Agent Interaction SDK Java Developer Guide

Genesys Interaction SDK 7.6.6

12/29/2021

Table of Contents

Agent Interaction SDK Java Developer's Guide

Welcome to the *Agent Interaction SDK Java Developer's Guide*.

- This document introduces you to the concepts, terminology, and procedures relevant to the Agent Interaction (Java API).
- This document provides a high-level overview of Agent Interaction (Java API) 7.6 features and functions, together with software-architecture information and deployment-planning materials.
- This document is valid only for the **7.6.6** release(s) of this product.

The Agent Interaction (Java API) is built around the Agent Interaction Layer library, which presents an API for developing voice and multimedia applications in either client or server modes. Because the library abstracts features of supported switches, your applications are portable across supported switches. Because the library supports connectivity with Genesys Multimedia servers, your applications can combine e-mail, chat, and other interaction management seamlessly with voice interaction management.

Intended Audience

This document, primarily intended for programmers developing Java-based applications for contact center agents, assumes that you have a basic understanding of:

- Computer-telephony integration (CTI) concepts, processes, terminology, and applications.
- Network design and operation.
- Your own network configurations.

You should also be familiar with:

- Java programming.
- Genesys T-Server features, events, and call models.
- Genesys Multimedia features.
- Voice Callback Solution features.

Chapter Summaries

In addition to this preface, this document contains the following chapters and appendixes:

- About Agent Interaction (Java API). This chapter introduces the Agent Interaction (Java API) with an

overview of its design features along with the structure and key concepts of the library API.

- About the Code Examples. This chapter introduces the supplied source code examples.
- Server Applications. This chapter introduces principles to write agent server applications developed on top of the Agent Interaction (Java API).
- Voice Interactions. This chapter introduces Voice API features for working with Interaction and Configuration objects using the `SimplePlace.java` and `SimpleVoiceInteraction.java` examples.
- Switch Facilities. This chapter discusses how to work with switch features.
- PSDK Bridges. This chapter discusses how to use the PSDK Voice and Config Bridges.
- E-Mail Interactions. This chapter covers programming techniques for managing multimedia interactions such as e-mail, and collaborative e-mails.
- Chat Interactions. This chapter covers programming techniques for managing chat interactions, and introduces CoBrowse in the `SimpleChatInteraction` example.
- Open Media Interactions. This chapter covers programming techniques for managing open media interactions.
- Contact. This chapter covers programming techniques for managing contacts.
- Standard Responses. This chapter covers programming techniques for managing standard response information.
- Outbound Service. This chapter covers programming techniques for outbound campaign interaction management.
- Routing Points. This chapter covers programming techniques for routing management, with references to supplied source-code examples.
- Service Status and Connection. This chapter covers programming techniques for administration management of the connected services.
- Voice Callback. This chapter examines the API features that support voice callback.
- Expert Contact. This chapter examines the API features that support an Expert Contact application.
- Additional Details presents details about handling user data, understanding AIL events, and working with library logging as well as your own application's logging.
- The appendix, Voice Sequence Diagrams presents event call flows for voice interactions.

Table of Contents

Voice Sequence Diagrams

Change History

This section lists all the changes between the 7.6.5 and 7.6.6 versions of this document.

Version 7.6.609.00

Page name	State	Additional details

About Agent Interaction Java API	Updated	• New sections
		  • PSDK Application Template Application Block
		• Updated sections
		  • Switch Facilities

API Overview	Updated	• New section: Generics • Updated section: Objects and Events in AIL (Not applicable to all scenarios)
About the Code Examples	Updated	• Updated sections • Required Third-Party Tools

Voice Interactions	Updated	• Updated sections
		  • Voice Interaction Data
		  • Implement a Listener
		  • Connect to AIL
		  • Set up Button Actions
		  • Register Your Application
		  • Add Event-Handling Code
		  • Set up Button Actions
		  • Synchronize the User Interface
		  • Add Event-Handling Code

Page name	State	Additional details
Switch Facilities	Updated	• Updated sections • Workmodes
Chat Interactions	Updated	Updated: Handling Chat Events
Outbound Service	Updated	• Updated sections • Active Campaigns

About Agent Interaction (Java API)

This chapter introduces the Agent Interaction Java API, its components, features, and scope of use.

Library Overview

The Agent Interaction (Java API) lets you build applications to control and manage voice and multimedia interactions issued by, or intended for, a contact center agent.
The 7.6 release is backward-compatible with the 7.x releases. It is backward-compatible with the 6.5.6 release for voice features if your application uses voice features only.

Components

The Agent Interaction (Java API) comprises the following:

- The Agent Interaction Layer (AIL) library, highly portable, is written entirely in the Java language, delivered as a set of `.jar` files.
- The *Agent Interaction SDK 7.6 Java API Reference*, which is an HTML tree in the `docs/` directory of the installed product directory.
- The Agent Interaction SDK Java Examples, which is a set of code examples that exercise some important features of the API, delivered in .zip and .tar.gz format.
- A set of Application Blocks available on the Product CD. See the *Agent Interaction SDK 7.6 Application Blocks Guide* for further details.

AIL Library

The heart of the Agent Interaction (Java API) is the Agent Interaction Layer (AIL) library. The library can be thought of as two parts: a library core and an API.

Library Core

The library core manages connections to Genesys solution components. It is designed to work only with the set of telephone system switches that are described in the *Interaction SDK 7.6 Java Deployment Guide*.
The core maintains connections to components of the Genesys Framework, Genesys Multimedia, Outbound Contact Solution 7.*x*, and other Genesys solutions.
The core internally maintains objects used by your applications. These internal objects are not directly available through the API. Rather, the library maintains an *AilFactory* object, which provides you with access to API objects.

> **Warning**
> You cannot directly use or modify classes and methods in the library core. The Agent Interaction (Java API) is restricted to the use of the library API as described in the *Agent Interaction SDK 7.6 Java API Reference* delivered with this product. See for downloading API references.

Library API

The library API provides access to the features, statuses, and events managed by the library core. It provides a complete set of classes and interfaces to handle data of the underlying Genesys solutions with which you can design your own multimedia application.
The example Java programs that accompany this product illustrate the use of the interfaces for the most commonly used objects and their events (DNs, places, agents, contacts, and so on), as well as interactions for such services as voice and callback.

Scope of Use

The Agent Interaction (Java API) enables you to develop applications for the following purposes:

- Creating a contact-center agent desktop application for Genesys software implementations.
- Integrating Genesys software with third-party software.
- Creating other, specialized applications specific to your needs.

Typical usage scenarios include:

- Managing agent login activity.
- Monitoring agent status.
- Handling e-mail and collaborative e-mail interactions: sending, receiving, replying.
- Handling voice interactions: calling, receiving, callback, outbound.
- Handling chat interactions.
- Handling open media interactions.
- Handling outbound campaign participation.
- Handling collaboration sessions.
- Accessing the Standard Response Library.

Your application can handle any inbound interaction, regardless of media:

- Answer a phone call.
- Accept an e-mail.
- Accept a request for a chat session.
- Accept an open media interaction.

Your application can initiate outbound interactions, regardless of media:

- Make a phone call.
- Send an e-mail.
- Make a preview outbound call.
- Make a predictive outbound call.
- Submit an open media interaction.

The AIL library offers you two primary modes of deployment or application development:

- A stand-alone application client. Your application code binds with the AIL library at runtime.
- A server application in *n*-tier architecture. You can write a server application that binds with the AIL library at runtime, or you can design your application to work within a container, such as Tomcat, to respond to web-browser client requests.

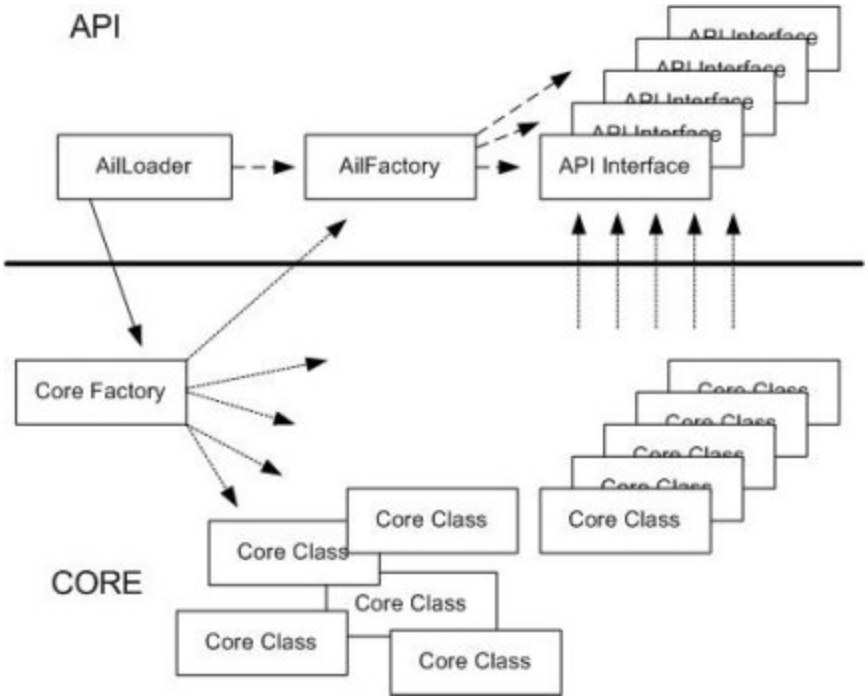
Refer to the *Interaction SDK 7.6 Deployment Guide* for configuration information pertinent to these two modes.

Architecture

The AIL library core is responsible for maintaining connections to servers, maintaining context, managing media, consolidating data, obtaining real-time object information, and providing switch facilities.
The API exposes objects, such as DNs, interactions, and agents, as interfaces that give access to all necessary information. The core manages the objects with respect to state machines that guarantee that the model is coherent with other Genesys components (for example, multimedia solutions) across supported switches. Changes in the object states are available through events.

Interfaces to Core Objects

You do not access core objects of the Agent Interaction (Java API) library directly. Rather, you get interfaces on them using the `AilFactory`.
Your application uses the `AilLoader` class to get an interface to the internal `AilFactory` object. The `AilLoader` class' methods establish connections at application startup, and provide access to the `AilFactory` .
The internal `AilFactory` object is itself available as an interface, through which you access the core Agent Interaction Layer factory object. The `AilFactory` object is a singleton.
Because of its singleton design, only one instance of the core `AilFactory` object exists at runtime. All `AilFactory` interfaces obtained through the `AilLoader` refer to this same object.
The `AilFactory` instantiates internal classes and makes them available through interfaces, as illustrated below. You do not instantiate objects directly by using a `new` instruction. You rather get an interface by calling a `get()` method.



