Agent Interaction SDK Java Developer Guide

Voice Interactions

12/20/2025

Contents

Voice Interactions

This chapter shows you how to write AIL client applications that can log in and out; send, receive, and transfer phone calls; and set up conference calls.
As explained in chapter About the Code Examples, Genesys is developing two sets of examples. This chapter will explain how to use standalone examples that demonstrate these voice interactions.

Voice Interaction Design

To follow the discussion in this chapter, you will need the *Agent Interaction SDK 7.6 Java API Reference*, which is located in the `doc/` subdirectory under the Agent Interaction (Java API) product installation directory, and the source code for the `SimplePlace.java` and `SimpleVoiceInteraction.java` examples. Refer to the discussion in About the Code Examples for more information on how to use these examples.

Voice Interaction Data

Voice interactions are available through the `InteractionVoice` interface of the `com.genesyslab.ail` package. The `InteractionVoice` interface inherits the `Interaction` interface, and thus provides a set of interaction data to manage the interaction; the following list is not exhaustive:

- The interaction ID available through the `getId()` method.
- The date the interaction was created, available with the `getDateCreated()` method. This method is only available when a Universal Contact Server is connected.
- The subject of the interaction available with the `getSubject()` method. This method is available only when a Universal Contact Server is connected.

The `InteractionVoice` interface manages voice-specific data that characterize a voice interaction, such as:

- Dialed Number Identification Service (DNIS) number available with the `getDNIS()` method.
- Automatic Number Identification (ANI) number available with the `getANI()` number.
- The call type defined with the `InteractionVoice.CallType` enclosed class and available through the `getCallType()` method.

The `InteractionVoice` interface also includes a set of methods that allow your application to perform agent actions on the interaction. The `InteractionVoice.Action` class describes the possible agent actions on voice interactions, and each of its values corresponds to a method of the `InteractionVoice` interface. For instance, `InteractionVoice.Action.HOLD` corresponds to the`InteractionVoice.holdCall()` method.
Since `InteractionVoice` inherits `Possible`, your application can use an `InteractionVoice.Action` value to test whether or not an action can be requested at a certain point in time.
`InteractionEvents` propagate:

- The results of the actions taken on a voice interaction. If successful, those actions can change the

status of the voice interaction.

- Changes in status or information (for example, in attached data, parties, or extensions).
- The availability of, and changes to, possible actions.

Your application receives an `InteractionEvent` that has an `InteractionEvent.EventReason.POSSIBLE_CHANGED` reason when only the possible actions of the monitored interaction have changed. This can happen due to third-party changes that your application might not monitor. (Refer to the *Interaction SDK 7.6 (Java) Deployment Guide* for further details).

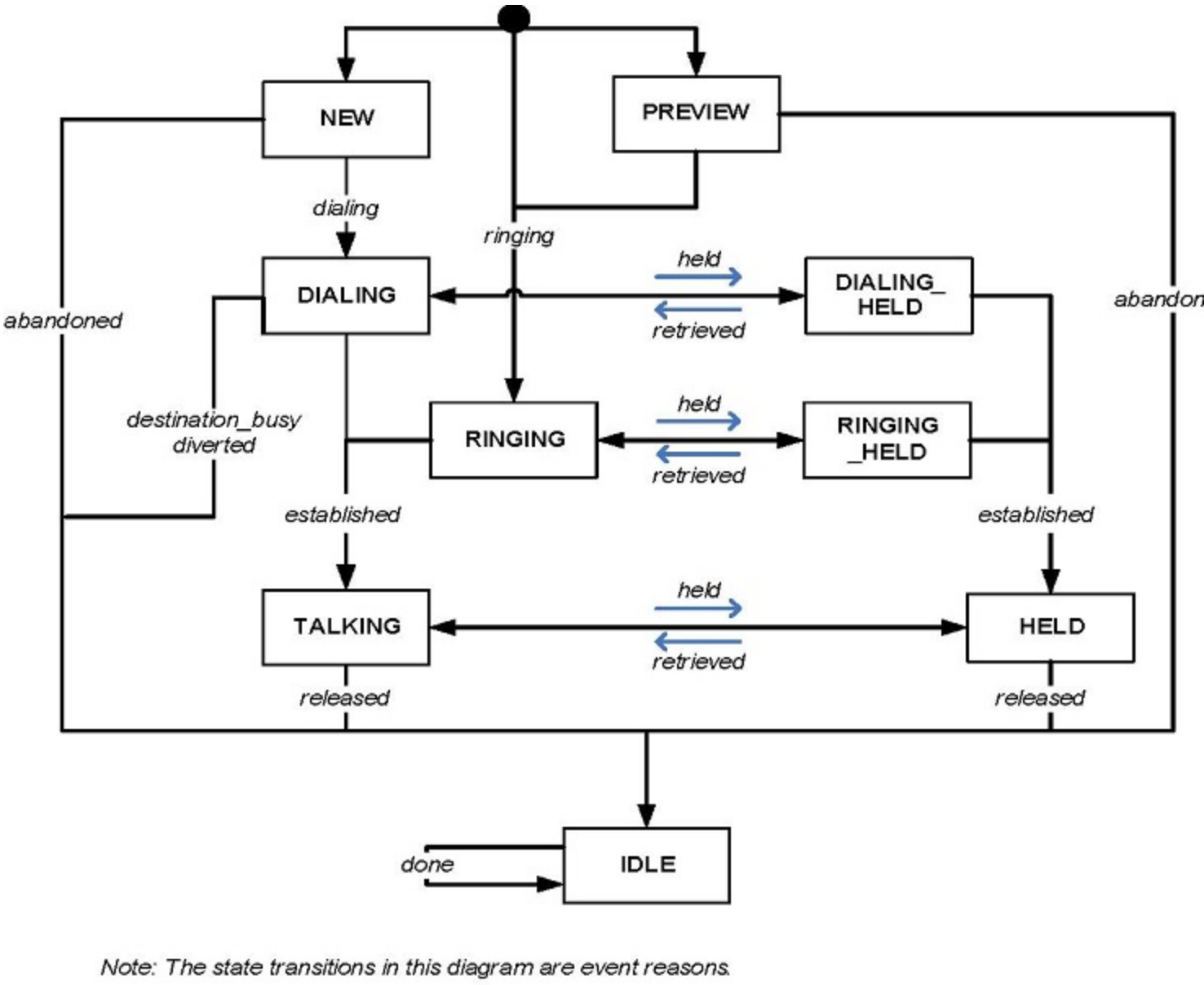
> **Important**
>
> Do not rely on event reasons to update your application; instead, use a refreshed list of possible actions. Event reasons, while they may change the value of what is possible, are primarily for information purposes.

