();
  }

  void propertiesMenuItem_actionPerformed(ActionEvent e) {
    int selectedRow = agentTable.getSelectedRow();

    if((selectedRow < 0) || (selectedRow >= agents.size())) {
      return;
    }
    CIAgent agent = (CIAgent) agents.elementAt(selectedRow);

    openCustomizer(agent, false);
  }
```

Once an agent has been created, it shows up in the **CIAgent** list in the *PAManager* application window. For each agent, the name, type of agent, state, and task are listed. You can select an agent by clicking on the row that contains the information for the agent. Once an agent is selected, the **Edit** menu can be used to change the state of the agent. Figure 8.5 shows the *PAManager* application populated with agents in various states.

When an agent has been initialized, its state changes to INITIATED, and the **Start processing** selection is available on the **Edit** menu. The **Start processing** menu item is used to activate an agent by calling the *startAgentProcessing()* method on the selected **CIAgent**. The agent's state is now ACTIVE, and that is reflected in the *PAManager* window. Once an agent is active, its timer is running and it is processing timer pops and events.

```
  void startProcessingMenuItem_actionPerformed(ActionEvent e) {
    int selectedRow = agentTable.getSelectedRow();
```

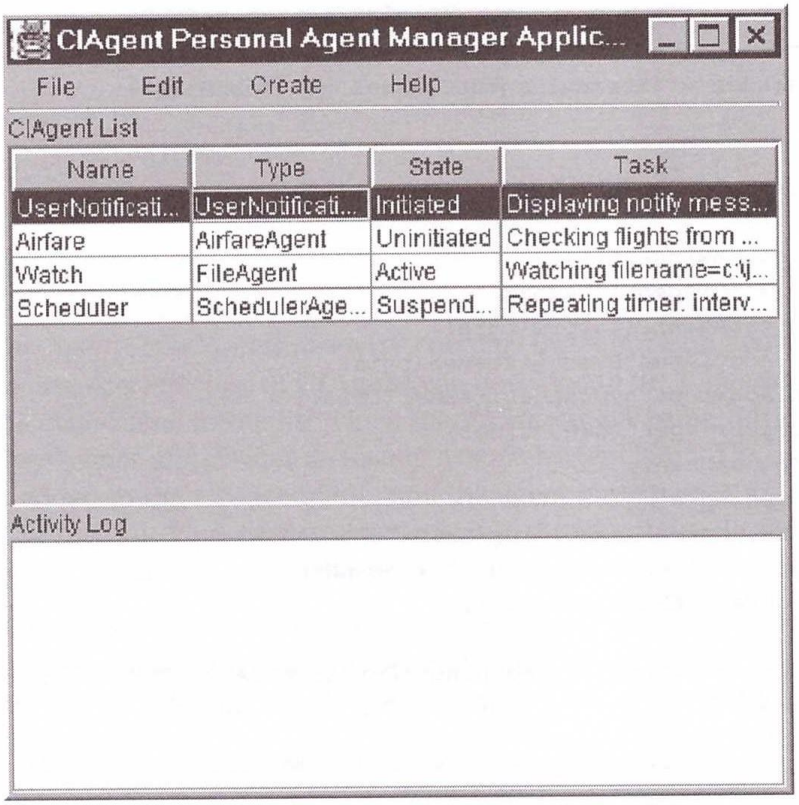


Figure 8.5 The *PAManager* application window.

```
    if((selectedRow < 0) || (selectedRow >= agents.size())) {
      return;
    }
    CIAgent agent = (CIAgent) agents.elementAt(selectedRow);

    if(agent.getState().getState() == CIAgentState.INITIATED) {
      agent.startAgentProcessing();
      setEditMenuItemStates();
      updateTable();
    }
  }
```

An agent's processing can be suspended using the **Suspend processing** menu selection and resumed using the **Resume processing** menu selection. These actions call the corresponding *suspendAgentProcessing()* and *resumeAgentProcessing()* **CIAgent** methods.

```
  void suspendProcessingMenuItem_actionPerformed(ActionEvent e) {
    int selectedRow = agentTable.getSelectedRow();

    if((selectedRow < 0) || (selectedRow >= agents.size())) {
      return;
    }
```

```
    CIAgent agent = (CIAgent) agents.elementAt(selectedRow);

    agent.suspendAgentProcessing();
    setEditMenuItemStates();
    updateTable();
  }

  void resumeProcessingMenuItem_actionPerformed(ActionEvent e) {
    int selectedRow = agentTable.getSelectedRow();

    if((selectedRow < 0) || (selectedRow >= agents.size())) {
      return;
    }
    CIAgent agent = (CIAgent) agents.elementAt(selectedRow);

    agent.resumeAgentProcessing();
    setEditMenuItemStates();
    updateTable();
  }
```

The **Cut** menu selection is used to stop an agent by calling the *stopAgentProcessing()* method on the selected agent. It also removes the agent from the platform. The **Clear** menu item in the **File** menu can be used to stop all agents and remove them from the platform.

```
  void Cut_actionPerformed(ActionEvent e) {
    int selectedRow = agentTable.getSelectedRow();

    if((selectedRow < 0) || (selectedRow >= agents.size())) {
      return;
    }
    CIAgent agent = (CIAgent) agents.elementAt(selectedRow);

    agent.stopAgentProcessing();
    agents.removeElementAt(selectedRow);
    refreshTable();
    setEditMenuItemStates();
  }

  void clearMenuItem_actionPerformed(ActionEvent e) {
    int size = agents.size();

    for(int i = 0; i < size; i++) {
      CIAgent agent = (CIAgent) agents.elementAt(0);

      agent.stopAgentProcessing();
      agents.removeElementAt(0);
    }
    refreshTable();
    traceTextArea.setText("");
  }
```

In addition to managing the user interface and the agent lifecycles, the **PAManagerFrame** implements the **CIAgentEventListener** interface and adds itself as a listener to every agent it creates. This enables the **PAManagerFrame** to receive **CIAgentEvents** which are processed by its *processCIAgentEvent()* method. All **CIAgentEvents** are processed by displaying the event information in the text area within the frame.

```
public void processCIAgentEvent(CIAgentEvent event) {
  Object source = event.getSource();
  String agentName = "";

  if(source instanceof CIAgent) {
    agentName = ((CIAgent) source).getName();
  }
  Object arg = event.getArgObject();
  Object action = event.getAction();