Core Factory Interface

The library provides a reference system of unique object IDs that are standard `String` objects. Thus, your application manipulates each object by passing its unique ID as a parameter in methods of the `AilFactory` interface.

Core Features

The library core is designed around the following features:

- Switch Facilities
- Multithread Implementation
- Synchronization and Error Handling
- Persistence
- Cache Mechanisms
- Back-End Server Connections Management (hot standby, ADDP)

Switch Facilities

The Agent Interaction (Java API) provides a general state model for all media, including voice, which relies on T-Server–switch pairings. Therefore, the library core takes into account switch-specific features on suitable switch facilities. For extensive details about writing applications with respect to switch features, see Switch Facilities.

Multithreaded

The Agent Interaction Layer library is thread-safe and can therefore be run in a multithreaded environment.
On startup, it creates a thread that maintains all Genesys server connections, a thread for processing T-Server requests, a thread for publishing events to the API, and so on. Events from a given T-Server are always forwarded to the API listeners in the same order as they were received from the T-Server.

Synchronization and Error Handling

Usually, communication with the underlying servers is asynchronous. So an acknowledgment or result of a request is returned through the common flow of events sent by a server.
To relieve the developer from managing such asynchronous requests, methods of the API are made synchronous: the server reply is awaited before the requesting method returns.
The API uses exceptions for notification about standard errors, communication errors with the Genesys servers, unavailable actions errors, or status errors.
Moreover, final results of actions are forwarded as events to the registered listeners.
Because the API is synchronized, a request will block the current thread until it returns. For voice applications, the request will block until the request has been completed on the switch or an error occurs.
A request that goes up to the switch or the database might take time. Because this is not what you want in a single-threaded application, you may want to develop a multi-threaded application.
The library implements a timing loop that you can configure in the constructor parameters at the time you create a new `AilLoader` . This is a defense mechanism to allow application recovery following switch, network, T-Server, or other connection failures.
In the event of a failure, an exception is thrown. See the Javadoc API Reference for details.

Persistence

The Agent Interaction Layer library can establish a link with a Contact Server and provides a high-level representation of its objects through the interfaces of the API.
Note that when you have a reference on an object, it can be modified by another thread. For example, a T-Server event can change the status of an interaction. If you call the `getStatus()` method twice on an `InteractionVoice`, the other party might have released the call in between. Then the first `getStatus()` method will return `TALKING` and the second `getStatus()` method will return `IDLE` .
The objects `Contact` and `Interaction` have a representation in the Contact Server and thus can be saved. They implement the `Savable` interface that controls the coherence between the objects and the Contact Server.
These objects can be manipulated with their identifiers.
For a client application to directly retrieve an `Interaction` object from the Contact Server, it can use the `getContactServerId()` method to get the internal DBID of the `Interaction` object.

Cache Mechanism

To improve performance one step further, two different cache mechanisms are implemented:

- An interaction cache on current manipulated interactions.
- A configuration cache for the objects found in the Configuration Layer, such as objects implementing the interfaces `Agent` , `Dn` , or `Place` .

Depending on the way you configure your environment, the configuration cache can be entirely preloaded on startup. This creates a load time proportional to the amount of data in the Configuration Layer.
The configuration cache is fully dynamic: a modification in the Configuration Layer is immediately updated in the cache.

Connectivity and Internal Features

The library core provides the event mechanism, through which your Agent Interaction (Java API) application can notify users about servers' statuses (notably, the loss of a connection).
The library core can maintain connections to multiple T-Servers.
The library core is designed to work in a single-tenant environment. It is possible to create applications that work in multi-tenant environments, but in this case, Configuration Layer objects that your application uses must be specified in the application's Tenants tab (in Configuration Manager), and these names must be unique. See the *Interaction SDK 7.6 Java Deployment Guide* for details.

Framework Compatibility

The Agent Interaction (Java API) connects to the following Genesys servers within the Genesys Framework:

- Configuration Layer—The Configuration Layer stores configuration information (such as application parameters) and objects' descriptions (such as DNs, places, and persons). The library core monitors the Configuration Layer to respond to modifications. The library provides full integration with Genesys Configuration Layer objects such as Agent, Place, and DN.
- Stat Server—this core component keeps track of resource state for your Genesys environment.

> **Important**
> Since 7.6.4, AIL cannot connect to more than one Stat Server.

- T-Server—The Telephony Server handles telephone requests and events by communicating with switches.

For voice-only mode, your application should connect to the Configuration Layer and to at least one T-Server. For details about supported switches, refer to the *Genesys Supported Media Interfaces* document.

PSDK Application Template Application Block

In 7.6.6, Agent Interaction SDK integrates the PSDK Application Template Application Block and provides the following additional features:

- TLS Support for connections to Genesys Servers.
- Policy International Data Support
- Management of AIL dependencies on JDK pluggable service providers.

For further details, read the PSDK Application Template Application Block documentation.

If you wish to replace the default XML parser with the XML JVM options, you will need the following set of JVM options:

```
-Dcom.genesyslab.platform.commons.xml-doc-builder-
factory=com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
-Dcom.genesyslab.platform.commons.xml-transformer-
factory=com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
-Dcom.genesyslab.platform.commons.xml-xpath-
factory=com.sun.org.apache.xpath.internal.jaxp.XPathFactoryImpl
```

AIL also takes those options into account for its internal needs and always uses the SUN XML parser.

Multimedia Compatibility

The Agent Interaction (Java API) is compatible with Genesys Multimedia, and provides full multimedia support for voice, e-mail, chat, and open media interactions.
The Agent Interaction Layer library connects to the following Genesys Multimedia servers:

- Interaction Server. This server manages non-voice interaction information.
- Chat Server. This server manages chat interactions between agents and web visitors.
- Universal Contact Server (UCS). This database server is used to retrieve and store e-mails, history, and contact information. You can manipulate the contact history and the standard response library.

For e-mail handling, your application should connect to a Configuration Layer, and a Universal Contact Server and an Interaction Server (both included with Multimedia).
For chat handling, your application should connect to a Configuration Layer, a Chat Server, an Interaction Server, and a Universal Contact Server (all three both included with Multimedia).
For open media handling, your application should connect to a Configuration Layer, an Interaction Server and optionally a Universal Contact Server (both included with Multimedia).

Outbound Campaign Support

The Agent Interaction (Java API) connects to the Genesys Outbound Solution through the Outbound Campaign Server. This server controls and organizes outbound campaigns.
For outbound campaign handling, your application should connect to, if using voice outbound, to a Configuration Layer, an Outbound Contact Server, a T-Server, and, optionally, UCS. If you are using outbound proactively, you should connect to Interaction Server and at least one T-Server.

Voice Callback Support

The Agent Interaction (Java API) connects to the Genesys Universal Callback Solution through the Callback Server. This server controls and organizes callback records.
For Voice Callback handling, the Agent Interaction (Java API) should connect to a Configuration Layer, a Callback Server, and at least one T-Server.

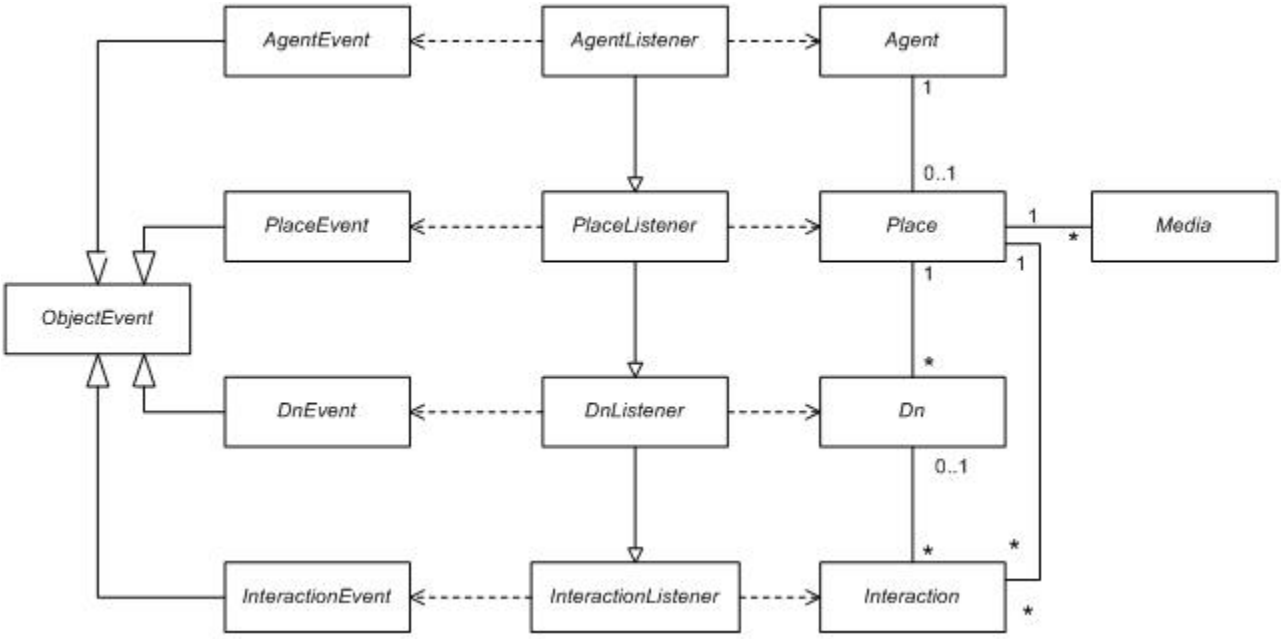
Multi-Tenancy Support

The Interaction SDKs are not suited for multi-tenant deployments. Although you can use them for a given tenant in a multi-tenant environment, you would need a separate instance of your application

for each tenant using it. (As an alternative, the Genesys Platform SDK supports multi-tenancy.)