Voice State Event Flow

The current state of a voice interaction is available with the `getStatus()` method as an `Interaction.Status` value.
The status of a voice interaction changes if:

- A successful action is confirmed by an event sent from the Genesys servers; for example, if the `InteractionVoice.Action.HOLD` action has been performed on the call, the voice interaction status changes to `Interaction.Status.HELD.`
- A CTI event changed it; for example, if a call is no longer dialing but now ringing, the voice interaction status changes to `Interaction.Status.RINGING`

The voice state diagram below is a generalized example that shows the main possible states of a voice interaction during its life cycle, considering it as an incoming or an outgoing phone call.



Generalized Example of a Voice State Diagram (Incomplete)

Warning

With respect to other roles for your custom application, that diagram is intended as a generalized example only. It does not include all possible life cycle for voice interactions. Both states, transitions, and EventReasons are switch specific.

Refer to the T-Server Deployment Guide for your environment, to the Genesys 7 Events and Models Reference Manual for model details, and to the Agent Interaction SDK 7.6 Java API Reference for the full lists of reference material relating to the Agent Interaction (Java API).

Statuses are switch-specific and are not available for switches that do not support the feature associated with this status. For example, if the held feature is not available on a particular switch, the`InteractionVoice.Action.HOLD` action is not available. This has consequences for `Interaction.Status.HELD` and `Interaction.Status.DIALING_HELD` status:

- If `InteractionVoice.Action.HOLD` is unavailable, the `Interaction.Status.HELD` and `Interaction.Status.DIALING_HELD` status are not reachable.
- Some switches have the`Interaction.Status.HELD` feature but do not allow its use during the dialing of the call, in which case the`Interaction.Status.DIALING_HELD` status is not reachable.

The possible statuses, transitions and event workflow differ from one switch to another. For additional details, see also Switch Facilities.

Six Steps to an AIL Client Application

Now that you have been introduced to the Agent Interaction (Java API), it is time to outline the steps you will need to work with its events and objects. There are six basic things you will need to do in your AIL applications:

- **Implement a listener** from among those provided by AIL. The new examples use a `PlaceListener`, since this listener has access to all three of the event types you will most likely need—namely, `Dn` events, `Place` events, and `Interaction` events. Here is how `SimplePlace` does this:

  ```
  public class SimplePlace implements PlaceListener {
  ```

- **Connect to AIL.** The code examples use the `Connector` application block to do this, as explained in Application Essentials:

  ```
  Connector connector = new Connector();
  connector.init(agentInteractionData.getApplicationParameters());
  ```

- **Set up button actions** (or actions on other GUI components) tied to AIL functions. The standalone code examples have a `linkWidgetsToGui()` method that does this.
- **Register your application** for events on the object that your listener refers to. The standalone code examples use a `PlaceListener`, so they use this method call to register with the `Place` object:

  ```
  samplePlace.addPlaceListener(this);
  ```

- **Synchronize the user interface** with the state of the AIL objects to which your application refers. The standalone examples have two methods for this: `setPlaceWidgetState()` and `setInteractionWidgetState()`. These methods make use of the `isPossible()` method to determine whether the action linked to a particular button is possible. If it is, the button is enabled, like this:

```
loginButton.setEnabled(sampleDn.isPossible(Dn.Action.LOGIN) );
```

- **Add event-handling code** to the appropriate AIL event handler. The standalone code examples use the `handleDnEvent()`, `handlePlaceEvent()`, and `handleInteractionEvent()` methods, which are required by the `PlaceListener` interface.

The standalone code examples have been designed to make these steps stand out so that you can quickly learn to write your own real-world applications. Now it is time to see how they are implemented in the `SimplePlace` example.

SimplePlace

The `SimplePlace` example provides a GUI-based desktop application that lets agents log in, set their status to ready, and perform other preliminary tasks. These tasks use `Dn` and `Place` events.
The buttons for these tasks are part of the `Simple Place` panel located in the upper-left corner of the user interface.

The panel containing these buttons has a light green background. Note that the buttons themselves will be changing from enabled to disabled, and back again, as the agent status changes based on the flow of `Dn` and `Place` events.
There is also some status information on the left, on the right and a log panel at the bottom of the application window.
This section will focus on the API features for working with `Dn` and `Place` events, but most of the concepts you will learn here can be applied to `Interaction` events, too.
The following subsections show how `SimplePlace` carries out the six steps to writing an AIL standalone application.

Implement a Listener

This is a simple step, which is accomplished in the class declaration:

```
public class SimplePlace implements PlaceListener {
```

AIL has four listener interfaces. `SimplePlace` uses `PlaceListener` because it can handle the three types of events used in the code examples:

- `DnEvent`—The standalone code examples use a login method that ties an agent to a DN. `DnEvent` s inform the application of the agent's status in relation to the DN; for instance, whether the agent can log in, or whether he or she can be made ready to make and receive calls.
- `PlaceEvent`—The standalone code examples also use a multimedia login that is associated with a place. This login allows the agent to process things like e-mail or open media interactions. These events are similar to `DnEvents` and inform the application whether the agent can log in for multimedia processing, and whether he or she can be made ready to send and receive multimedia interactions.
- `InteractionEvent`—These events are generated as interactions go through their life cycle. For instance, when there is an incoming call, the application will receive an interaction event with a status of `RINGING.` When the agent answers the call, the status of the interaction changes to `TALKING` and the application will receive an event to that effect.

Connect to AIL

The standalone code examples include the `SimpleConnector` class which implements a `WindowListener`. This class makes calls to the `Connector` application block to establish the all-important connection to the AIL, and to release the connection when the user closes the application.

For more information on how the `Connector` application block connects, please refer to the Application Essentials section of About the Code Examples. For the purposes of this example, here is all you need to do:

```
Connector connector = new Connector();
connector.init(agentInteractionData.getApplicationParameters());
```

Set up Button Actions

`SimplePlace` can carry out the five actions that an agent takes to set his or her status: log in, log out, become ready, become not ready, and carry out after-call work. The `AgentInteractionGui` class has created buttons for each of these actions, but at this point they do nothing. It is the job of `SimplePlace` to bring these buttons to life.<

To do this, `SimplePlace` has a method called `linkWidgetsToGui()` that links to the GUI buttons and then sets up actions for them. For each button, there is a statement like this:

```
loginButton = agentInteractionGui.loginButton;
```

Now that `loginButton` is available, `SimplePlace` can assign an action to it:

```
loginButton.setAction(new AbstractAction("Log In") {
        public void actionPerformed(ActionEvent actionEvent) {
                try {
                        if(voice)
                                // Perform a voice-only login
                                samplePlace.login(agentInteractionData.getLoginId1(),
                                                agentInteractionData.getPassword1(),
                                                agentInteractionData.getQueue(), null, null,
null);
                        else if(mediaList != null)
                                // Perform a multimedia login (this login is for all
                                // media types other than voice)
                                samplePlace.loginMultimedia( sampleAgent, mediaList, null,
null);
                } catch (Exception exception) {
                        exception.printStackTrace();
                }
        }
});
```

As mentioned above, there are two logins here. The first one is for voice use only. While the `login()` method is explicitly associated with a place, the Configuration Layer already has information on the agent's DN. This DN will be the basis for the `DnEvent` activity associated with this login. For more information on the voice-based `login` method, see `Place` in the API Reference.