  if(action != null) {
    if(action.equals("trace")) {
      if(((arg != null) && (arg instanceof String))) {
        trace("\n" + (String) arg);
      }
    } else {
      trace("\nPAManager received action event: " + action
            + " from agent " + agentName);
    }
  }
}
```

FileAgent Example

The basic functions provided by the **FileAgent** and the *PAManager* application can be illustrated using a simple example. We will use the **FileAgent** to monitor a file called *testfile.txt* and send an alert message when the file is deleted. The scenario is as follows:

1. Create a file called *testfile.txt* on the local file system.
2. Run the *PAManager* application: *java pamanager.PAManager*.
3. Select **FileAgent** from the **Create** menu. Fill in the customizer as shown in Figure 8.2. Be sure to specify the correct path and filename, depending on the operating system on which you are running this application. Click on the **Initialize** button.
4. Select *Watch* **FileAgent** from the **CIAgent** list in the *PAManager* application window. Select the **Start processing** option in the **Edit** menu.
5. Outside of the *PAManager* application, delete the *testfile.txt* that you created in Step 1.

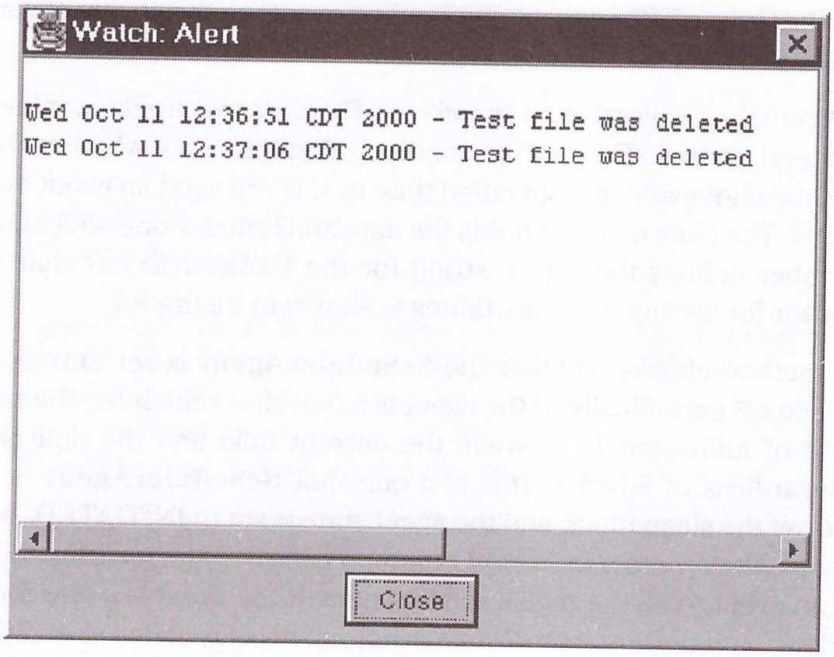


Figure 8.6 The AlertDialog window.

6. Once the **FileAgent**'s timer pops, an **AlertDialog** window will appear and display the message that the file was deleted. The window will look similar to what is shown in Figure 8.6. Note that the agent will send an alert message every 15 seconds, once the file is deleted. Note, too, that trace messages are appearing in the text area at the bottom of the Personal Agent Manager Application window.
7. Use the **Cut** selection on the **Edit** menu to stop the *Watch* **FileAgent** and remove it from the application.
8. At this point you can either leave the *PAManager* running to use later in this chapter, or exit the *PAManager* application by choosing the **Exit** option from the **File** menu.

Now that you've seen the **FileAgent** and the *PAManager* application in action, let's take a look at a few other agents you can run on this platform.

The SchedulerAgent

Our next agent, the **SchedulerAgent**, is not very intelligent but is certainly autonomous and provides an extremely useful function. A **SchedulerAgent** can be used to set one-shot alarms that go off at a specified time, or it can be used for recurrent alarms at specified intervals. When an alarm condition occurs, the **SchedulerAgent** sends a **CIAgentEvent** to all of its registered listeners. In effect, the **SchedulerAgent** can be

used to schedule the tasks of other agents by sending events to the agents at a specified time or after a specified interval of time has passed.

The **SchedulerAgent** has several data members. The *interval* holds a value, in milliseconds, for interval alarms. The *oneShot* boolean flag indicates whether this agent will send a single one-shot event at a specified time or if it will send an event every time the *interval* expires. The *time* member holds the date and time for one-shot alarms. The *actionString* member defines the action string for the **CIAgentEvent** that is generated. The customizer for setting these attributes is shown in Figure 8.7.

The *initialize()* method checks whether the **SchedulerAgent** is set to trigger a one-shot alarm or will go off periodically. If the agent is a one-shot scheduler, the *interval* is set to the number of milliseconds between the current time and the time the alarm should go off. Regardless of whether this is a one-shot **SchedulerAgent** or not, the interval is used to set the sleep timer, and the agent state is set to INITIATED. Whenever the sleep timer pops, the *processTimerPop()* method calls the *notifyCIAgentEventListeners()* to send an event to all the registered listeners. If the agent is a one-shot scheduler, its work is done, and the *stopAgentProcessing()* method is called to end the agent. Note that the *process()* and *processCIAgentEvents()* methods are not used in this agent. A listing of the **SchedulerAgent** class is shown in Figure 8.8.

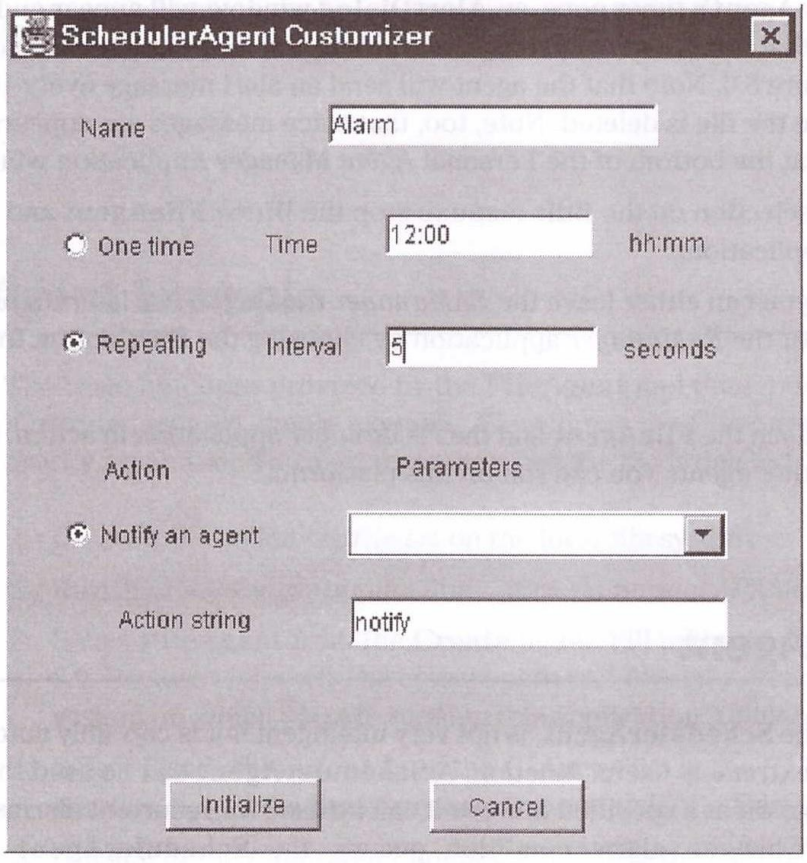


Figure 8.7 The SchedulerAgentCustomizer window.

```
package pamanager;

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

public class SchedulerAgent extends CIAgent implements Serializable {
  protected int interval = 60000;
  protected boolean oneShot = false;
  protected Date time = null;
  protected String actionString = "notify";

  public SchedulerAgent() {
    this("Scheduler");
  }

  public SchedulerAgent(String name) {
    super(name);
  }

  public String getTaskDescription() {
    if(oneShot) {
      return "One-shot timer: time = " + time.toString();
    } else {
      return "Repeating timer: interval = " + (interval / 1000);
    }
  }

  public void setInterval(int secs) {
    interval = secs * 1000;
  }

  public int getInterval() {
    return interval / 1000;
  }

  public void setOneShot(boolean flag) {
    oneShot = flag;
  }

  public boolean getOneShot() {
    return oneShot;
  }
```

(continues)

Figure 8.8 The SchedulerAgent class listing.