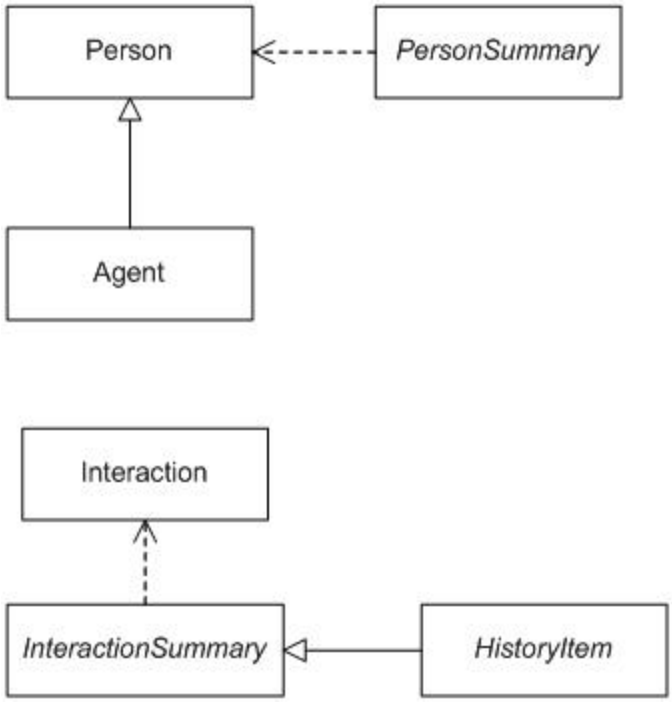
API Overview

The Agent Interaction (Java API) presents a common programmatic interface to working with interactions between agents and customers, or between agents, regardless of media. The API abstracts the Configuration Layer objects and interaction event flow to a generally common model. Your client application design is largely a matter of managing the event flow of an interaction from the agent's arriving to his or her leaving. You do this by implementing event listeners on objects. As events reflect changes in the state of an interaction, your application should test which actions are possible and make method calls accordingly. The following figure shows some commonly used objects that deal with events, including interactions.



Objects and Events in AIL (Not applicable to all scenarios)

To ensure good performance and avoid deadlocks, listeners should not run a time-consuming process, but rather should use a thread to do the work (see the section Implementation).
To optimize the network activity, the API also includes fast access through summary classes which provide lightweight interfaces to commonly required data for main objects. These concepts are illustrated for agent and interactions in the following figure.



Some Lightweight Interfaces for Agents and Interactions

Packages

The Agent Interaction SDK 7.6 Java API Reference (open `index.html` in the product installation directory's `docs/` subdirectory) shows that the API comprises the following packages:

- `com.genesyslab.ail` —Exposes the interfaces and classes for interactions, media, and configuration objects.
- `com.genesyslab.ail.event` —Exposes interfaces and classes pertinent to Event and Listener interfaces for interactions, media, and configuration objects.
- `com.genesyslab.ail.exception` —Exposes the classes and exceptions for errors that occur.
- `com.genesyslab.ail.collaboration` —Exposes interfaces for collaboration features.
- `com.genesyslab.ail.srl` —Contains interfaces to manage standard responses.
- `com.genesyslab.ail.srl.event` —Contains interfaces to manage standard response events.
- `com.genesyslab.ail.workflow` —Contains interfaces to manage workbins.
- `com.genesyslab.ail.monitor` —Contains interfaces to get real-time information about agent status.

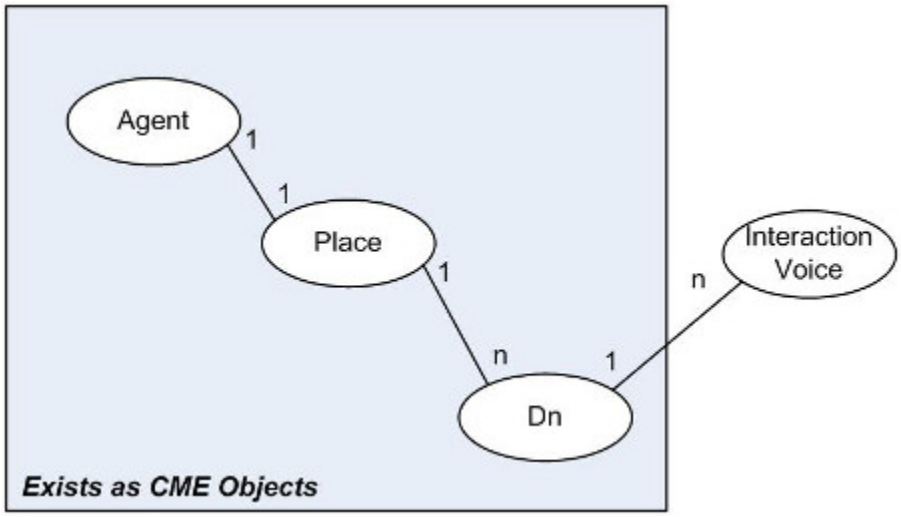
Generics

In 7.6.6, AIL uses Java generics when dealing with collections. If you use non-generic collections, compilation generates warning messages.

```
//Example of change
//in previous release
java.util.Collection getIncomingAddresses()
    throws ConfigServiceException
//Change introduce
java.util.Collection<EmailAddress> getIncomingAddresses()
    throws ConfigServiceException
```

Agents

The `Agent` interface contains the data for a `Person` and allows this person to work in a `Place`. The `Place` is a set of media and DNs. Media include e-mail, chat, and open media. DNs are specific to voice interactions.



The Agent Model

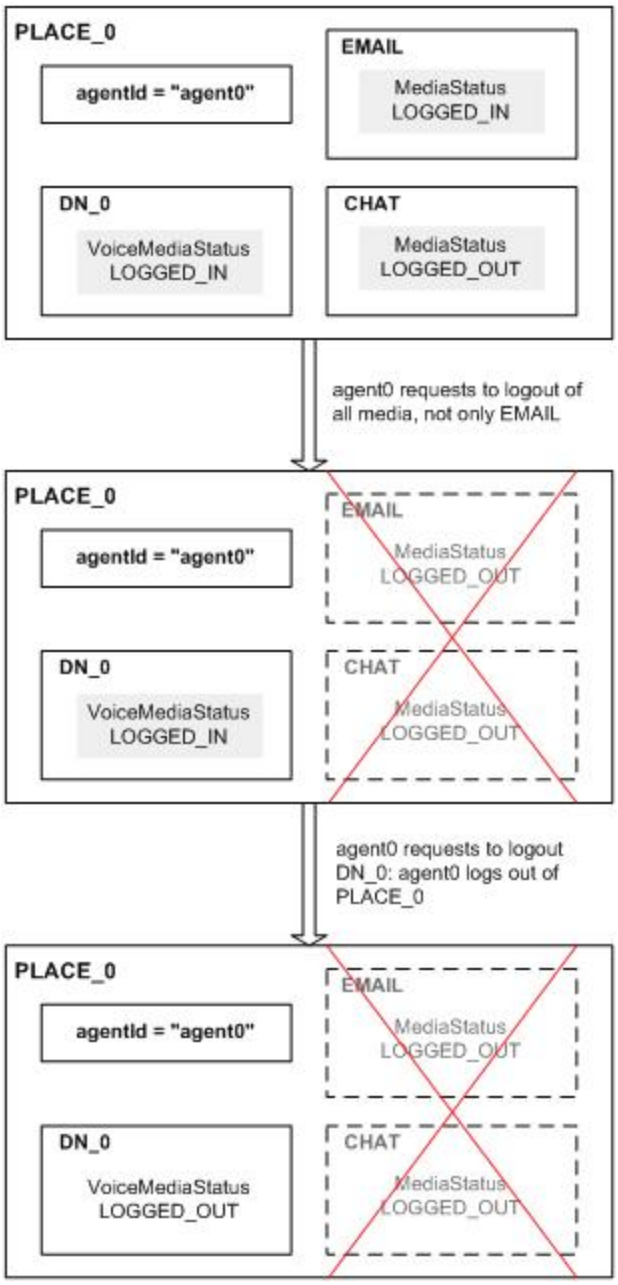
The `Agent` interface includes the following features:

- Access to data for the `Person` corresponding to this agent.
- Management of the agent's activity (login, logout, ready, not ready) on the `Place` and its media.
- Management of the agent's status on the `Place` and its media.
- Access to the `Place`'s features:

- Create interactions according to the available media (including voice).
- Use the outbound and callback services of this place.
- Monitor the agent status on the place's media and DNs.

Place, DNs, and Media

To access CTI features as well as multimedia features, the agent must be logged in. If the agent is logged into one media type or DN of the place, the agent is logged into the place. In this case the place, its DNs, and its media are associated with the agent.
The DNs and medias assigned to a place are defined in the Configuration Layer, and both are managed through separated and distinct methods of the `Agent` and `Place` objects.
At runtime, the medias only exist in the Interaction Server. To get the media of a place, your application must perform a login to at least one media type by calling the `loginMultimedia()` method. As a result, the available media are added to the place, and the ones specified in the request are logged in. If your application logs out media per media, the Interaction Server considers that the place is still logged in. To definitely logout of the place, your application must perform `alogoutMultimedia(null, reason, description)` call, that will logout all medias at once, including in the Interaction Server.
The following figure shows the place `PLACE_0`, where agent `agent0` has logged into a DN `DN_0` and to the place's e-mail media.