The second login (`loginMultimedia`) is for non-voice media and uses a `Collection` of media types that inherited multimedia examples set up in the `setSampleType()` method, as shown here.

```
// Collection of media types for multimedia methods
```

```
mediaList = new LinkedList();
// Add the media types used by these examples
mediaList.add("email");
voice = false;
```

Since inherited examples will be working with e-mail, chat, and open media interactions, they disable voice and add the required media to the `mediaList`, using the Configuration Layer's terms for each of them (`email`, `chat`,and `workitem`, respectively). As pointed out in the API Reference, you can also issue the `loginMultimedia()` method with a parameter of `null` instead of an explicit media list. This will log the agent into all of the media types available for the specified place.

Now that the agent is logged in, `SimplePlace` needs to update the GUI by calling `setInteractionWidgetState()` and `setPlaceWidgetState()`. This will be explained in detail below.

The other buttons have a similar structure that allows them to perform logout, ready, not ready, and after-call-work functions.
After the buttons have been set up, there are a few lines of code that link various status fields to the GUI and populate them with configuration information.

```
loginNameLabel = agentInteractionGui.loginNameLabel;
loginNameLabel.setText("Login Name: "
+ agentInteractionData.getAgent1UserName());
...
```

Register Your Application

The next step is to register your application so it can send and receive the events you will need to work with interactions. This is the last thing done by `linkWidgetsToGui()`:

```
try {
   // THIS IS AN IMPORTANT STEP:
   // Register this application for events on the sample place
   samplePlace.addPlaceListener(this);
} catch (Exception exception) {
   exception.printStackTrace();
}
```

`SimplePlace` has access to a DN (`sampleDn`), an agent (`sampleAgent`), media (`sampleEmail`, `sampleChat`, and `sampleOpenMedia`), and a place (`samplePlace`). Since it is using the `PlaceListener` interface, it adds a place listener to `samplePlace`.

Synchronize the Widgets

The standalone code examples use two similar methods to synchronize their user interface widgets with the application state: `setPlaceWidgetState()` and `setInteractionWidgetState()`. `SimplePlace` implements only the first one, since it does not process interactions. Each of these methods uses the `isPossible()` method to determine whether a particular button should be enabled. Here is the code to enable or disable the `loginButton` for voice examples:

```
loginButton.setEnabled(sampleDn.isPossible(Dn.Action.LOGIN));
```

As you can see if you look in the API Reference, this method is checking whether the `LOGIN` action is possible on the sample DN. If it is, the button will be enabled. Otherwise, it will be disabled.

The same thing is done for each of the other buttons in the SimplePlace user interface.

Add Event-Handling Code

Each type of event handled by the PlaceListener interface has its own event-handling method. Classes implementing this interface must include each of these methods, although the method bodies may be empty. Because SimplePlace is interested in DN and place events, it has functional handleDnEvent() and handlePlaceEvent() methods.

As explained in the Threading section in About Agent Interaction (Java API), the standalone code examples use threads to avoid delaying the propagation of events.

In this purpose, the SimplePlace uses DnEventThread, PlaceEventThread, and InteractionEventThread classes to respectively process DnEvent, PlaceEvent, and InteractionEvent events.

Most of the code in these classes writes messages to the log panel at the bottom of the SimplePlace user interface. SimplePlace is not actually processing interactions, but further examples use place actions for widgets used to create interactions, so the code in each of the run() methods of these DnEventThread and PlaceEventThread classes is:

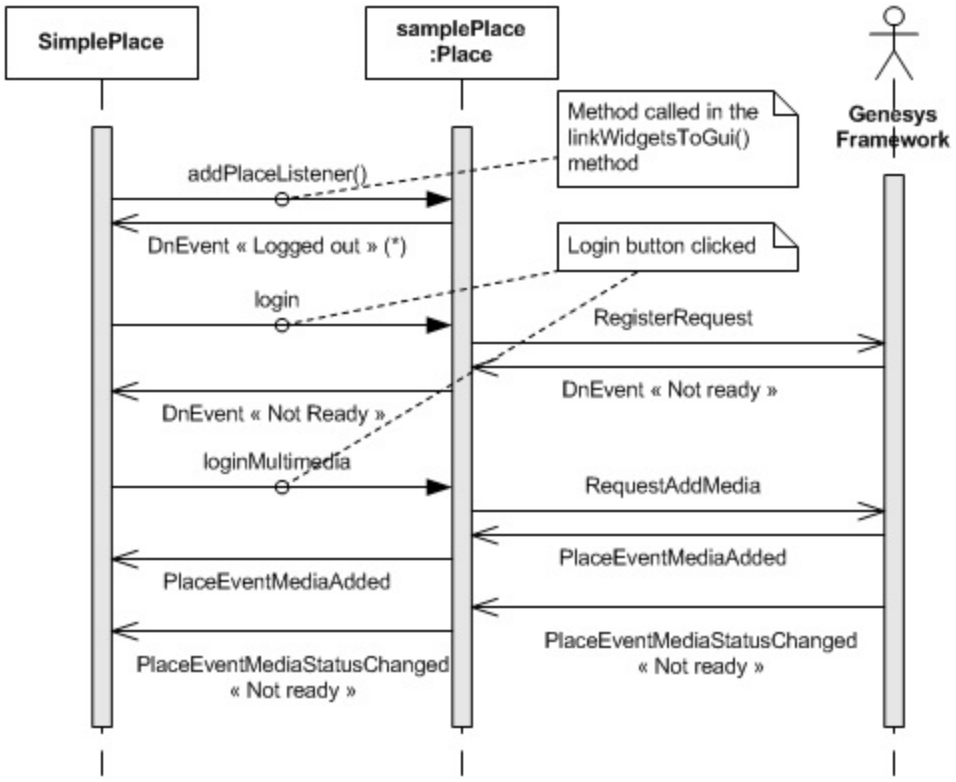
```
// THIS IS AN IMPORTANT STEP:
// As the status changes, enable or disable the buttons
setPlaceWidgetState();
setInteractionWidgetState();
```

As the comments indicate, this is an important step. If you do not have a line like this in your event-handling threads, the user interface will be out of sync with the state of the objects and events with which you are working.

As for the handlePlaceEvent() and handleDnEvent() methods, the handleInteractionEvent() method code uses the InteractionEventThread classes to write messages to the log panel. However, it does not include event-handling logic. The SimpleVoiceInteraction example will handle interaction events. At that point, you will see some more complicated event-handling code, but for this example, this is all you have to do for your event handlers.