```
public void setTime(Date date) {
  time = date;
}

public Date getTime() {
  return time;
}

public void setActionString(String actionString) {
  this.actionString = actionString;
}

public String getActionString() {
  return actionString;
}

public void initialize() {
  if(oneShot) {
    long currentTime = Calendar.getInstance().getTime().getTime();

    interval = (int) (time.getTime() - currentTime);
  }
  setSleepTime(interval);
  setAsyncTime(interval);
  setState(CIAgentState.INITIATED);
}

public void process() {}

public void processTimerPop() {
  String timeStamp = Calendar.getInstance().getTime().toString();

  notifyCIAgentEventListeners(new CIAgentEvent(this, actionString,
      timeStamp));
  if(oneShot) {
    stopAgentProcessing();
  }
}

public void processCIAgentEvent(CIAgentEvent e) {}
}
```

Figure 8.8 The SchedulerAgent class listing (Continued).

The **SchedulerAgent**, while performing a useful task, needs to cooperate with other agents since sending an event serves no useful purpose unless another object is waiting to catch and process the event. Fortunately, the *PAManager* platform listens for events

from any agent on the platform. We can use this to illustrate the function of the **SchedulerAgent**. The scenario is as follows:

1. Run the**PAManagerApp**, if it is not already running.
2. Select **SchedulerAgent** from the **Create** menu. Fill in the customizer as shown in Figure 8.7. Click on the **Initialize** button.
3. Select the *Alarm* **SchedulerAgent** from the **CIAgent** list in the *PAManager* application window. Select the **Start processing** option in the **Edit** menu.
4. Every 5 seconds, when the **SchedulerAgent**'s timer pops, an event will be displayed in the text area at the bottom of the Personal Agent Manager Application window.
5. Use the **Cut** selection on the **Edit** menu to stop the *Alarm* **SchedulerAgent** and remove it from the application.
6. At this point you can either leave the *PAManager* running to use later in this chapter, or exit the *PAManager* application by choosing the **Exit** option from the **File** menu.

The **PAManagerApp**, as shown in Figure 8.9, is useful for testing the **SchedulerAgent**, but a more realistic scenario would be for the **SchedulerAgent** to notify another agent when its timer pops. Earlier we looked at the **FileAgent** which had the ability to alert the user by displaying a message when a certain file condition occurred. It would be nice if the **SchedulerAgent** had this same capability. In fact, many agents that we develop may need the ability to notify the user. Rather than duplicate this function in all of our

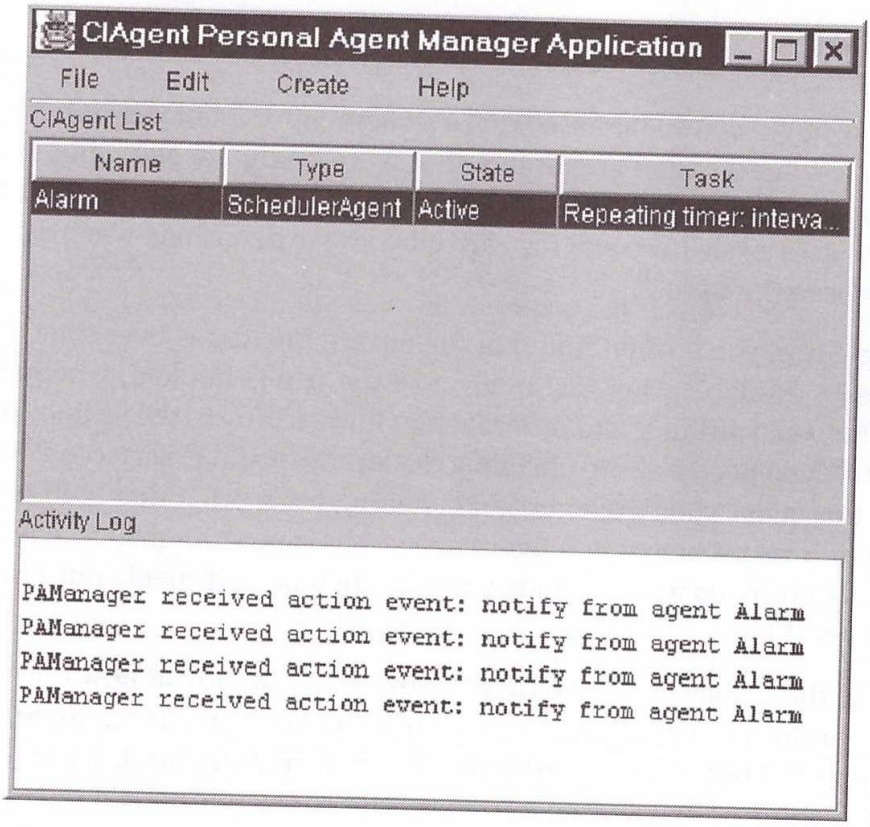


Figure 8.9 The SchedulerAgent example.

agents, we develop one agent, the **UserNotificationAgent**, whose primary function is to notify the user when it receives an event.

The UserNotificationAgent

The **UserNotificationAgent** is a very simple agent, as shown in Figure 8.10. It contains only one unique member, the dialog in which the notification messages are displayed. Its *initialize()* method creates this dialog, using the same **AlertDialog** class we used in the **FileAgent**. The sleep timer is set to five seconds, and the state is set to INITIATED. Once the **UserNotificationAgent** is started, its *processCIAgentEvent()* method is called to process any events it receives. Note that it does not process timer pops or do any synchronous processing. When an event is received, the event action is checked to see if it is a "notify" event. If it is, the message text is retrieved from the event and displayed.

Even though the **UserNotificationAgent** is a very simple agent, it is quite useful. It provides a centralized user interface for the display of notification messages and can be used by multiple agents in a system. This prevents the duplication of function and code as well as provides a nicer user interface, because all messages are displayed in a single window for the user. We are not going to show an example of the **UserNotificationAgent** in action until we demonstrate the final **CIAgent** in this chapter, the **AirfareAgent**.

The AirfareAgent

All of the agents we have seen so far have been autonomous, but not very intelligent. The **FileAgent** has a few hard-coded conditions that were being checked, but it was not using any rules or inferencing. The final agent we will examine in this chapter is the **AirfareAgent**, which uses a rule base and forward chaining to determine when a published airfare is of interest to the user.

When scheduling a trip, you'll often find that the airfare fluctuates over time, based on the number of seats available, how far in advance the trip is booked, whether a "fare war" is going on between airlines, and other factors known only to the airline. To get the best fares available, you need to keep checking the airline or travel services Web site for flight schedules that meet your travel criteria, waiting for a price that is in your price range. In this section we create an intelligent agent that is programmed to check a Web site for us and only notify us when it finds a flight schedule that meets our criteria at a price we find acceptable.

Before we get into the details of the **AirfareAgent**, let's first look at how one would go about getting the information without the use of an agent. For this example, we have created a Web page at http://www.bigusbooks.com/airfareQuery.html. When you bring this page up in a Web browser, you see a form similar to the one shown in Figure 8.11.

In our example, we want to fly from Rochester, MN to Orlando, FL on July 1st and return on July 8th. After entering the travel dates and the originating and destination cities in

```
package pamanager;

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

public class UserNotificationAgent extends CIAgent
    implements Serializable {
  protected AlertDialog notificationDialog;

  public UserNotificationAgent() {
    name = "UserNotification";
  }

  public UserNotificationAgent(String name) {
    super(name);
  }

  public String getTaskDescription() {
    return "Displaying notify messages";
  }

  public String getMsgText() {
    return notificationDialog.getMsgText();
  }

  public void setMsgText(String text) {
    notificationDialog.setMsgText(text);
  }

  public void appendMsgText(String text) {
    notificationDialog.appendMsgText(text);
  }

  public void setDialog(JDialog dlg) {
    notificationDialog = (AlertDialog) dlg;
  }

  public void initialize() {
    if(notificationDialog != null) {
      notificationDialog.dispose();
    }
    JFrame frame = new JFrame();
```

(continues)

Figure 8.10 The UserNotificationAgent class listing.