Place and Agent Login

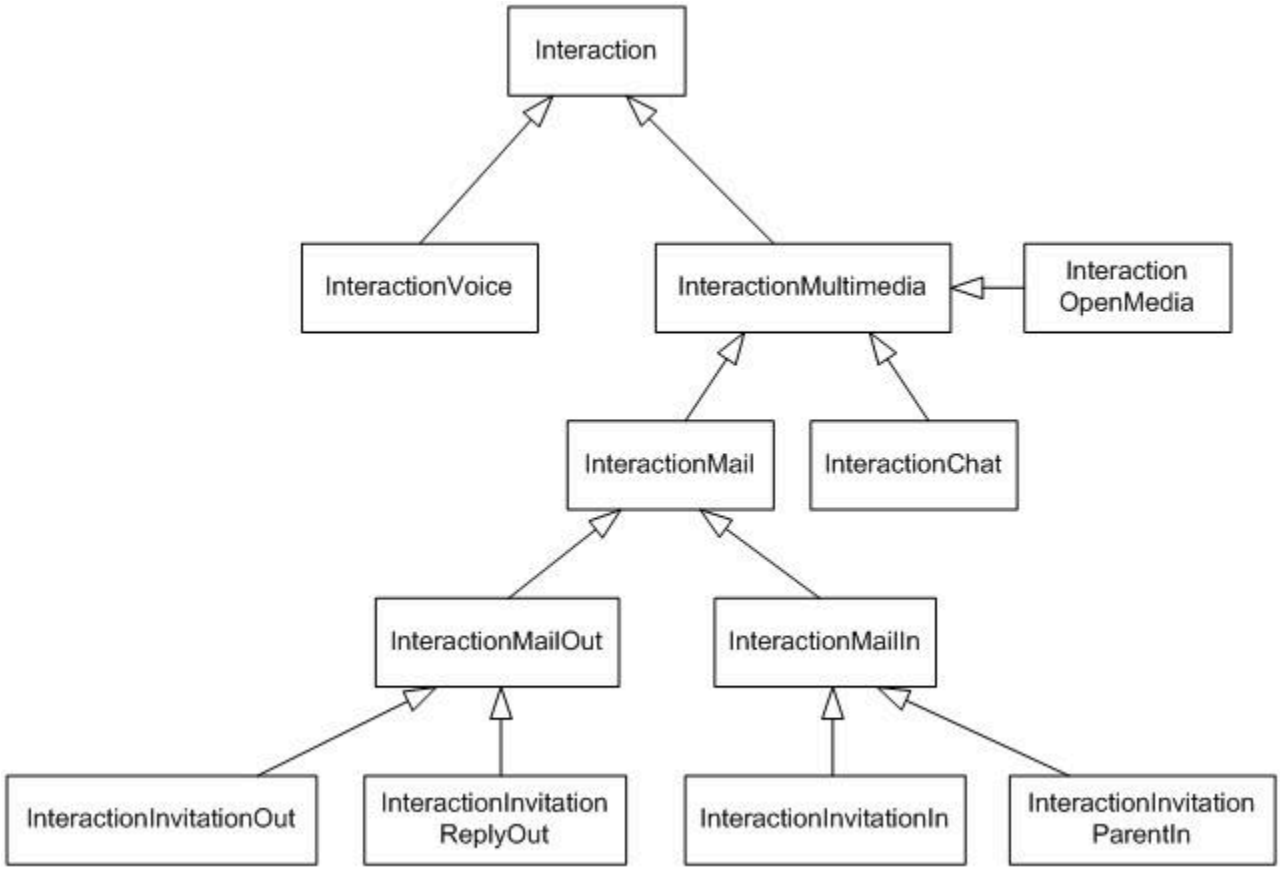
As shown above, if the agent logs out from all the media, they are removed from the place. Once the agent is logged in to the place, that agent has to log out of that place before another agent can use the place.

Interactions

An `Interaction` object represents an interaction between a customer and an agent. This is true regardless of what media the interaction relies on, and regardless of whether a customer or an agent initiated the interaction.

Types of Interactions

Because of its Factory-based design, the API provides interfaces for interactions on all media. These are abstractions of mechanisms and objects found in the Genesys servers, or consolidated views of them. Therefore, they might not exactly match the Genesys objects, but might instead represent them with a higher level of abstraction.
An interaction can be an e-mail, a voice call, a chat session, or any number of other types of media. Several interfaces describe the available interactions, as illustrated below.



Interaction Hierarchy

In the 7.*x* releases, some interactions have disappeared as, for example, `InteractionVoiceCallback`, and `InteractionVoiceOutbound`. New interactions have been added to

offer new features and to consolidate the interaction model. For example, `InteractionMultimedia` is the superinterface for any interaction handled by a media type, and `InteractionInvitation*` are collaborative interactions. For further information, see E-Mail Interactions.
An agent can use interactions according to:

- The underlying media: Interactions depend on the media available on the place. For example, if an agent is logged into a place and its DNs, he or she can use voice interactions. If an agent is not logged into the e-mail media of a place, the agent cannot use any e-mail interactions.
- The underlying servers connected to the factory: According to the type of server connected, the corresponding feature is either available or not. For example, if no interaction server is connected, the agent cannot use e-mail interactions.

Note also that an interaction can be created and manipulated well before it actually exists in the Genesys servers. It may still exist afterwards, as well.

Interaction Characteristics

An interaction has the following characteristics:

- A type that is associated with its interface. For example, `Interaction.Type.EMAILIN` is associated with the `InteractionMailIn` interface describing an incoming e-mail interaction.
- A status that is refreshed in events. The different interaction statuses are available in the `Interaction.Status` enumeration. For example, your application can use the interaction status for informative purposes, or for GUI purposes, such as displaying panels.
- Agent actions that correspond to the methods of the interaction's interface. Your application can determine whether an action is possible at a certain time by calling the methods of the interaction's `Possible` superinterface.
- Attached data, available as key-value pairs. These include Business Attributes defined in the Configuration Layer. See Attached Data for further details.

The `com.genesyslab.ail` package includes an `AbstractInteraction` interface, which is the base for all interactions, including agent and routing interactions.
An interaction's status is made available through the event mechanism, as explained in the following sections.

Events

The event mechanism provides your application with the means of detecting changes in the status or structure of some objects.
Some objects have either a status (such as `RINGING`), or a structural relationship (such as a DN that has an `Interaction`), or both. Because such objects are active, they may change their state themselves. Alternately, another object may change an object's state because of some external event.
In general, there are two techniques by which applications can become aware of changes in other objects: the pull model and the push model.

- In the pull model, the application constantly requests state or relationship changes from objects (pulling).
- In the push model, also known as the event model or callback model, the application implements a well-known method or class as an interface for the objects to call at the time their states or structures change (interrupt).

Event Listeners

The AIL library core provides the push model through the Observer pattern. Objects such as `Interaction`, `Dn`, or `OutboundService` implement this pattern.
The mechanism is that of sending an event on an object to a listener. This permits each object to implement its own set of listeners and methods. Generally, a listener declares only one method, a `handleXxxEvent()` method, that takes an event interface as an inbound parameter. The inbound event interface is highly dependent on the original interface for which it is intended.

Implementation

To receive notifications of events on an object, your application must have a class that implements the listener interface that the object requires. Your application must also register its listener class with the object by using an explicit `addXxxListener()` method on the object interface. As events occur on the object, the library passes event interface objects to the listener's event-handling method with information about changes in the object's state.
The pattern can be defined as follows: An object, `Xxx`, implements the event-listener pattern by defining two methods, the `addXxxListener()` method and the `removeXxxListener()` method, that take as an inbound parameter a reference to an interface `XxxListener`. The `XxxListener` class within the application declares a `handleXxxEvent()` method, which takes an `XxxEvent` as its inbound parameter.
For example, the `Place` interface provides two methods: the `addPlaceListener()` method and the `removePlaceListener()` method. These two methods take an argument that is a reference to a `PlaceListener` interface. Consult the Javadoc API Reference to see which methods the interface `PlaceListener` contains.
Your application must define a class that implements a `PlaceListener` interface. This class must define, at least, a `handlePlaceEvent()` method that takes a `PlaceEvent` argument. (Alternately, your application can reuse an existing class, redefining its method). Then your application registers this listener class by calling the `addPlaceListener()` method on the `Place` object, referencing your `PlaceListener` class as the argument.
When something has changed on the `Place` object, the AIL core directly calls the `handlePlaceEvent()` method of your listener, passing in a `PlaceEvent` reference, and executing the code you defined there. See the Javadoc API Reference for features of the `PlaceEvent`.
To discontinue event communication between the `Place` object and your application, call the `removePlaceListener()` method on the `Place`.
The code you write for a listener should be extremely lean, primarily passing event data to another thread in your application that first determines events and their attributes as they occur, and then takes appropriate actions.
The library propagates some events through several listeners. For example, `InteractionEvent` is sent to the `InteractionListener`, then to the `DnListener` of the `Dn` handling the interaction, then to the `PlaceListener` of the `Place` to which the `Dn` belongs, and finally to the `AgentListener` of the `Agent` logged into the `Place`.
Thus `PlaceListener` inherits from `DnListener`, and `AgentListener` inherits from `PlaceListener`. Eventually, an `AgentListener` receives `AgentEvents, PlaceEvents, DnEvents,` and

`InteractionEvents`.
The table below shows an example of which events the listeners can receive.

Received Events' Example

	Agent Listener	Place Listener	Dn Listener	Interaction Listener	Campaign Listener	Interaction Callback Listener	Interaction Chat Listener	Interaction Outbound Listener
Agent Event	X							
Place Event	X	X						
DnEvent	X	X	X					
Interaction Event	X	X	X	X				
Campaign Event					X			
Interaction Callback Event						X		
Interaction Chat Message Event							X	
Interaction Outbound Event								X

Threading

AIL events are time-ordered and should be published in listeners as soon as they occur to ensure workflow and information consistency. Therefore, the library core manages listeners with respect to the events' order, using a special thread—the Publisher Thread—dedicated to this task. When an event occurs, this Publisher Thread directly calls the registered listeners. It sequentially executes the listeners' code: a listener must return (terminate) in order for the following listener to execute. If a listener undertakes an extended operation, it delays the following processes:

- The propagation of the current event for the following listeners.

- The propagation of new incoming events.

Moreover, a deadlock may occur if a listener waits for the return of an AIL method that itself may be waiting for an event.
If you want to perform an extended treatment, or a treatment making calls to AIL methods, be sure that your application implements such code in a separate thread, as illustrated in the following code snippet:

```
// Avoid:
public void handleXxxEvent(XxxEvent myEvent){
   ///...
   // my treatment
   ///
}

// Prefer:
public void handleXxxEvent(XxxEvent myEvent){
   java.lang.Runnable treatEvent = new java.lang.Runnable() {
      public void run() {
         //...
         // my treatment
         ///...
      }
   }
   java.lang.Thread doTreatment = new java.lang.Thread(treatEvent);
   doTreatment.start();
}
```

The above code snippet shows one example of thread implementation. You should choose the thread implementation that best fits your application requirements.