The Importance of Timing

It is important to note that if you want your application to work, certain steps must be executed before others. For example, you need to register your application—by issuing the samplePlace.addPlaceListener(this) method call—*before* you can receive events. Timing For Login shows the sequence of method calls and events involved for managing login with the SimplePlace example.



Timing For Login

Likewise, you will need to synchronize the user interface every time you handle an event, or else your buttons will not reflect the appropriate capabilities. This synchronization can be tricky, but if you experiment with the code examples, you will start to get a feel for how things fit together. You might want to run the examples with certain lines commented out, or placed in a different order, so that you can see how this affects your event handling.

Wrapping Up

If you can master the preceding six steps, you will have the foundation for writing your own AIL standalone applications. However, there is also some code in the `SimplePlace` constructor that you might be curious about. In order to make it easier to understand this example—and the other standalone examples—here is a brief explanation of how the `SimplePlace()` constructor performs the setup tasks for the `SimplePlace` object.

Set Sample Type

The first statement calls the `setSampleType()` method, which sets the value of a field that will tell the GUI which example is being executed.

Connect to AIL and Make Configuration Data Available

Next, the `SimplePlace()` constructor creates a new instance of `Connector`. This class reads the

configuration data from AgentInteraction.properties and connects to AIL, as described in Application Essentials.
After Connector returns, the constructor links to the AgentInteractionData instance that makes Configuration Layer data available to the examples, including the IDs of an Agent, Place, and Dn, which are retrieved via the AilFactory instance:

```
sampleAgent = (Agent) connector.ailFactory.getPerson(
agentInteractionData.getAgent1UserName());
samplePlace = connector.ailFactory.getPlace( agentInteractionData.getPlace1());
sampleDn = connector.ailFactory.getDn( agentInteractionData.getDn1());
```

Create and Link to the GUI

At this point, the constructor calls AgentInteractionGui , which creates the graphical user interface.

```
// Create the GUI
agentInteractionGui = new AgentInteractionGui(windowTitle, sampleType);
```

With the GUI components created, it is possible to link them to actions that affect AIL objects. This is done with a call to the linkWidgetsToGui() method. As explained above, this method also includes the statement that registers the application to receive events on samplePlace.

```
// Link the GUI components to API functionality
linkWidgetsToGui();
```

Start the Application

Finally, there are a few lines of code that set up the GUI and make it visible.

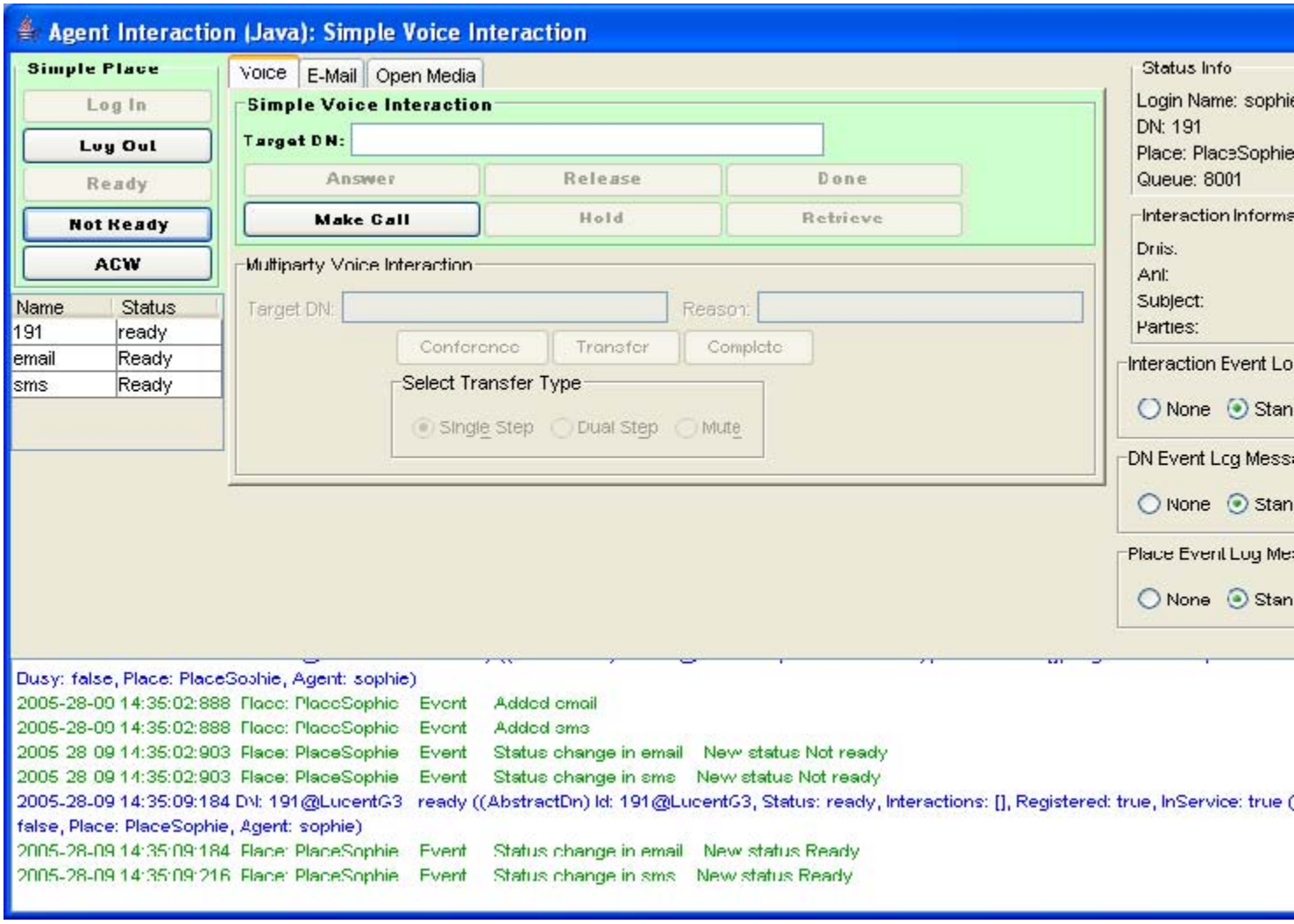
```
// Start the application
agentInteractionGui.mainGuiWindow.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE);
agentInteractionGui.mainGuiWindow.addWindowListener(connector);
agentInteractionGui.mainGuiWindow.pack();
agentInteractionGui.mainGuiWindow.setVisible(true);
```

About the User Interface

Now that you understand the basics of the SimplePlace application, you can start running it in your environment. As you do so, you will notice that you are receiving event messages in the log panel at the bottom of the application window. The user interface is designed to make it easy for you to track these messages by giving each type its own color. The DnEvent messages are blue, PlaceEvent messages are green, and InteractionEvent messages are red.
You can also turn off each of the message types so that you can focus on certain messages. To do this, use the radio buttons on the right side of the application window. In addition, you can get more detailed messages by clicking the Debug radio button for a given message type. If you want to, you can customize the log messages created in the event handlers. As you experiment with these messages, you will get a better understanding of the event flow in your application.