```
    notificationDialog = new AlertDialog(frame,
        "CIAgent User Notification Agent: " + name, false);
    setSleepTime(5 * 1000);
    setState(CIAgentState.INITIATED);
  }

  public void process() {}

  public void processCIAgentEvent(CIAgentEvent e) {
    if(e.getAction().equalsIgnoreCase("notify")) {
      Date time = Calendar.getInstance().getTime();
      String timeStamp = time.toString();
      String text = (String) e.getArgObject();
      String source = e.getSource().getClass().toString();

      if(e.getSource() instanceof CIAgent) {
        source = ((CIAgent) e.getSource()).getName();
      }
      notificationDialog.appendMsgText("Event received on " +
        timeStamp
          + " from " + source + ":\n" + text);
      if(!notificationDialog.isVisible()) {
        notificationDialog.show();
      }
    }
  }

  public void processTimerPop() {}
}
```

Figure 8.10 The UserNotificationAgent class listing (Continued).

the form, pressing the **Get Airfare** button would send the entries to the Web server for processing. In this case, the data is sent to a CGI-bin program that processes the request and returns the results. An example results page is shown in Figure 8.12.

Note that the URL displayed by the Web browser contains the name of the CGI-bin program we are invoking and the parameters with the travel information we entered. In our example, the full URL string is http://www.bigusbooks.com/cgi-local/airfare.pl?&dmon=JUL&dday=1&orig=RST&dest=MCO&rmon=JUL&rday=8. In order to minimize network traffic, the HTML code shortens the months to three character abbreviations (JUL for July) and the cities to three character airport codes (MCO for Orlando).

Let's further suppose that it is our desire to fly from Rochester in the evening and catch our return flight either in the morning or the afternoon. We are willing to pay up to $1000.00 for our ideal flight times.

But often, when scheduling a trip, we're willing to settle for less than ideal if the price is right. We may take a flight that departs in the morning and returns in the afternoon, if

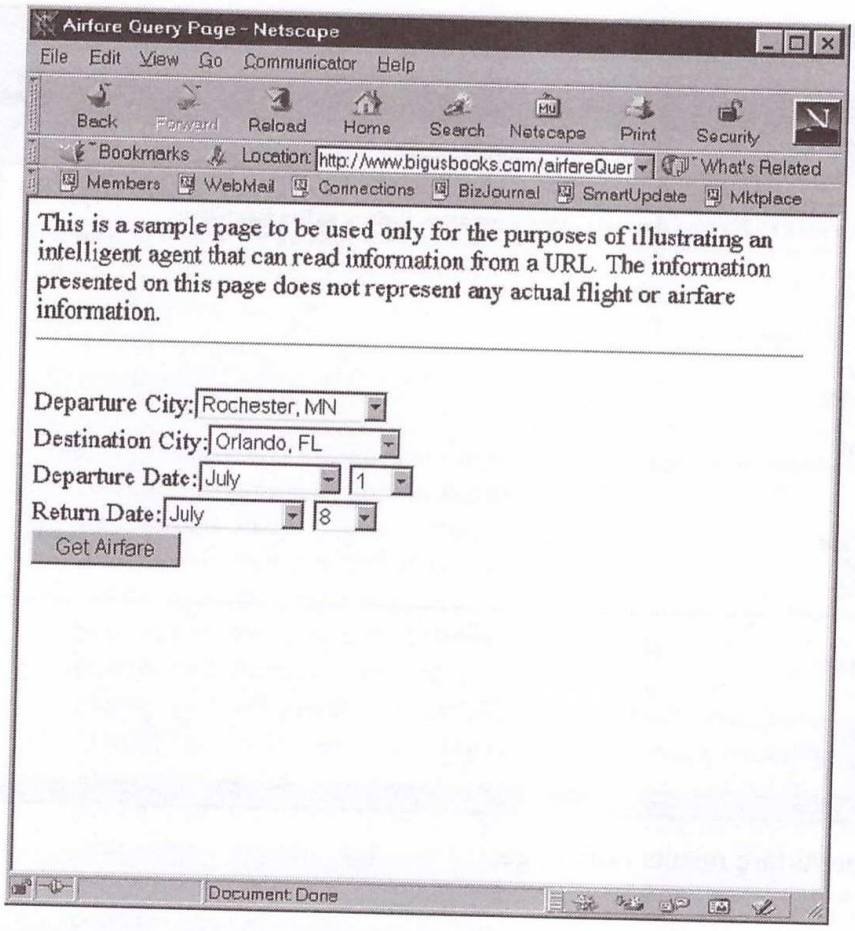


Figure 8.11 The Airfare query html page.

the price is less than $800.00. Or we may be willing to leave in the morning and return in the evening if the price is less than $600.00. Without using an agent, we would periodically go out to the Web site, enter in our dates and cities, and check the flights that were returned to see if any match our flight schedule and pricing criteria. At a time when prices are changing rapidly, as is often the case when a fare war is being waged, we may need to repeat this process quite often in order to get the right flights at the right price. Wouldn't it be nice to have an agent that could check the flights for us and only notify us when it finds something in our price range? The **AirfareAgent** does just that.

The **AirfareAgent** has a number of data members that are used to hold the query parameters sent to the Web page. They are the *departMonth*, *departDay*, *origCity*, *destCity*, *returnMonth*, and *returnDay*. A **String** data member, *actionString*, contains the action for the **CIAgentEvent** sent when a desirable travel itinerary is found. The customizer for setting these attributes is shown in Figure 8.13.

In addition to the attributes set in the customizer, the **AirfareAgent** contains a **BooleanRuleBase**, *rb* , with rules that specify the desired flight times and prices. A few **RuleVariables** (*departs*, *returns*, and *price*) are used to hold time and price information during inferencing.

As shown in the listing in Figure 8.14, the *initialize()* method in the **AirfareAgent** sets up the flight rule base by calling the *initFlightRuleBase()* method. The sleep timer is

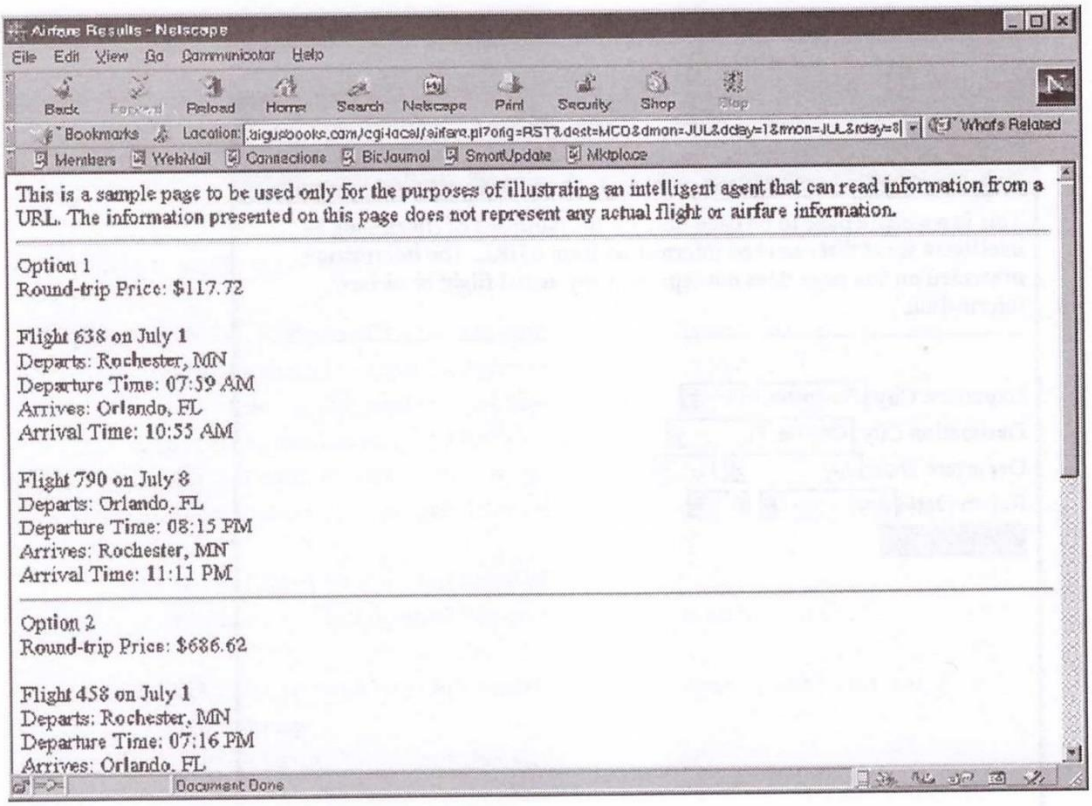


Figure 8.12 The Airfare results html page.

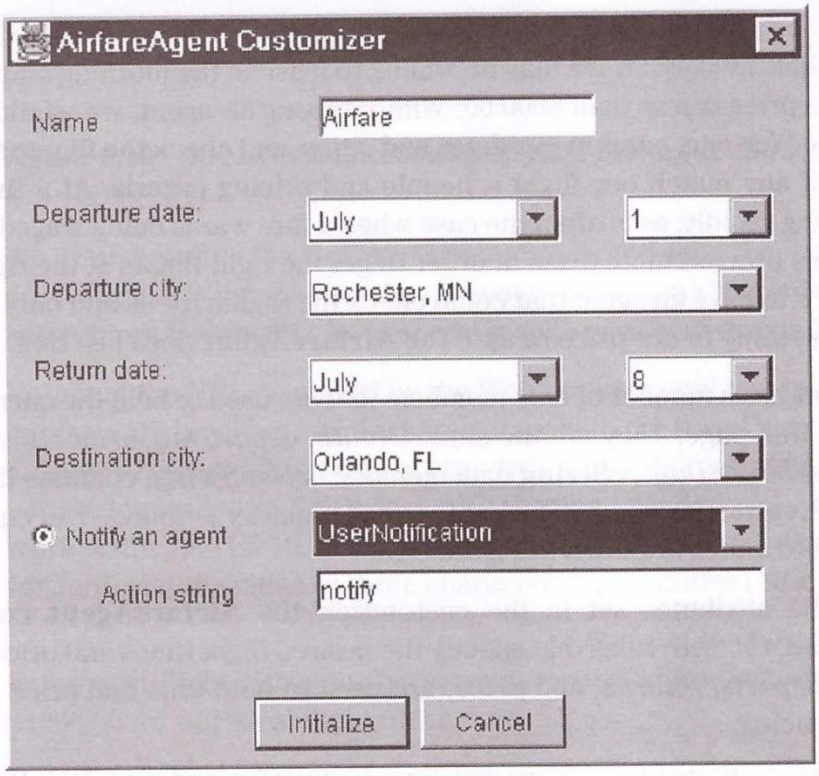


Figure 8.13 The AirfareAgentCustomizer window.