State and Possible Actions

As activities occur on configuration objects and interactions, the AIL core updates their states and fires events accordingly. But for any one state, your application can make only a limited set of method calls.
For example, at the time an agent starts your application, the agent must first log into a DN. Only after logging in can the agent become ready to work. In other words, an agent cannot become ready if the state of the DN is `LOGGED_OUT`.
Assume you are creating an agent-facing GUI application that implements a set of buttons to allow actions, each button corresponding to a particular appropriate method. Your application should activate buttons only for those actions that are possible with respect to current state, graying out GUI buttons for actions that are not possible. For example, at startup, your application might make only its `login` button active. After the agent has logged in, your application might make its `ready` button and its `logout` button active, graying out the `login` button.
Clearly, your application must test current state before calling any methods. The API features designed to let your application respond to the states of interactions and Genesys objects include:

- An event-listener mechanism that passes information, particularly changes in state, about configuration objects (such as a DN) and interactions.
- A `Status` enumeration class with values for each possible state.
- An `Action` enumeration class with values that match associated methods. For example, the `READY`

action value of a DN is associated with the `ready()` method on an `Agent`.

- An `isPossible()` method that takes an action value as its parameter and returns `true` in the case that your application can call the method associated with the particular action value. The `isPossible()` method is available on objects that inherit from the `Possible` superinterface.

When an agent attempts to log into a DN, your application's event listener for that DN receives an event reflecting the state of the DN. The success of the agent's login is not predictable, so your application must test the state of the DN. If the agent has successfully logged in to the DN, the status of the `Dn` object is `READY` or `NOT_READY`. Your application must use the `Dn`.isPossible() method, passing in the `Dn`.Action.READY value, to test if it is possible to call the `Agent.ready()` method.
In the case that the `dn.isPossible(Dn.Action.READY)` method returns `true`, your application can activate the button that calls the `Agent.ready()` method.

About the Code Examples

This chapter introduces the code examples that accompany this developer's guide. It presents essential design considerations and also some initial tasks that an application undertakes in using the Agent Interaction (Java API).

Setup for Development

When you install Agent Interaction (Java API), be sure that you have the required tools, environment-variable values, and configuration data. See the Interaction SDK 7.6 Java Deployment Guide for details.
The Agent Interaction (Java API) product includes all Genesys libraries and third-party libraries needed for proper operation. See this page for downloading the code examples used in this guide.

Agent Interaction (Java API) Installation Directory

The installation directory contains the following subdirectories:

- The `ConfigLayerTemplates\` subdirectory contains the `.apd` files you must use to create proper client application objects in Genesys' Configuration Layer.
- The `doc\` subdirectory contains the Javadoc API Reference.
- The `lib\` subdirectory contains the `.jar` files for the Agent Interaction (Java API).

Source-Code Examples

Discussions in subsequent chapters of this guide refer to the supplied source-code examples. These examples are provided on the download page both in a `.tar.gz` and in a `.zip` archive file.
The code examples illustrate the use of the most common features of the Agent Interaction (Java API). The examples are not tested and are not supported by Genesys.
When you expand the `76sdk_exmpl_ixn_java-agent` archive file containing the code examples, you will find two directories, which divide code examples into two categories: standalone and server. Each directory contains java source files, as well as batch files and shell scripts designed for compiling and running the examples with reference to that directory structure.
The structure of the `StandAloneExamples` directory tree is:

- The top-level directory contains the following files:
  - `README.HTML` provides instructions for compiling and running the examples.
  - `compile.sh` and `compile.bat` are shell scripts (respectively for UNIX and for Windows) that, with a little editing, you can use to compile the examples. They take a single argument, which is the name of the example you want to compile (without the `.java` extension).
  - `go.sh` and `go.bat` are shell scripts (respectively for UNIX and for Windows) that, with a little editing, you can use to run the compiled examples. They take a single argument, which is the name of the compiled class you want to run.

    - an agentInteraction.properties file (used by the Common class in Common.java).

- The agent/ directory contains the example source files.
- The classes/ directory is where the scripts store or access compiled classes.
- The doc/ directory contains the Javadoc API reference of the code examples.

The AgentServer directory has a different structure and contains a single example. This directory should be copied to the webapps directory of your Jakarta Tomcat server:

- The top-level directory contains the following files:
    - README.HTML provides instructions for compiling and running these examples.
    - .jsp files, that the Jakarta Tomcat Server runs.
- Both source and class files are stored in the WEB-INF/ directory, including shell scripts that you use to compile the example before you launch the Agent Server example.

Required Third-Party Tools

In order to develop applications with the Agent Interaction (Java API), you will need a compiler, such as the one delivered in the Java Platform, Standard Edition, Development Kit (JDK), version 1.7, from Sun Microsystems.
In this guide, JDK 7 was used to compile and run the code examples.

> **Important**
> No JDK prior to 1.7 versions is supported in AIL 7.6.6.

Environment Setup

In order to compile and run, the compiler or the JVM needs access to the libraries of the Agent Interaction Layer. They are located in the lib/ subdirectory of the Agent Interaction (Java API) product installation directory.
Set the following environment variables:

- Specify all Agent Interaction (Java API) .jar files in the CLASSPATH environment variable.
- Specify the location of the Java Runtime Environment in the JAVA_HOME environment variable.

Configuration Data

For the examples provided for this document to work, they need valid configuration data, including connections to servers and configuration objects such as Place, Dn, Agent, and so on.
See the Interaction SDK 7.6 Java Deployment Guide for configuration details. The following items are particularly important:

- The Application Name in the Configuration Layer is used by the AilLoader.

- Your application must use either the `Client` or the `Server` templates.
- Be sure the examples are using correct information for the Configuration Layer host and port, as well as correct information for configuration objects such as agents, places, media, and DNs.

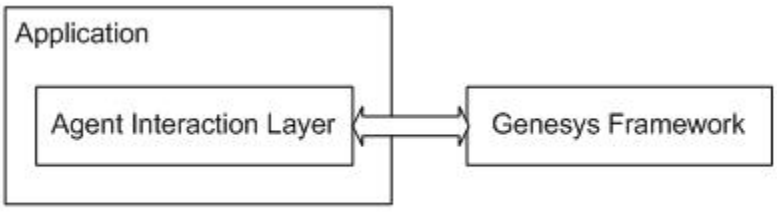
Application Development Design

The AIL library is designed to work across any network that presents TCP/IP access to Genesys servers.
You can create a stand-alone client application or an application service for multiple clients.

- A client application is represented in the Configuration Layer by the client application template.
- A server application is represented in the Configuration Layer by the server application template.

Client Architecture

If the AIL library is used for a client application (for example, for a stand-alone agent desktop application), then as a matter of deployment, the AIL library runs in the same JVM as do the client application code, the GUI, and other related processes. This is illustrated in the following figure.



Client Architecture

Usually, in this case, the AIL library manages only one agent per instance of the application. The library lives as long as the enclosing application.

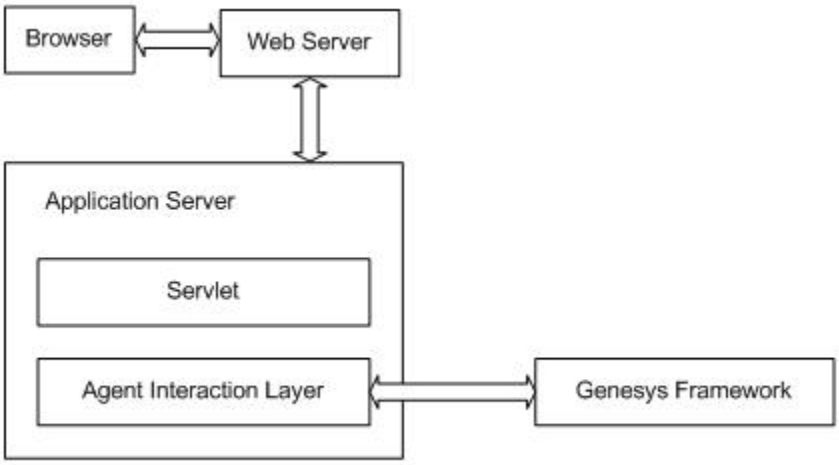
Server Architecture

There are two variations that a server application may take:

- The classic server model, in which the application manages network connections, making itself available on a TCP port.
- The web server model, in which the application and AIL library are embedded in a web container, such as Tomcat.

In server applications in which the library is loaded in a web container, a presentation service instantiates the AIL library, which establishes the connections to the Genesys servers. This is shown

in Web Container Server Architecture.

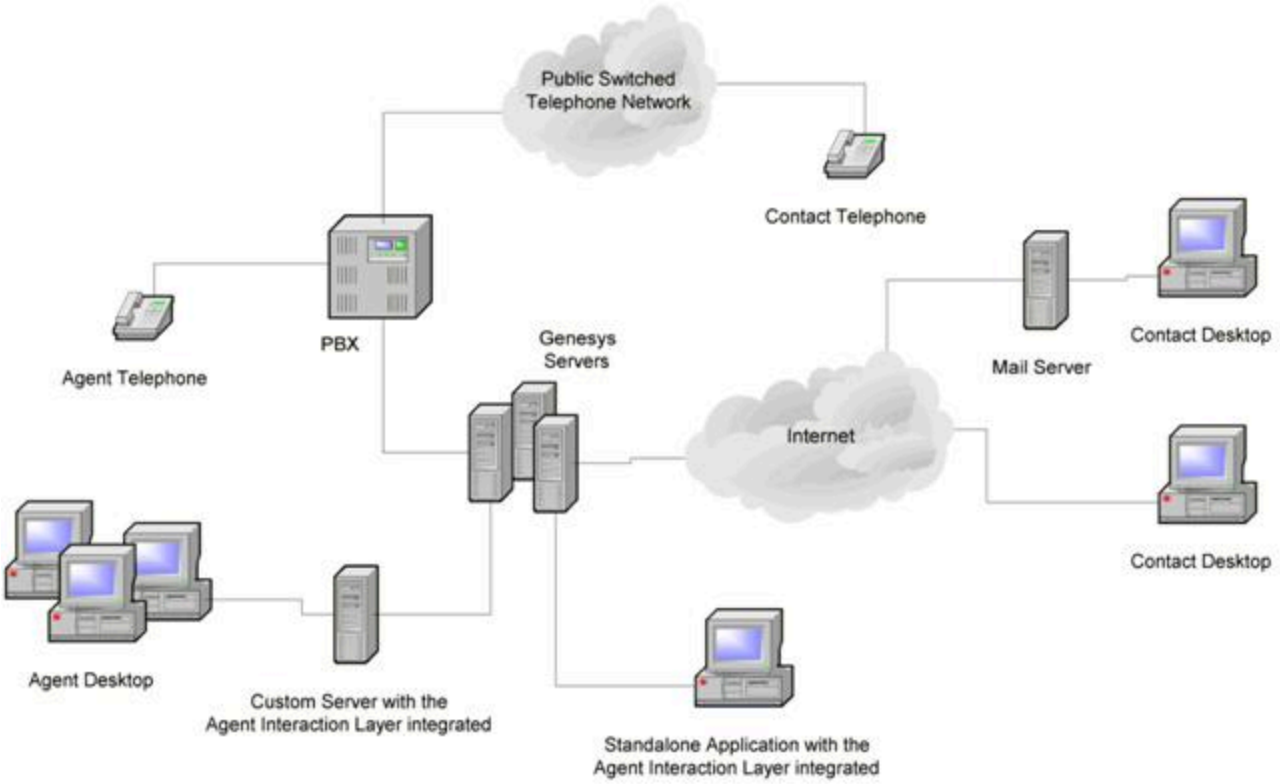


Web Container Server Architecture

Clients access the web server, which requests pages from the web container. A servlet requests data from the AIL library to build a page according to the current states and events.
Either type of server application is likely to handle multiple agents and sessions simultaneously. The AIL library is designed to support such activity. See Server Applications, for details on server development.