SimpleVoiceInteraction

`SimpleVoiceInteraction` extends `SimplePlace` . While `SimplePlace` shows how to log your agent in and out and otherwise change his or her status, `SimpleVoiceInteraction` shows how to make and receive calls. `SimpleVoiceInteraction` uses the same user interface as `SimplePlace`, but one more panel of the GUI is activated. To make or receive calls, your agent must be logged in and ready, as shown in Agent Is Ready. As you can see, the `Make Call` button is enabled, indicating that the agent can type a number into the `Target DN` field and press the button to initiate the call. Likewise, if there is a call waiting for the agent to answer it, the `Answer` button will be enabled and the agent can click it to receive the call.



Agent Is Ready

As you might expect, the `Hold` button allows you to put a call on hold, the `Retrieve` button re-

activates the call, the `release` button cuts your connection to the call, and the `Mark Done` button marks the interaction as done.

Now that you have an idea of what this example does, here is a description of how it carries out the six steps in writing an AIL application.

Implement a Listener

`SimpleVoiceInteraction` is a subclass of `SimplePlace`. Because of this, it already implements the `PlaceListener` interface. Here is the class declaration for `SimpleVoiceInteraction`:

```
public class SimpleVoiceInteraction extends SimplePlace {
```

Connect to AIL

This step has been done for you already, since the `SimpleConnector` constructor calls `Connector` to make the connection to AIL. For further details, see Connect to AIL.

Set up Button Actions

Since `SimpleVoiceInteraction` needs to use the `SimplePlace` buttons, the first thing done by the `linkWidgetsToGui()` method is call the superclass method:

```
super.linkWidgetsToGui();
```

The voice-interaction-based examples share the same tab in the middle of the user interface. Now `SimpleVoiceInteraction` can link to the GUI buttons and add button actions to them. The code to carry out these actions must be wrapped in a `try/catch` block, but beyond that, it can be fairly simple, as shown in these examples for the `Answer` and `Release` buttons:

```
sampleInteraction.answerCall(null);
...
sampleInteraction.releaseCall(null);
```

Other buttons require a bit more code to allow the application to process the interaction. For example, the `Make  Call` button needs to create a new voice interaction that is associated with `sampleInteraction` before making the call, as shown below.

```
// Create a new interaction for use in making the call
sampleInteraction = (InteractionVoice) samplePlace.createInteraction(MediaType.VOICE, null,
agentInteractionData.getQueue());
if (sampleInteraction instanceof InteractionVoice) {
   // Make the call, using the phone number provided by
   // the agent
   sampleInteraction.makeCall(
      simpleVoiceTargetDn.getText(), null,
      InteractionVoice.MakeCallType.REGULAR, null,null, null);
}
```

For more information about these steps, see the *Agent Interaction SDK 7.6 API Reference*.

Register Your Application

This step was done for you by SimplePlace when you called super.linkWidgetsToGui(), as described in the previous section.

Synchronize the User Interface

The setInteractionWidgetState() method is very similar to the setPlaceWidgetState() method used by SimplePlace. It is called by the handleInteractionEvent() handler, but it can also be called in any other situation requiring an update to the interaction-related buttons.
This method checks to see whether there is a voice interaction associated with the application. At that point, it uses the isPossible() method to enable or disable the user interface buttons, as shown here for the Answer button:

```
if (sampleInteraction!=null) {
   answerButton.setEnabled(sampleInteraction.isPossible(InteractionVoice.Action.ANSWER_CALL));
   makeCallButton.setEnabled(sampleInteraction.isPossible(InteractionVoice.Action.MAKE_CALL));
   //...
}
```

If there is no interaction associated with the application, the buttons are all disabled, except the Make Call button. This button is enabled if the MAKE_CALL action is available for sampleDn, as shown here:

```
answerButton.setEnabled(false);
makeCallButton.setEnabled(
sampleDn.isPossible(Dn.Action.MAKE_CALL));
releaseButton.setEnabled(false);
doneButton.setEnabled(false);
holdButton.setEnabled(false);
retrieveButton.setEnabled(false);
```

Add Event-Handling Code

Because SimplePlace implements the PlaceListener interface, it must implement the handleInteractionEvent() method. But since SimplePlace does not process interactions, this method body does not include event-handling logic, and only writes messages to the log console. SimpleVoiceInteraction , on the other hand, is designed to handle voice interactions. This means there needs to be interaction-related, event-handling code.
As explained in the Threading section in About Agent Interaction (Java API), the standalone examples use threads to avoid delaying the propagation of events. The SimpleVoiceInteraction uses VoiceInteractionEventThread instances to process InteractionEvent events.
Since SimpleVoiceInteraction needs to write a message to the log console, the first thing that the handleInteractionEvent() method does is to call the superclass method (which will create a thread to process this task):

```
super.handleInteractionEvent(event);
```

As with SimplePlace, the event-handling code in VoiceInteractionEventThread is fairly simple. It

checks several statements and implements the following action items:

1. Check whether the interaction event involves a voice interaction:

```
if(event.getSource() instanceof InteractionVoice)
```

2. If no voice interaction is associated with the application, check whether the event provides notification of a `RINGING` voice interaction. In this case, the event means there is an incoming phone call: `sampleInteraction` needs to be associated with the event's interaction so the application can process the call:

```
if (sampleInteraction == null
&& event.getStatus() == Interaction.Status.RINGING) {

// Associate sampleInteraction with the event source
sampleInteraction = (InteractionVoice) event.getSource();

//...
}
```

3. Check whether the interaction associated with the example is idle and is done. If so, the interaction is removed:

```
 // If the interaction is idle and done,
// the example no longer handles it.
if (sampleInteraction!=null && interaction.getId() == sampleInteraction.getId()
   && event.getStatus() == Interaction.Status.IDLE
   && sampleInteraction.isDone() )
{
   sampleInteraction = null;
   simpleVoiceTargetDn.setText("");
}
```