```
package pamanager;

import java.awt.*;
import javax.swing.*;
import java.io.*;
import java.util.*;
import java.text.*;
import java.net.*;
import ciagent.*;
import rule.*;

public class AirfareAgent extends CIAgent implements Serializable {
  protected String departMonth = "JAN";
  protected String departDay = "1";
  protected String origCity = "RST";
  protected String destCity = "MCO";
  protected String returnMonth = "FEB";
  protected String returnDay = "2";
  protected BooleanRuleBase rb = new BooleanRuleBase("Flight");
  protected RuleVariable departs = new RuleVariable(rb, "departs");
  protected RuleVariable returns = new RuleVariable(rb, "returns");
  protected RuleVariable price = new RuleVariable(rb, "price");
  protected String actionString = "notify";

  public AirfareAgent() {
    name = "Airfare";
  }

  public AirfareAgent(String name) {
    super(name);
  }

  public void setDepartMonth(String departMonth) {
    this.departMonth = departMonth;
  }

  public String getDepartMonth() {
    return departMonth;
  }

  public void setDepartDay(String departDay) {
    this.departDay = departDay;
  }

  public String getDepartDay() {
    return departDay;
```

(continues)

Figure 8.14 The AirfareAgent class listing.

```
}

public void setReturnMonth(String returnMonth) {
  this.returnMonth = returnMonth;
}

public String getReturnMonth() {
  return returnMonth;
}

public void setReturnDay(String returnDay) {
  this.returnDay = returnDay;
}

public String getReturnDay() {
  return returnDay;
}

public void setOrigCity(String origCity) {
  this.origCity = origCity;
}

public String getOrigCity() {
  return origCity;
}

public void setDestCity(String destCity) {
  this.destCity = destCity;
}

public String getDestCity() {
  return destCity;
}

public void setActionString(String actionString) {
  this.actionString = actionString;
}

public String getActionString() {
  return actionString;
}

public String getTaskDescription() {
  return "Checking flights from " + origCity + " to " + destCity;
}

public void initialize() {
```

Figure 8.14 The AirfareAgent class listing (Continued).

```
    initFlightRuleBase();
    setSleepTime(30000);
    setState(CIAgentState.INITIATED);
  }

  public void process() {
    URL url;
    String parms = "?" + "&dmon=" + departMonth
                       + "&dday=" + departDay
                       + "&orig=" + origCity
                       + "&dest=" + destCity
                       + "&rmon=" + returnMonth
                       + "&rday=" + returnDay;
    HttpURLConnection connection;
    String rank = null;

    try {
      url =
        new URL("http://www.bigusbooks.com/cgi-local/airfare.pl");
      connection = (HttpURLConnection) url.openConnection();
      connection.setDoOutput(true);
      PrintWriter out =
        new PrintWriter(connection.getOutputStream());

      out.println(parms);
      out.close();
      BufferedReader in = new BufferedReader(
        new InputStreamReader(connection.getInputStream()));
      String inputLine;
      int i, j = 0, k = 0;
      int x, y;

      i = 0;
      String prices[] = new String[3];
      String flight[][] = new String[3][2];
      String times[][] = new String[3][2];

      while((inputLine = in.readLine()) != null) {
        x = inputLine.indexOf("Price:");
        if(x != -1) {
          y = inputLine.indexOf('.', x);
          prices[i] = inputLine.substring(x + 8, y + 3);
          i++;
          j = k = 0;
        }
        x = inputLine.indexOf("Flight");
```

(continues)

Figure 8.14 Continued.