Topology

The Agent Interaction (Java API) uses a distributed architecture. It can connect to the Genesys servers, wherever they are, via TCP/IP. This is illustrated in Server Application Topology.
The network topology requirement is that the instance of the Interaction SDK library must be able to connect to Genesys servers.
The Agent Interaction (Java) library core listens to the events coming from such services as the Configuration Layer, the T-Server, the Interaction Server, and so forth. The core manages the status of the different objects, and it filters, preprocesses, and forwards consolidated events to the application using the library.
Event features are similar, whatever the media. Thus, the Agent Interaction (Java) library handles activities of all media in a similar way.



Server Application Topology

Introducing the Standalone Code Examples

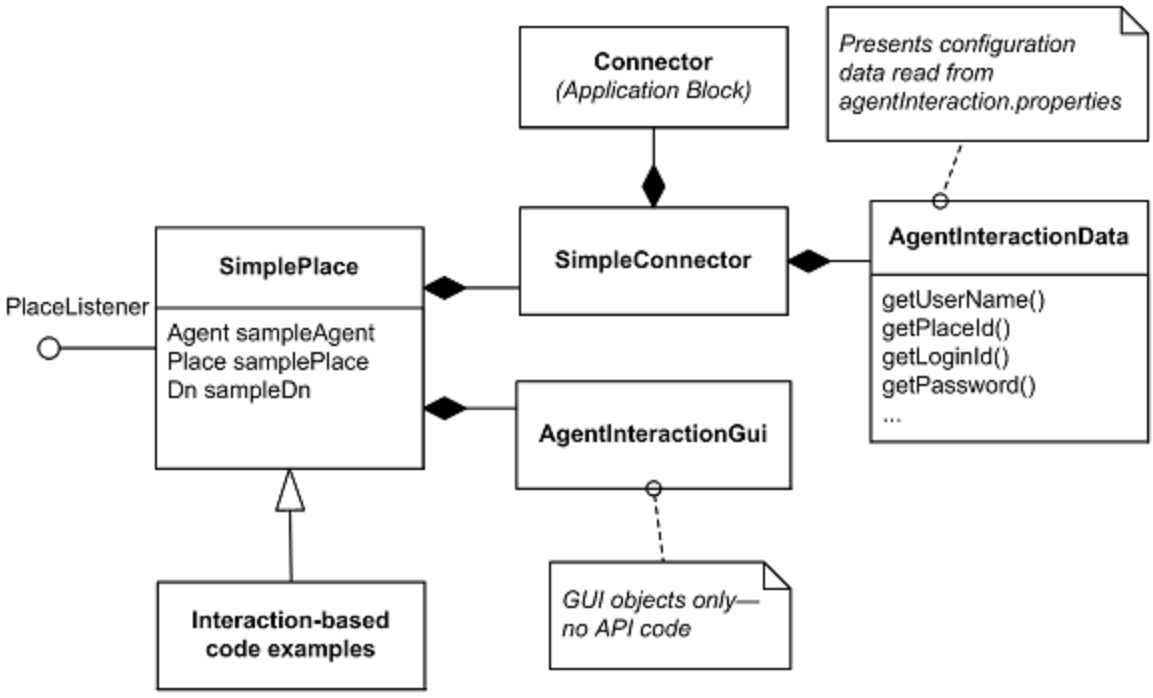
The set of standalone code examples was designed to be interactive. It was also designed to isolate API-related code from presentation code as much as possible. Both of these features should make it easier for you to learn the functionality of the Agent Interaction (Java API).

In order to isolate the API code, separate classes have been set up to read properties information and to create the application's graphical user interface, as shown below.

As you are learning the API functionality, you can ignore the `AgentInteractionData` and `AgentInteractionGUI` classes.

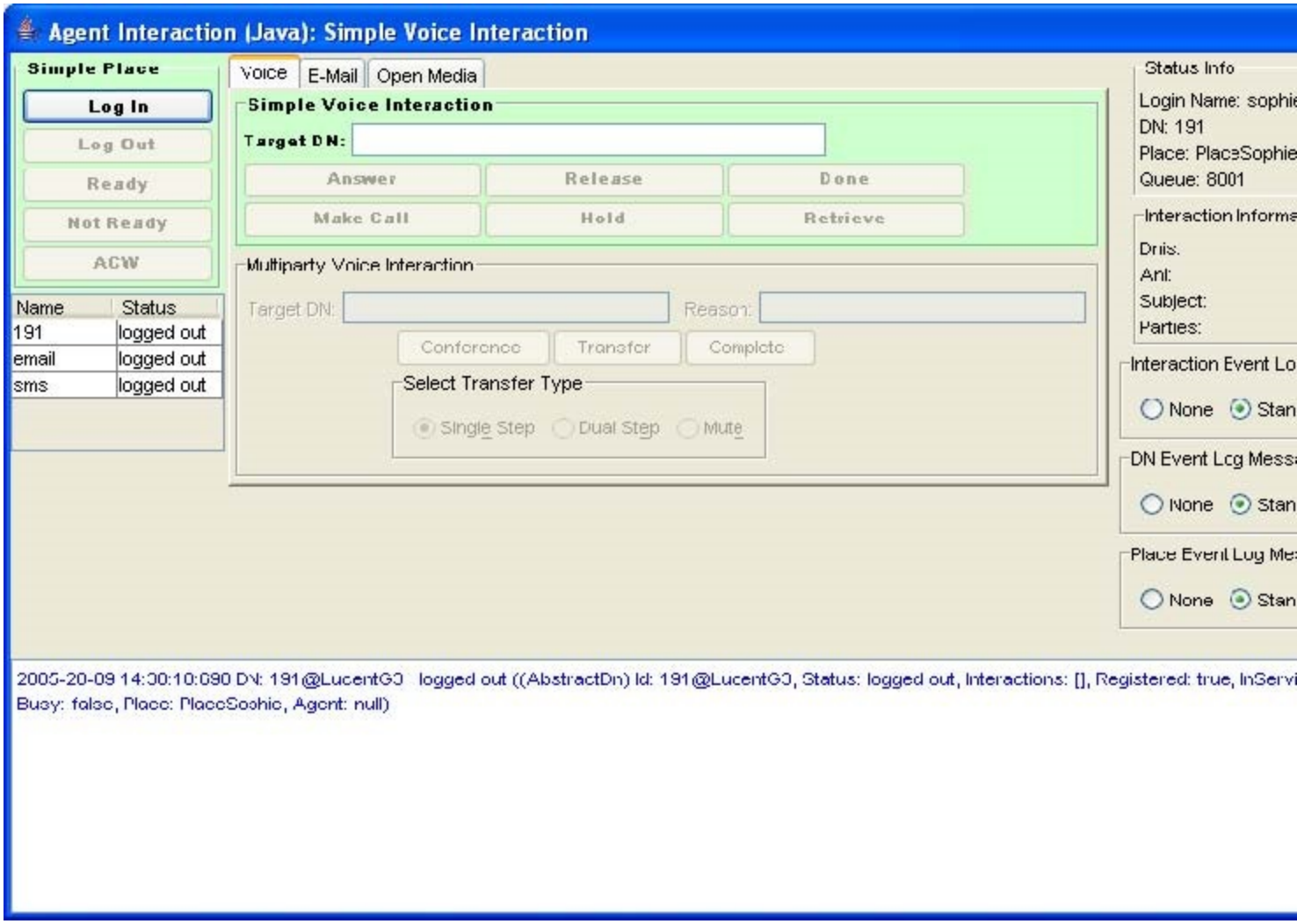
In 7.6, code examples include some of the Agent Interaction Application Blocks for Java, which present best practices for the Agent Interaction (Java API). All examples are built on top of the `Connector` application block that connects to the Agent Interaction Layer.

The examples include the `SimplePlace` class, which is explained in the next chapter. This class implements the `PlaceListener` interface and is the base class for the examples that demonstrate the use of various interaction and event types. `SimplePlace` also calls the GUI class and thereby makes various GUI components, or widgets, available to the examples.



Architectural Overview of the standalone Code Examples

A user interface is available for the standalone code examples. The window title shows the name of the current example, `SimplePlace`. The upper-left corner of the window has a light green background. This shows you which section of the GUI is active for the example you are working on. To the right are four tabs. When you are working on voice, e-mail, open media, or chat examples, the corresponding tab will be selected.