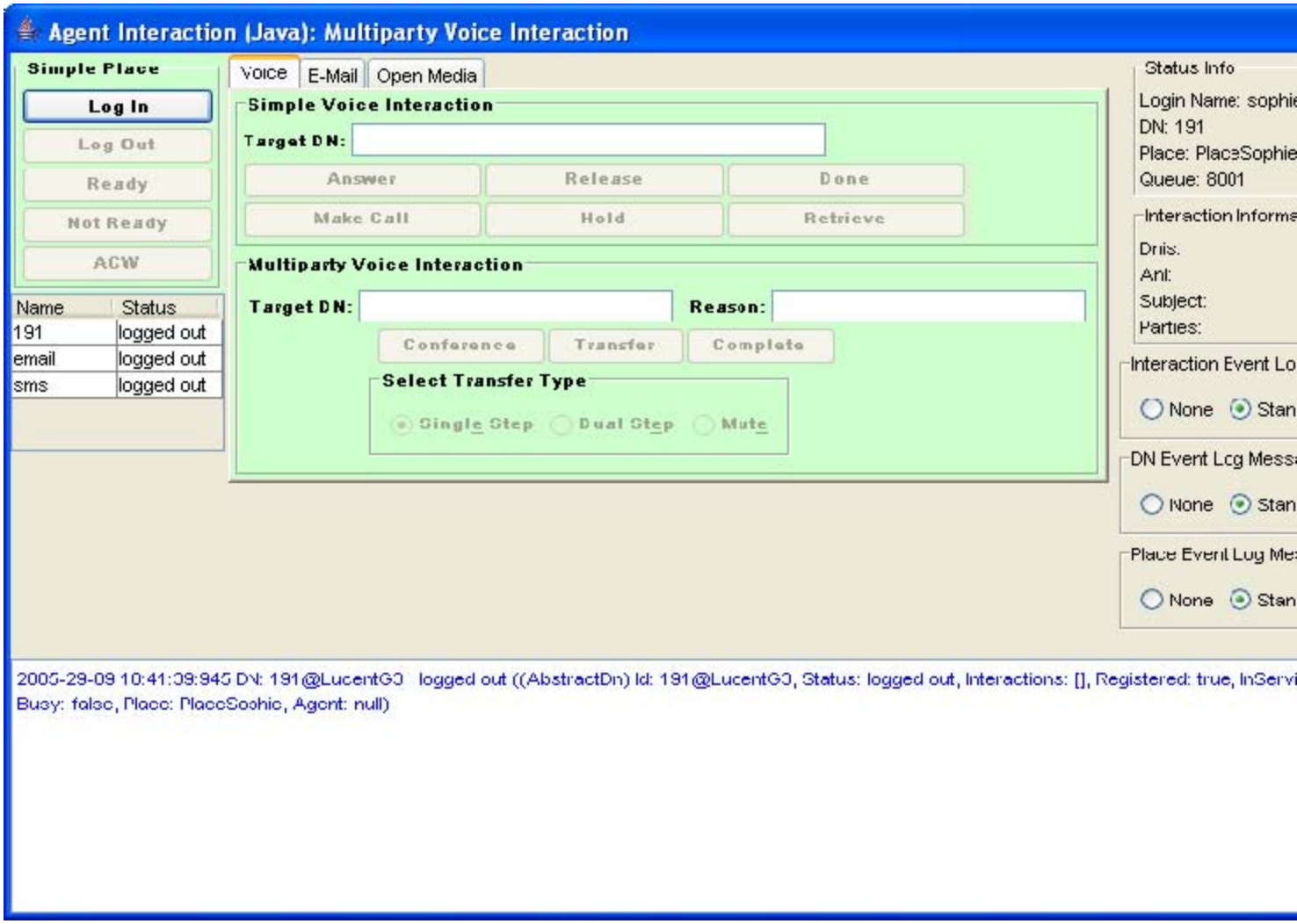
4. Finally, the GUI must be updated to keep in sync with the state of the application:

```
 setInteractionWidgetState();
```

Notice that the interaction-related widgets are updated here—not the place widgets. Interaction events do not normally affect the status of the DN.
As you can see, there were only a few additional items to take care of when extending `SimplePlace` to handle voice interactions.

MultipartyVoiceInteraction

You now know how to make and receive calls. But what if your agents need to transfer a call or set up a three-way conference call?
The `MultipartyVoiceInteraction` example shows how to do this. As you can see below, the `Multiparty Voice Interaction` panel is activated in the user interface.



MultipartyVoiceInteraction at Launch Time

This panel has fields to enter the DN to which you want to transfer, or conference with, and the reason for this action. It also has buttons to carry out the conference or transfer and then, if you are doing a dual-step transfer or a conference call, to complete it. At the bottom of the panel there are three radio buttons letting you choose the type of transfer you want to carry out.

As we have seen in Six Steps to an AIL Client Application, there are six steps you will need to carry out to write this application. But this example is a subclass of `SimpleVoiceInteraction`, so many of the steps you would otherwise need to accomplish have already been done for you. In discussing this example, we will omit those steps.

However, it is important to note that this example involves actions that require a call to be in progress. So before you can conference or transfer a call, you will have to use the buttons in the upper panel to answer or make a call. At that point, you will have an interaction available for further action.

Set up Button Actions

Since `MultipartyVoiceInteraction` needs to use the `SimplePlace` and `SimpleVoiceInteraction` buttons, the first thing the `linkWidgetsToGui()` method does is call the superclass method:

```
super.linkWidgetsToGui();
```

After that, it links the application to the GUI widgets, this time including two text fields and toggle buttons:

```
multipartyVoiceTargetDnLabel = agentInteractionGui.multipartyVoiceTargetDnLabel;
multipartyVoiceTargetDnText = agentInteractionGui.multipartyVoiceTargetDnTextField;
multipartyVoiceReasonText = agentInteractionGui.multipartyVoiceReasonTextField;

singleStepTransfer = agentInteractionGui.singleStepTransferRadioButton;
dualStepTransfer = agentInteractionGui.dualStepTransferRadioButton;
muteTransfer = agentInteractionGui.muteTransferRadioButton;
```

Now you can set up the button actions. In this example, these actions are generally more complicated than in `SimpleVoiceInteraction`. One reason for this is that you have to keep track of whether you are going to do a transfer or a conference call. In order to help with this, there is a `boolean` field called `thisIsAConferenceCall`. This field will be set to `true` if you are making a conference call, or to `false` for a transfer.

The toggle buttons indicate which type of conference or transfer is selected. When the user clicks each radio button, the user interface deselects the others. The `Transfer` and `Conference` buttons invoke dedicated methods that take this selection into account and perform the appropriate action, as shown here to transfer a call:

```
thisIsAConferenceCall = false;
performTransfer();
```

The `performTransfer()` method has to take into account the possibilities that you are doing a single-step, a dual-step, or a mute transfer. For each transfer, the method call is fairly simple, but all transfer types have to be accounted for. A `try/catch` block surrounds the following code snippet:

```
// If "Single step" is selected...
if (singleStep.isSelected() && sampleInteraction != null) {
   // ...a single step conference is required
   sampleInteraction.singleStepConference(getTransferTarget(), null, null, null, null);

   // If "Dual step" is selected...
} else if (dualStep.isSelected() && sampleInteraction != null) {
   // ...a dual step conference is required
   sampleInteraction.initiateConference(getTransferTarget(), null, null, null, null);
// If "Mute" is selected...
} else if (muteTransfer.isSelected() && sampleInteraction != null) {
// ...by default, a single step conference is performed
   sampleInteraction.singleStepConference(getTransferTarget(), null, null, null, null);
}
```