```
    if(x != -1) {
      flight[i - 1][j] = inputLine.substring(x);
      j++;
    }
    x = inputLine.indexOf("Departure Time:");
    if(x != -1) {
      y = inputLine.lastIndexOf(':');
      String hours = inputLine.substring(x + 16, y);

      if(inputLine.indexOf("PM") != -1) {
        int hrs = Integer.parseInt(hours) % 12 + 12;

        hours = Integer.toString(hrs);
      }
      times[i - 1][k] = hours + inputLine.substring(y + 1, y + 3);
      k++;
    }
  }
  in.close();
  for(i = 0; i < 3; ++i) {
    rb.reset();
    trace("Checking departure at " + times[i][0] + " return at "
          + times[i][1] + " for a price of " + prices[i]);
    departs.setValue(times[i][0]);
    returns.setValue(times[i][1]);
    price.setValue(prices[i]);
    rb.forwardChain();
    rank = rb.getVariable("flightRank").getValue();
    if(rank != null) {
      String msg = "An iterary was found that meets your \""
                   + rank
                   + "\" flight criteria.\nDeparture from "
                   + origCity
                   + ": " + flight[i][0] + " departs at "
                   + times[i][0].substring(0, 2) + ":"
                   + times[i][0].substring(2)
                   + "\nReturn from "
                   + destCity + ": " + flight[i][1]
                   + " departs at "
                   + times[i][1].substring(0, 2) + ":"
                   + times[i][1].substring(2) + "\nPrice: $"
                   + prices[i];

      notifyCIAgentEventListeners(
        new CIAgentEvent(this, actionString, msg));
    }
  }
```

Figure 8.14 The AirfareAgent class listing (Continued).

```
    } catch(Exception e) {
      String msg =
        "Problems retrieving information from the web site.\n"
        + "Please make sure that you are connected to the Internet\n"
        + "and that the web site is up. To do this, use your
           browser \n"
        + "to go to www.bigusbooks.com/aifareQuery.html and enter
           the\n"
        + "same dates and cities. If it doesn't work, the server \n"
        + "must be having problems and you can try again later.";

      notifyCIAgentEventListeners(new CIAgentEvent(this, actionString,
          msg));
      e.printStackTrace();
    }
  }

  public void processCIAgentEvent(CIAgentEvent e) {
    if(e.getAction().equalsIgnoreCase("process")) {
      process();
    }
  }

  public void processTimerPop() {}

  public void initFlightRuleBase() {
    RuleVariable departureTime =
      new RuleVariable(rb, "departureTime");

    departureTime.setLabels("morning afternoon evening");
    RuleVariable desiredDeparture =
      new RuleVariable(rb, "desiredDeparture");

    desiredDeparture.setLabels("yes no");
    RuleVariable returnTime = new RuleVariable(rb, "returnTime");

    returnTime.setLabels("morning afternoon evening");
    RuleVariable desiredReturn =
      new RuleVariable(rb, "desiredReturn");

    desiredReturn.setLabels("yes no");
    RuleVariable flightRank = new RuleVariable(rb, "flightRank");

    flightRank.setLabels("good better best");
    Condition cEquals = new Condition("=");
```

(continues)

Figure 8.14 Continued.

```
Condition cNotEquals = new Condition("!=");
Condition cGreaterThan = new Condition(">");
Condition cLessThan = new Condition("<");
Rule morningDeparture =
  new Rule(rb, "morningDeparture",
    new Clause(departs, cLessThan, "1200"),
    new Clause(departureTime, cEquals, "morning"));
Rule afternoonDeparture =
  new Rule(rb, "afternoonDeparture",
    new Clause[]{
      new Clause(departs, cGreaterThan, "1159"),
      new Clause(departs, cLessThan, "1700") },
    new Clause(departureTime, cEquals, "afternoon"));
Rule eveningDeparture =
  new Rule(rb, "eveningDeparture",
    new Clause(departs, cGreaterThan, "1659"),
    new Clause(departureTime, cEquals, "evening"));
Rule morningReturn =
  new Rule(rb, "morningReturn",
    new Clause(returns, cLessThan, "1200"),
    new Clause(returnTime, cEquals, "morning"));
Rule afternoonReturn =
  new Rule(rb, "afternoonReturn",
    new Clause[]{
      new Clause(returns, cGreaterThan, "1159"),
      new Clause(returns, cLessThan, "1700") },
    new Clause(returnTime, cEquals, "afternoon"));
Rule eveningReturn =
  new Rule(rb, "eveningReturn",
    new Clause(returns, cGreaterThan, "1659"),
    new Clause(returnTime, cEquals, "evening"));
Rule desireableDeparture =
  new Rule(rb, "desirableDeparture",
    new Clause(departureTime, cEquals, "evening"),
    new Clause(desiredDeparture, cEquals, "yes"));
Rule undesirableDeparture1 =
  new Rule(rb, "undesirableDeparture1",
    new Clause(departureTime, cEquals, "morning"),
    new Clause(desiredDeparture, cEquals, "no"));
Rule undesirableDeparture2 =
  new Rule(rb, "undesirableDeparture2",
    new Clause(departureTime, cEquals, "afternoon"),
    new Clause(desiredDeparture, cEquals, "no"));
Rule undesirableReturn =
  new Rule(rb, "undesirableReturn",
    new Clause(returnTime, cEquals, "evening"),
    new Clause(desiredReturn, cEquals, "no"));
```

Figure 8.14 The AirfareAgent class listing (Continued).