User Interface for the Standalone Code Examples

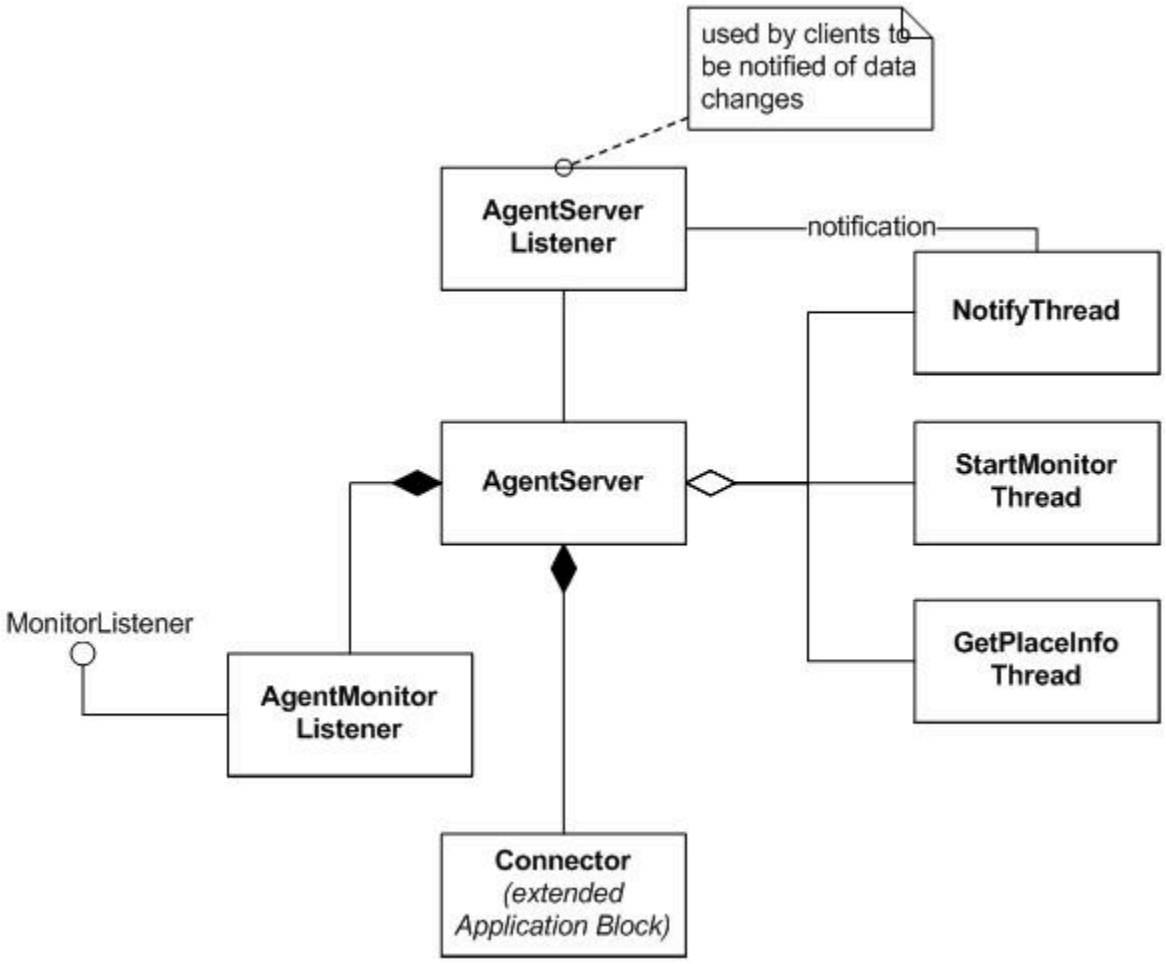
There are two main sections on the right side of the user interface. Status information is provided on the top, and three sets of radio buttons in the middle allow you to control the display of the event messages that appear in the user interface's bottom pane.

The examples generate DN, place, and interaction events. Using the radio buttons, you can display any of them, or none of them. You can also determine how much information you want displayed for each log event.

In order to make the event messages easier to tell apart, they have been assigned their own colors. DN events appear in blue, as shown in User Interface for the Standalone Code Examples. Place events appear in green and interaction events appear in red.

Introducing the Agent Server Code Example

The Agent Server code example was designed to be interactive, and to illustrate a possible implementation of a web server application on top of the Agent Interaction (Java API).
The source code of the Agent Server code example separates API code from the web dynamic code used to display the web graphical user interface, available in a web browser.
The Agent Server java classes include several thread implementation, used to collect or update data, and also to notify the clients with the changes reported by the Agent Server. Also, this code example extends the `Connector` application block, used to connect to the Genesys framework, as shown below.



Architectural Overview of the Agent Server Code Example

The servlet of the Agent Server code example is composed of several `.jsp` pages, that enable the user to send requests to the Agent Server, as shown below.

Web User Interface for Starting the Agent Server

The above interface shows a `.jsp` page used to start the Agent Server. After the server is started, further `.jsp` pages are associated with a `jsp` session that registers for the being notified by the Agent Server. This makes possible to send agent login and logout requests, and also to pull events to get data changes.

Application Essentials

This section discusses some of the essential API features needed for every application. The discussion refers to the `SimplePlace.java` example in the `StandAloneExamples/agent/interaction/samples/` directory, or to the `Connector` Application Block.
Before running this example, be sure to edit the `agentInteraction.properties` file to specify the correct data in your Configuration Layer (host, port, application name, and so on). In addition to compiling `SimplePlace.java`, you will also need to compile Agent Interaction Java Application Blocks, and the following code example files: `AgentInteractionData.java` , `AgentInteractionGui.java` , and `SimpleConnector.java`.
Every application, whether client or server, must use the `AilLoader` class to get a reference to the `AilFactory` , passing correct configuration data arguments.
T he `Connector` Application Block is in charge of this task. `Connector` is a very simple class that shows how to get an instance of the `AilLoader` and then uses the `AilLoader`.getAilFactory() method to get the `AilFactory` interface.

Use AilLoader

The `AilLoader` is a wrapper to the startup process of the AIL library. Its primary goal is to pass configuration information to the core factory and to return an `AilFactory` interface on the `AilFactory` object in the library core.
Use the `AilLoader` object to make connections to Framework components, to set up logging, and to

get a reference to the `AilFactory` . Be sure that your arguments to the `AilLoader` constructor match the data in the Configuration Layer.

Get Configuration Data

Before you create a new `AilLoader` object, you must set or obtain the following minimum configuration data:

- Configuration Layer host name
- Configuration Layer port
- Backup Configuration Layer host name
- Backup Configuration Layer port
- Application login name to the Configuration Layer
- Application login password
- Interaction SDK application mode: either CLIENT or SERVER
- Connection checking time period
- Server request timeout limit

The following code snippet is from the `Connector` class:

```
// Connect to the Agent Interaction Layer
if(applicationParameters != null) {
   mAppParam = applicationParameters;
   mAilLoader = new AilLoader(
      mAppParam.getPrimaryHost(),
      mAppParam.getPrimaryPort(),
      mAppParam.getBackupHost(),
      mAppParam.getBackupPort(),
      mAppParam.getDefaultUsername(),
      mAppParam.getDefaultPassword(),
      mAppParam.getApplicationName(),
      AilLoader.ApplicationType.getApplicationType( mAppParam.getApplicationType().toInt()),
         mAppParam.getReconnectionPeriod(),
         mAppParam.getTimeout());
}
```

Set Up Logging

As you can see from the Javadoc description in the `com.genesyslab.ail` package, the `AilLoader` class includes methods for setting logging.
To set the logging level to debug, use this statement:

```
ailLoader.debug();
```

Likewise, to turn off the console log, issue this statement:

```
ailLoader.noTrace();
```

Similarly, you can issue a method that tells the Agent Interaction Layer not to output log messages to a file:

```
ailLoader.noLogFile();
```

AIL allows you to change the location of your log file. The default log file destination is `./ail.log`. You can specify a different log file directory location and a new filename for the log file, as shown in the `Connector` application block, like this:

```
if(mAilLoader != null) {
    if(file != null) {
        mAilLoader.setDefaultLogFileName(file);
    }
    if(path != null) {
        mAilLoader.setDefaultLogFilePath(path);
    }
}
```

Get a Reference to AilFactory

Use the `AilLoader` class to get a reference to the `AilFactory` interface.

```
// Initialize and return the AIL Factory
ailFactory = AilLoader.getAilFactory();
```

Use AilFactory

If you supply the correct parameters to `AilLoader` , the call to the `AilLoader`.getAilFactory() method triggers the startup process (if the core factory of the Agent Interaction Layer has not yet been instantiated). This is what AIL does:

- Creates the instance of the core Agent Interaction Layer factory.
- Creates and initializes connections to the Genesys Server.
- Initializes the caches.
- Returns an `AilFactory` interface to the core factory.

The `AilFactory` , when instantiated, requests an application name from the Configuration Layer. When the `getAilFactory()` method returns, the Agent Interaction Layer is initialized and ready. (If `NULL`, an error occurs. See `initexception.`)
Through the `AilFactory` interface, you can now instantiate almost all of the objects your application needs. When each object is created, the core assigns it a unique `String` identifier. Retrieving an object's identifier is important if your application works with more than one instance of an object type (for instance, multiple DNs on a place).

Get Application Information

The `AilFactory.getApplicationInfo()` method returns an `ApplicationInfo` object that has methods for retrieving most of the data that the Configuration Layer has about your application. See

the Javadoc description for the `ApplicationInfo` class in the `com.genesyslab.ail` package.

```
ApplicationInfo appInfo = ailFactory.getApplicationInfo();
```

The following code snippet shows some of the information available from `ApplicationInfo`.

```
int appID = appInfo.mApplicationDBID;
String appName = appInfo.mApplicationName;
String appVer = appInfo.mApplicationVersion;
Map appOpts = appInfo.mOptions;
```