The `Complete` button must take into account whether you are doing a transfer or a conference, but it

is otherwise fairly simple:

```
if (thisIsAConferenceCall) {
   sampleInteraction.completeConference(null, null);
} else {
   sampleInteraction.completeTransfer(null, null);
}
```

Now you can set up which toggle buttons are visible. Not all switches can perform every transfer and conference function. The `Switch` class tells you which functions are available through its `isCapable()` method. The `MultipartyVoiceInteraction` example displays only those toggle buttons whose mode is available at some point during runtime.

```
if(sampleDn instanceof Dn )
{
        Switch theSwitch = sampleDn.getSwitch();
        if (theSwitch != null) {

                switchCanDoSingleStep =
                        theSwitch.isCapable( InteractionVoice.Action.SINGLE_STEP_TRANSFER)
                        || theSwitch.isCapable(
InteractionVoice.Action.SINGLE_STEP_CONFERENCE);

                switchCanDoMuteTransfer =
theSwitch.isCapable(InteractionVoice.Action.MUTE_TRANSFER);

                switchCanDoDualStep =
theSwitch.isCapable(InteractionVoice.Action.INIT_TRANSFER)
                || theSwitch.isCapable(InteractionVoice.Action.CONFERENCE);
        }
}
singleStep.setVisible(switchCanDoSingleStep);
muteTransfer.setVisible(switchCanDoMuteTransfer);
dualStep.setVisible(switchCanDoDualStep);
```

For further details about switch features, refer to Switch Facilities.

Synchronize the User Interface

The first step for the `setInteractionWidgetState()` method is, as usual, to call the superclass method. After that, you do the usual checks to see if the various buttons and radio buttons should be enabled or disabled.
The `Transfer` and `Conference` buttons are enabled if at least one type of transfer or conference is available, as shown in the following code snippet.

```
//The transfer button should be enabled if at least one type
// of transfer is available:
// single step OR mute OR dual step
boolean transfer =
(sampleInteraction.isPossible(InteractionVoice.Action.SINGLE_STEP_TRANSFER))
 || (sampleInteraction.isPossible(InteractionVoice.Action.MUTE_TRANSFER))
 || (sampleInteraction.isPossible(InteractionVoice.Action.INIT_TRANSFER));
transferButton.setEnabled(transfer);
```

Add Event-Handling Code

The first step for the `handleInteractionEvent()` method is, as usual, to call the superclass method and create a thread to process the interaction event. Since `MultipartyVoiceInteraction` handles multiparty interactions, interaction events may include multiparty-related information. This information is described in `InteractionEvent.Extension` and is available by calling the `InteractionEvent.getExtension()` method.

```
HashMap map = (HashMap) event.getExtensions();
String info ="";
if(map.containsKey(InteractionEvent.Extension.RINGING_TRANSFER_REASON))
{
   info += "Transferred ("
      + ((String)map.get(InteractionEvent.Extension.RINGING_TRANSFER_REASON))+")";
   multipartyVoiceReasonText.setText(info);
}
```

Instant Messaging

The Instant Messaging feature is available only for places which include a SIP DN that is configured for multimedia. Because of this relationship to a place, your application needs an `InteractionVoice` instance to handle instant messaging features. The Instant Message (IM) interactions have a `MediaType.CHAT` and are tightly coupled to `IMInteractionContext` objects.
Handling Instant Messaging also leads your application to deal with additional classes of the `com.genesyslab.ail.im` package, as explained in following subsections.

> **Important**
>
> The Agent Interaction Code Samples include an instant messaging example, the `SimpleIM` class.

Starting an Instant Messaging Session

To start an instant messaging session, your application should first create a voice interaction of `MediaType.CHAT` , as shown in the following code snippet:

```
InteractionVoice sampleInteraction = (InteractionVoice)
samplePlace.createInteraction(MediaType.CHAT, null, agentInteractionData.getQueue());
```

Your application can then retrieve an `IMInteractionContext` instance tight to this interaction by calling the `AilFactory.getIMInteractionContext` () method, as shown here:

```
sampleContext = ailFactory.getIMInteractionContext(sampleInteraction);
```

Then to connect a party, make a call, as shown below:

```
sampleInteraction.makeCall( "SIP DNID", null, InteractionVoice.MakeCallType.REGULAR, null,
null, null);
```

Handling Instant Messages

Send a Message

To send a message, your application needs a call to the `IMInteractionContext.sendMessage(String, String)` method, as shown here:

```
sampleContext.sendMessage(“My instant message”, "text/plain");
```

Get the Session Transcript

When your application gets an `IMInteractionContext` instance, it can retrieve all the session transcript messages and party events by calling the `getTranscript()` method. The following code snippet shows how to read the transcript.

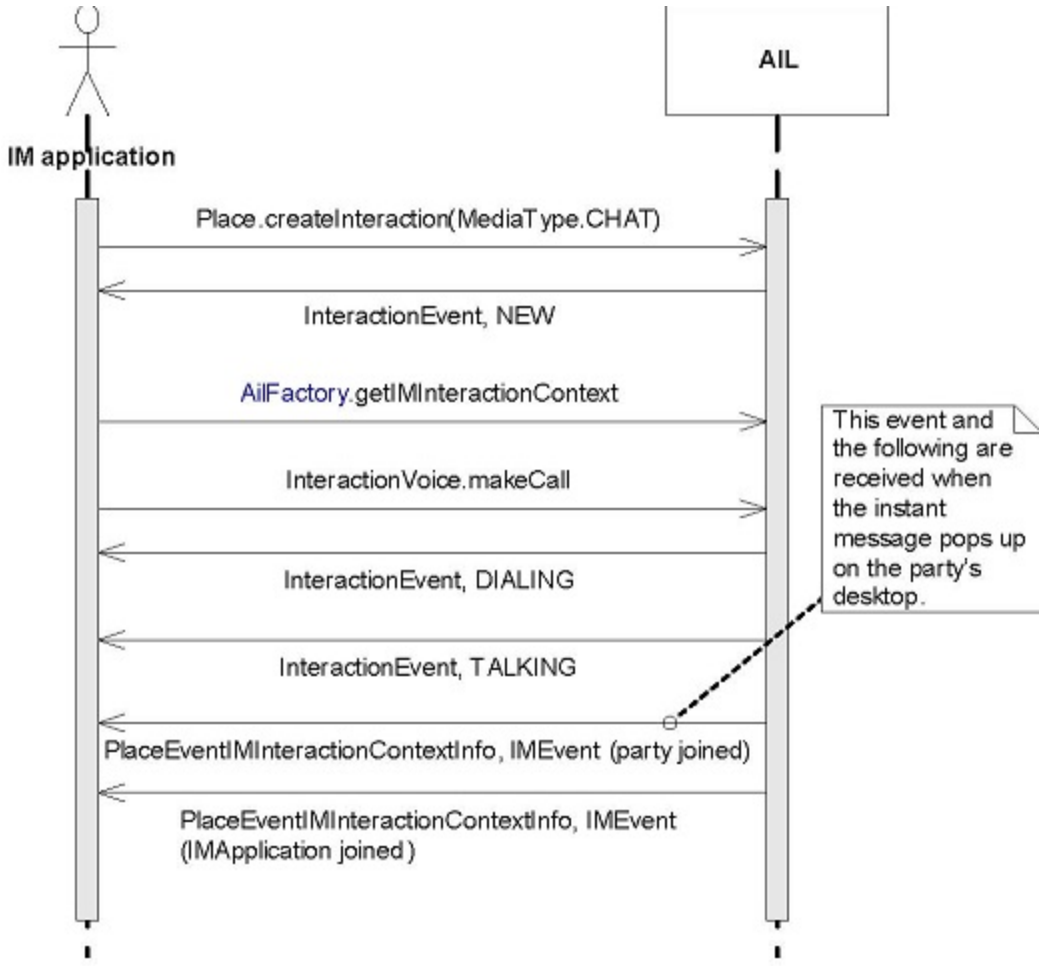
```
Iterator it = sampleContext.getTranscript().iterator();
while (it.hasNext())
{
        IMEvent ev = (IMEvent) it.next();
        // Process the event
        if( ev instanceof IMMessage)
        {
                IMMessage msg = (IMMessage) ev;
                IMParty party = msg.getParty();
                System.out.println(party.getNickname()+"> "+msg.getContent());
        } else if(ev instanceof IMPartyJoined)
        {
                System.out.println(ev.getParty().getNickname()+" has joined ");
        }
        else if(ev instanceof IMPartyLeft)
        {
                System.out.println(ev.getParty().getNickname()+" has left "); }
}
```