```
        Rule desirableReturn1 =
          new Rule(rb, "desirableReturn1",
            new Clause(returnTime, cEquals, "morning"),
            new Clause(desiredReturn, cEquals, "yes"));
        Rule desirableReturn2 =
          new Rule(rb, "desirableReturn2",
            new Clause(returnTime, cEquals, "afternoon"),
            new Clause(desiredReturn, cEquals, "yes"));
        Rule bestFlight =
          new Rule(rb, "bestFlight",
            new Clause[]{
              new Clause(desiredDeparture, cEquals, "yes"),
              new Clause(desiredReturn, cEquals, "yes"),
              new Clause(price, cLessThan, "1000.00") },
            new Clause(flightRank, cEquals, "best"));
        Rule betterFlight1 =
          new Rule(rb, "betterFlight1",
            new Clause[]{
              new Clause(desiredDeparture, cEquals, "yes"),
              new Clause(desiredReturn, cEquals, "no"),
              new Clause(price, cLessThan, "800.00") },
            new Clause(flightRank, cEquals, "better"));
        Rule betterFlight2 =
          new Rule(rb, "betterFlight2",
            new Clause[]{
              new Clause(desiredDeparture, cEquals, "no"),
              new Clause(desiredReturn, cEquals, "yes"),
              new Clause(price, cLessThan, "800.00") },
            new Clause(flightRank, cEquals, "better"));
        Rule goodFlight =
          new Rule(rb, "goodFlight",
            new Clause[]{
              new Clause(desiredDeparture, cEquals, "no"),
              new Clause(desiredReturn, cEquals, "no"),
              new Clause(price, cLessThan, "600.00") },
            new Clause(flightRank, cEquals, "good"));
    }
}
```

Figure 8.14 Continued.

set to thirty seconds, and the state is set to INITIATED. After the **AirfareAgent** is initialized, the *processCIAgentEvent()* method is called to process any events it receives. The **AirfareAgent** only processes events with the *process* action. All other events are ignored. If a *process* event is received or if the agent is called synchronously, the *process()* method sends the airfare request to the Web page and checks the results to see if the returned flights are of interest to the user.

Let's examine the *initFlightRuleBase()* method in more detail. First, a **RuleVariable** called *departureTime* is defined that can take the value *morning*, *afternoon*, or *evening*. Another **RuleVariable**, *desiredDeparture*, can take the value *yes* or *no* to indicate whether the departure time falls within the user's preferences. Similar **RuleVariables** are defined for the return time as well. A **RuleVariable** called *rank* is used to rate each set of flight options and can take on the value of *good*, *better*, or *best*. In addition, **Condition** objects are defined for "=", "!=", ">", and "<".

The first **Rule**, *morningDeparture*, has an antecedent **Clause** defined as the *departs* **RuleVariable** being less than 1200 (noon). The consequent **Clause** of this **Rule** assigns *morning* to the *departureTime*. Rules for *afternoonDeparture*, *eveningDeparture*, *morningReturn*, *afternoonReturn*, and *eveningReturn* are defined in the same way, using the appropriate times and conditions in the antecedent **Clauses**.

The next set of **Rules** defines whether the departure and return times are in the desired range. The *desiredDeparture* is set to *yes* when the *departureTime* is *evening*. The *desiredReturn* is set to *yes* when the *returnTime* is *morning* or *afternoon*.

The last set of **Rules** ranks the flights. If the flight has a desired departure time, a desired return time, and costs less the $1000.00, its flight *rank* is set to *best*. If either the departure time or the return time is undesirable, but the cost is less than $800.00, the flight *rank* is set to *better*. If neither the departure time nor the return time are in the desired time periods, but the cost is less than $600.00, the flight *rank* is set to *good*. Note that if the flight falls outside of these times and prices, the flight *rank* is *null*.

The rule base is actually a little more complex than it needs to be, but it was created that way in order to illustrate rules with antecedent clauses that are dependent on the consequent clauses in other rules. This provides a slightly more interesting forward-chaining scenario because some rules cause new rules to be added to the conflict set as the processing progresses.

The *process()* method provides the bulk of the work done by the **AirfareAgent**. The first thing the *process()* method does is connect to the Web page and CGI-bin program that generates the flight information. The **java.net.URL** class and associated **java.net.HttpURLConnection** are part of the standard Java environment that enables our agent to easily send and retrieve information from the Web. Once the connection is open, a **PrintWriter** is used to send the parameters for the query to the CGI-bin program. After the parameters are sent, the output stream is closed and the input stream is read. A *while()* loop is used to read the HTML that is returned as a result of the airfare request. Each *inputLine* is parsed, and the times and prices for all the flights are stored in their respective arrays. The flight numbers are also stored in an array.

Once the input stream is closed, the flight times and prices are used to set the *departs*, *returns*, and *price* **RuleVariables**. The *forwardChain()* method is then called on the *rb* **BooleanRuleBase**. After the inferencing is complete, the *rank* **RuleVariable** is retrieved and if it is set to *good*, *better*, or *best*, a **CIAgentEvent** is created using the *actionString* and an appropriate message. Finally, the *notifyCIAgentEventListeners()* method is called to send the event to any interested agents.

The HTML Parsing Problem

When we developed the AirfareAgent, the original intent was that it would go out to a commercial Web site like *Travelocity*, or one of the airline sites, for its flight and airfare information. We actually implemented the agent and everything worked well until the format of the Web site changed. The problem we encountered was that we needed to retrieve certain content from the Web site, and the only way we could do that was to parse the HTML, looking for certain strings like "Price" or "Flight" to help us determine where the price or flight information was on the page. Unfortunately, with this approach, you are at the mercy of the Web page implementer. When the Web page changes, your parsing no longer works.

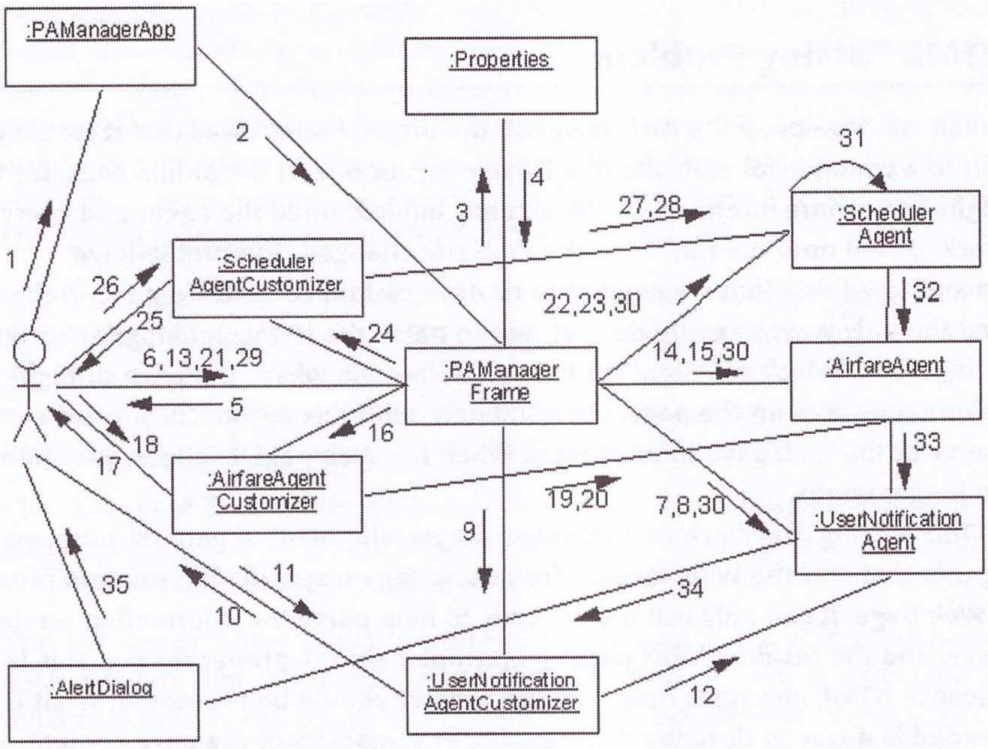
This is a big drawback that impedes the development of general-purpose agents that surf the Web, looking for and acting on specific information found in a Web page. It can only use textual cues to help parse the information on the page, and the results of this parsing may not yield the proper results. This is because HTML describes only how the content should be presented. What is needed is a way to describe the meaning or semantics of the page content. This is one of the reasons the eXtensible Markup Language (XML) was developed.