Handle Events

At runtime, AIL provides two types of events that your application can handle by implementing the `PlaceListener` interface:

- `InteractionEvent` for status changes and information modification (when a party sends a message, or joins, or leaves).
- `PlaceEventIMInteractionContextInfo` to get the `IMEvent` that contains a new message or the modified party’s information.

When your application starts a new session and is connected to parties and ready for sending and receiving instant messages, the `Interaction.Status` of your interaction changes to `TALKING`, as shown below.



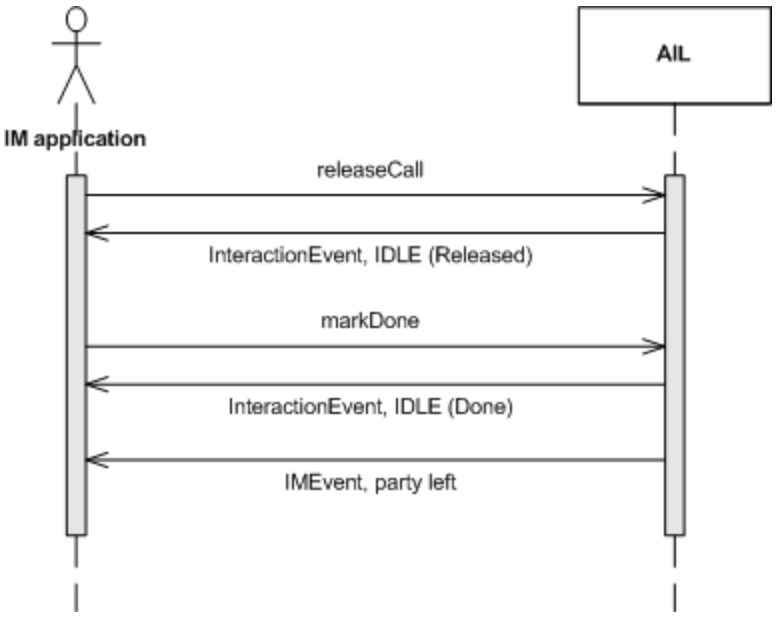
New Instant Messaging Session

Then, when your application receives a message, it gets an `InteractionEvent` and a `PlaceEventIMInteractionContextInfo`.

New Instant Message

Terminate the Instant Messaging Session

To terminate the Instant Messaging session, your application releases the `InteractionVoice` instance. An `IMEvent` notifies your application as a party left, and the interaction can be mark done, as shown here:



Terminating the Instant Messaging Session

SIP Preview

The Agent Interaction SDK offers a preview feature which enables your application to accept or reject incoming SIP interactions. Previously, SIP interactions were incoming in `RINGING` status. The agent using the AIL application had no choice but accept the call, or (in the worst case) terminate it.
The SIP Preview feature solves this issue. If the agent is not willing to process the call, he or she can refuse it and redistribute the call in the system.

> **Important**
>
> The code snippets presented in this section extends the `SimpleVoiceInteraction` sample.

The SIP Preview Interaction

SIP Preview Interactions are similar to standard voice interactions and do not require a specific integration effort. Your application should handle them identically to other voice interactions.
The SIP Preview feature is an addition to the `InteractionVoice` interface, and does not modify the event cycle and the management of the interaction.

Managing a SIP Preview interaction

Instead of receiving an interaction in `RINGING` status, your application receives an interaction in

`Interaction.Status` .`PREVIEW` status.

```
if (event.getStatus() == Interaction.Status.PREVIEW) {

   // Associate sampleInteraction with the event source
   sampleInteraction = (InteractionVoice) event.getSource();
```

As shown in Generalized Example of a Voice State Diagram (Incomplete), the `PREVIEW` status occurs prior to the `RINGING` status.

At this point, the application can accept the call by calling the `InteractionVoice.acceptPreview()` method, and as a result, the interaction status changes to `RINGING`. Your application can then call the`InteractionVoice.answerCall()` to change the interaction status to `TALKING`.

Otherwise, if the application rejects the interaction by calling the `InteractionVoice.rejectPreview()`, the interaction status changes to `IDLE`.

As for standard voice interactions, your application can benefit from `InteractionVoice.Action.ACCEPT_PREVIEW` and `InteractionVoice.Action.REJECT_PREVIEW` enumerate types to check whether the SIP preview feature is available.

```
if(sampleInteraction.isPossible(InteractionVoice.Action.ACCEPT_PREVIEW)
                sampleInteraction.acceptPreview( null, null);
else if(sampleInteraction.isPossible(InteractionVoice.Action.REJECT_PREVIEW)
                sampleInteraction.rejectPreview        ( null, null);
```

If one of these actions is successful, your application receives an `InteractionEvent` as notification of the status change, indicating the new interaction status.

> **Important**
>
> For instant messaging interactions, if your application accepts the interactions, then the interaction status automatically changes to `TALKING` (your application does not need to answer the call).