XML is the next generation of Web content markup languages. It is extensible in that it allows you to make up your own XML tags. To insure portability, it also provides a way to communicate what the tags mean so that others can read and process your XML documents. Using Document Type Definitions (DTDs), you can specify the new elements you have introduced, what the elements' attributes are, what values the elements can take, and relationships between elements. The rules specified in the DTD enable others to parse the XML document in an intelligent manner, and writing an XML parser is a fairly simple endeavor.

Unfortunately, XML is still in its fairly early stages of adoption in the industry. Most Web content is specified using only HTML, although a few XML editors and browsers are available. Once XML becomes more prevalent in the industry, writing intelligent agents that can do useful work out on the Web will become a much easier task.

The **AirfareAgent** can be used along with the **SchedulerAgent** and the **UserNotificationAgent** to check airfares periodically and notify the user whenever a flight is found that might be of interest to the user. The major interactions among the agents are illustrated in Figure 8.15.

The agents can be created and started using the *PAManager* application. It is important to create the agents in the correct order so that the event listener agent is created before the agent that generates the event. The scenario is shown on the following page.



1. start()
2. create()
3. readProperties()
4. return(availableAgents)
5. display()
6. select(UserNotificationAgent)
7. create()
8. registerListener(self)
9. open()
10. display()
11. enterData()
12. set()
13. select(AirfareAgent)
14. create()
15. registerListener(self)
16. open()
17. display()
18. enterData()
19. set()
20. registerListener(UserNotificationAgent)
21. select(SchedulerAgent)
22. create()
23. registerListener(self)
24. open()
25. display()
26. enterData()
27. set()
28. registerListener(AirfareAgent)
29. start(agents)
30. startAgentProcessing()
31. processTimerPop()
32. notify()
33. notify()
34. sendText()
35. display()

Figure 8.15 The PAManager collaboration UML diagram.

1. Run the **PAManagerApp**, if it is not already running.
2. Select **UserNotificationAgent** from the **Create** menu. Set the name and click on the **Initialize** button.
3. Select **AirfareAgent** from the **Create** menu. Fill in the customizer as shown in Figure 8.12. Use the drop-down list to select the **UserNotificationAgent** created in Step 2 as the agent to send an event to when a flight is found that meets the user's criteria. Click on the **Initialize** button.

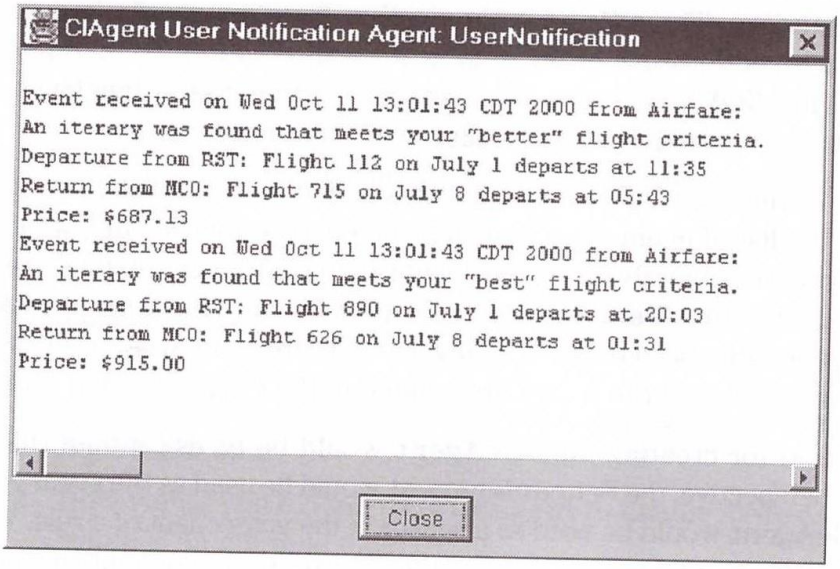


Figure 8.16 The UserNotification Alert dialog.

4. Select **SchedulerAgent** from the **Create** menu. Set the interval to 15 seconds. Use the drop-down list to select the **AirfareAgent** created in Step 3 as the agent to send an event to when its timer pops. Set the **Action string** to *process*. Click on the **Initialize** button.
5. Select each agent in the **CIAgent** list in the *PAManager* application window and use the **Start processing** option in the **Edit** menu to start each agent.
6. Every 15 seconds, when the **SchedulerAgent**'s timer pops, an event will be sent to the **AirfareAgent**. When a flight is found that matches the *good*, *better*, or *best* criteria, an event will be sent to the **UserNotificationAgent**. Figure 8.16 shows an example of the information displayed by the **UserNotificationAgent**. The text area at the bottom of the Personal Agent Manager Application window can be used to monitor activity as it occurs.
7. Use the **Exit** selection on the **File** menu to stop all the agents and end the application.

Discussion

In this chapter, we have sketched out quite a bit of function. The base *PAManager* application is quite flexible and can be used as a testbed for multiagent experiments. The agents we provided run the gamut from the monolithic and not-very-intelligent **FileAgent** to the more specialized **AirfareAgent**. We introduced the idea of having agents play roles with the **SchedulerAgent** and the **UserNotificationAgent**.

The agent services provided by *PAManager* are basic life-cycle services (create, start, suspend, resume, and stop). A more general implementation would provide a mechanism for agents to register their capabilities and possibly their interests. This function is

called directory services. The agent communication facilities are limited to passing **CIAgentEvent**s with action strings and simple arguments. A more general implementation would provide KQML or some other agent communication language. Both of these capabilities are explored in the following two chapters.

However, the use of the agent's JavaBean features added quite a bit to the application. Because we read the list of agent class names from a text properties file, we can easily extend the platform. Also, by relying on the agent's own customizer dialogs, we have no hard dependencies in the **PAManagerFrame** GUI on the individual agents. As long as the agent customizer can talk to the underlying agent, there is no problem. The *PAManager* application makes use of the behaviors defined in the **CIAgent** base class.

An alternate method for creating the **FileAgent** would be to use if-then rules as the engine. From this perspective, the **SchedulerAgent** would be used as the event-generating agent, and the **FileAgent** would be used as a sensor in the antecedent of a rule. Whenever the **SchedulerAgent** sent a **CIAgentEvent** to the **FileAgent**, it would invoke a forward-chaining inference cycle, with rules specifying the various file watch conditions. The three actions currently defined in the **FileAgent** would instead be defined as effector methods in the **BooleanRuleBase**. When a rule fires, the effector method would be called, and the correct action would be taken.

We also could have moved additional function, like the command execution function, out of the **FileAgent** and into a separate agent, much like we did with the alert capabilities in the **UserNotificationAgent**. 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 based on 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. The **UserNotificationAgent** can serve as the unifying interface between many agents running on the *PAManager* platform and a user, so it seemed like an obvious choice to break out into a separate agent.

Summary

In this chapter we developed a *Personal Agent Manager* application and four **CIAgents** that run on the *PAManager* platform. The main points include the following:

- The *PAManager* application allows a user to create **CIAgents** that communicate and cooperate with one another to perform useful tasks for the user. Any agent that extends the **CIAgent** class can be run on the *PAManager* platform.
- The **FileAgent** sleeps for 15-second intervals, and when it wakes up, it checks the state of the target file or directory. If a certain file condition is found, it can alert the user, execute a command, or send a **CIAgentEvent** to another agent.
- The **SchedulerAgent** can be used as a one-shot alarm or a repeated, interval alarm. When its timer goes off, it sends a **CIAgentEvent** to its registered listeners.
- The **UserNotificationAgent** receives **CIAgentEvent**s and displays the event messages in a dialog for the user.