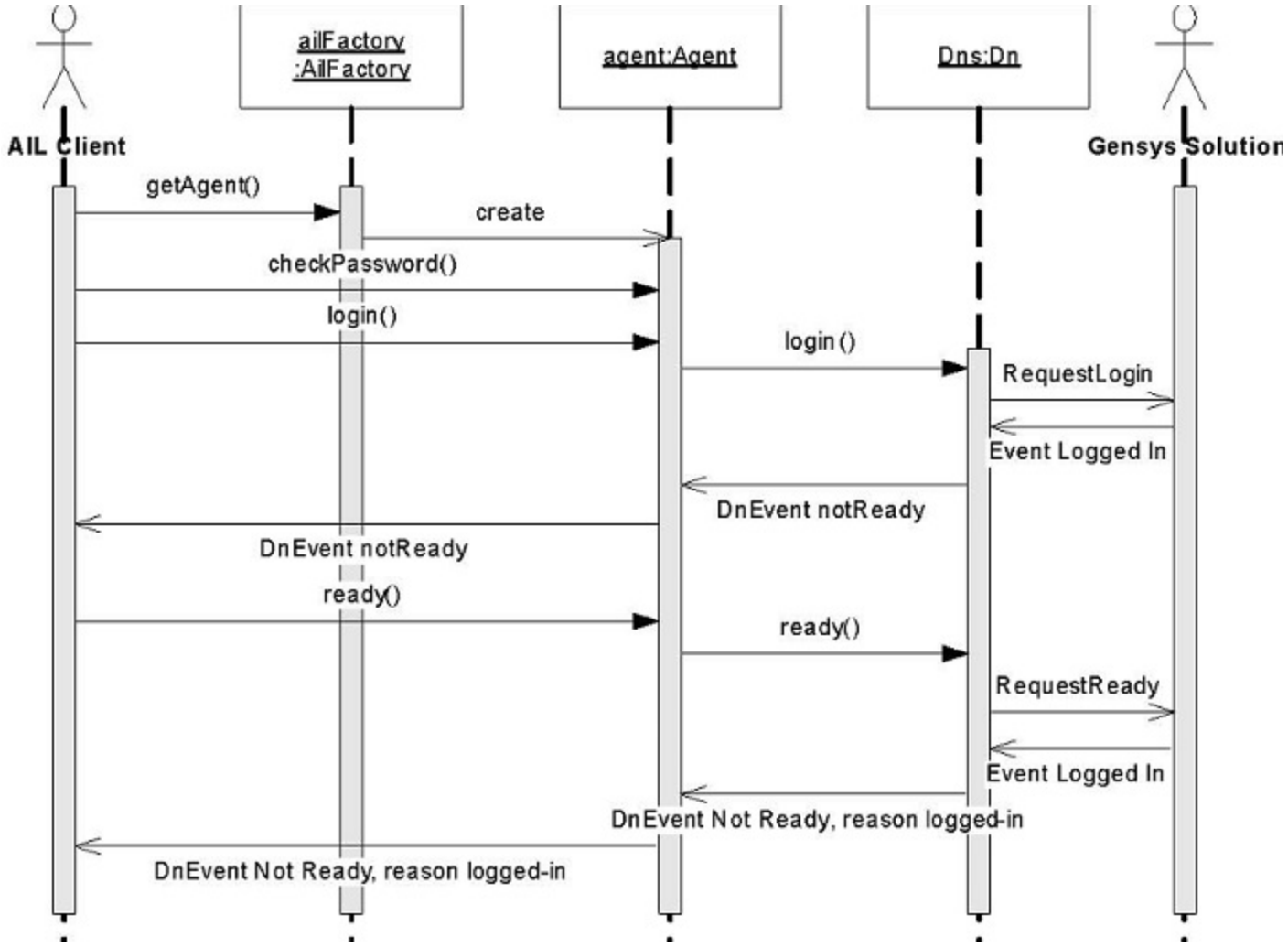
Use Agent

Once you have your `AilFactory` object, you can log in an agent to start working. An agent is a person sitting at a place. A place contains a set of DNs and media that the agent uses to work. The Dn object identifies an access point in a switch that can handle a phone call. The `Media` object identifies a media type that handles multimedia interactions.
The `Agent` and `Place` interfaces have two distinct sets of methods, one for agent activity occurring on the non-voice media of a place and one for agent activity occurring on the DNs of the place.
For example, to have your application associate an agent with a place's media, you would use the `loginMultimedia` method. This method enables you to specify the media types to link to the agent's session.
If, on the other hand, you want to allow the agent to use the place's DNs, your application can either:

- Use the `login` method, which attempts to associate the agent with every one of a place's DNs.
- Retrieve the place's DNs with the `Place.getDns()` method, and attempt to link the agent to a restricted set of DNs.

> **Important**
> Although agents can log into only a single place, they can also be linked to DNs or media types that are not associated with that place.

The event flow for an agent login on the DNs of a place is illustrated below.



Configuration and Logging In

This sequence diagram shows the `DnEvent` events received due to two consecutive agent actions, `login` and `ready`, performed on DNs. Those events propagate the status changes of the DNs as a result of the agents' actions.
Similarly, when performing agent actions on media, your application receives `PlaceEventMediaStatusChanged` events propagating media status changes.
The following code snippet gets an `Agent` object interface from the `AilFactory.getPerson()` method. It passes in a String that names an agent person in the Configuration Layer, and casts the returned person as an `Agent`. The `Agent.checkPassword()` method takes the agent's password as a String and returns `true` if the password is correct.

```
Place place = ailFactory.getPlace(place_name);
Agent agent = (Agent) ailFactory.getPerson(agentname);
if (agent.checkPassword(agentpassword)) {
   agent.login( place, loginId, agentpassword,
      queuename, Dn.Workmode.MANUAL_IN, null); }
```

The Agent.login() method activates the login() methods on the place and its DNs. The arguments to the Agent.login() method are:

- place—Place interface for the Agent's place.
- loginId, agentpassword, and queuename—String names for valid Configuration Layer objects.
- Dn.Workmode.MANUAL_IN—Constant value that sets the workmode for the default DN on the agent's place.
- The final argument, in this case null, is a Map object that stores reasons for the login event on the DN.

After the Agent.login() method successfully executes, the agent is logged in on all the DNs of its default place. The agent must change its status to ready. After the call to the Agent.ready() method, the agent is ready to receive voice events and interactions. This same ready() method is available on the Agent, Dn, and Place interfaces.

Receive Events

To receive events on an object requires a class that implements the appropriate listener interface and implements all the methods for the listener interface. The class must register as a listener on that object. Each event sends an appropriate event object to a well-known listener event-handling method in the class. The code for the handler inspects the inbound event object and takes appropriate action. This follows the Observer Design Pattern, similar to the listeners in the JDK.
Take, for example, the job of tracking event flow on an agent. The API features involved include the Agent interface (in the com.genesyslab.ail package) along with the AgentEvent interface and the AgentListener interface (in the com.genesyslab.ail.event package).

Agent Interface

To work with a particular agent, get an Agent interface from the AilFactory for the agent:

```
Agent mAgent;
mAgent = ailFactory.getPerson(strAgentName);
```

The Agent interface has many methods, but this discussion concerns its addAgentListener() method. The following call registers this class as a listener for events on mAgent

```
mAgent.addAgentListener(this);
```

AgentListener Interface

Create a class that implements the AgentListener interface and implements all of the AgentListener methods. There are several methods on the AgentListener interface: implement methods according to your application requirements. For instance, if your application needs to be updated with interaction status, you should implement the handleInteractionEvent() method. For every interaction event on the agent, this handler method receives an InteractionEvent object, which stores current status and other updated information.
The other AgentListener methods can be empty if you are not interested in inspecting their inbound

event objects.

InteractionEvent Interface

The `InteractionEvent` interface supports a variety of methods; its `getStatus()` method returns the status of the interaction object at the time the event occurred:

```
Interaction.Status getStatus()
```

The handler code should pass the current state and other information (such as the `InteractionId` for the interaction) to another thread that can take appropriate actions. Design your handlers to return as quickly as possible, because the library core works with all handlers sequentially, waiting for each handler to return before working with the next handler.

Get Real Time Information

The `com.genesyslab.ail.monitor.Monitor` interface provides monitoring features for agent status. You can subscribe to an agent, and get real-time information about that agent's status which is available in the `AgentCurrentState` category from the Stat Server.
To get a `Monitor` instance, call the `AilFactory.getMonitor()` method, as shown here:

```
Monitor mMonitor = ailFactory.getMonitor();
```

Then, to monitor status changes, implement a `MonitorListener` class that receives `MonitorEvent` events, as shown in the following code snippet:

```
public class SimpleMonitorExample implements MonitorListener
{
   public SimpleMonitorExample (Monitor exampleMonitor, String object_id )
   {
      //Adding the listener
      exampleMonitor.subscribeStatus(ObjectType.PERSON, object_id,
Notification.CHANGES_BASED, this); }

      public void handleMonitorEvent(MonitorEvent event)
      {
         //Implementation of the listener method
         //...
      }
}
```

The Agent Server code example deals with the `com.genesyslab.ail.monitor` package. For further details, see Agent Server.

Server Applications

This chapter introduces principles to write agent server applications developed on top of the Agent Interaction (Java API).
As explained in About the Code Examples, Genesys is developing two sets of examples. This chapter will detail the server code example that demonstrates these voice interactions. It consists of the following sections:

- Five Rules to Build an AIL Server Application
- Agent Server

Five Rules to Build an AIL Server Application

Now that you have been introduced to the Agent Interaction (Java API), it is time to outline the rules you will need to observe if you wish to develop a server application.
There are five basic things you will need to do in your server applications:

- **Get the AilFactory singleton.** When your application gets the reference on this factory, your application should test whether it is `null` to get the corresponding exception, as shown in the `getAilFactory()` method of the `Connector` application block.

```
mAilFactory = AilLoader.getAilFactory();
if(mAilFactory == null) {
   ServiceException _es = AilLoader.getInitException();
   throw new RequestFailedException("AilFactory is not initialized " + _es);
}
```

> **Important**
>
> Using the Connector application block ensures that your server application properly handles connection.

- **Manage the AilFactory singleton.** At runtime, your application deals with a unique instance of the `AilFactory`. If you need to restart the AIL library and its connections to Genesys servers, first kill your instance of `AilFactory` by calling the `AilLoader.killFactory()` method, as shown in the `release()` method of the `Connector` application block.

```
mAilLoader.killFactory();
```

If this method call succeeds, you can get a new reference on the `AilFactory` singleton as previously detailed in this section.

> **Important**
>
> Genesys recommends the use of ailLoader.getFactory() in your AIL client application (instead of having a reference to the singleton throughout the code). This decreases the risk of reference issues associated with killFactory usage.

- **Keep references on AIL objects.** If your server application gets a reference on an AIL object, for example a `Place` instance by calling the `AilFactory.getPlace()` method, this instance exists as long as you keep its reference alive. When the reference no longer exists, the object is garbage-collected. In terms of performance, if your application is likely to use this object often, your application should keep a reference to it to avoid having to rebuild the instance. Building AIL objects is time consuming, as it requires collecting data from Genesys servers.
- **Implement multi-threading for event-handling.** As explained in Threading, a `Publisher` thread ensures that AIL events are published sequentially with respect to their time order. So, in listeners' methods, your server application should use multi-threading to process events, and thereby avoid deadlocks.
- **Implement a PlaceListener or an AgentListener**. If your server application listens to a place, you are sure to get all interaction, DN, and media events that occur on the place.

The `Agent Server` code example has been designed to provide you with a very simple server that makes stand out two of these rules, that is, multi-threading implementation and `AilFactory` management through the `Connector` application block.
Now it is time to see how they are implemented in the `Agent Server` example.

Agent Server Example

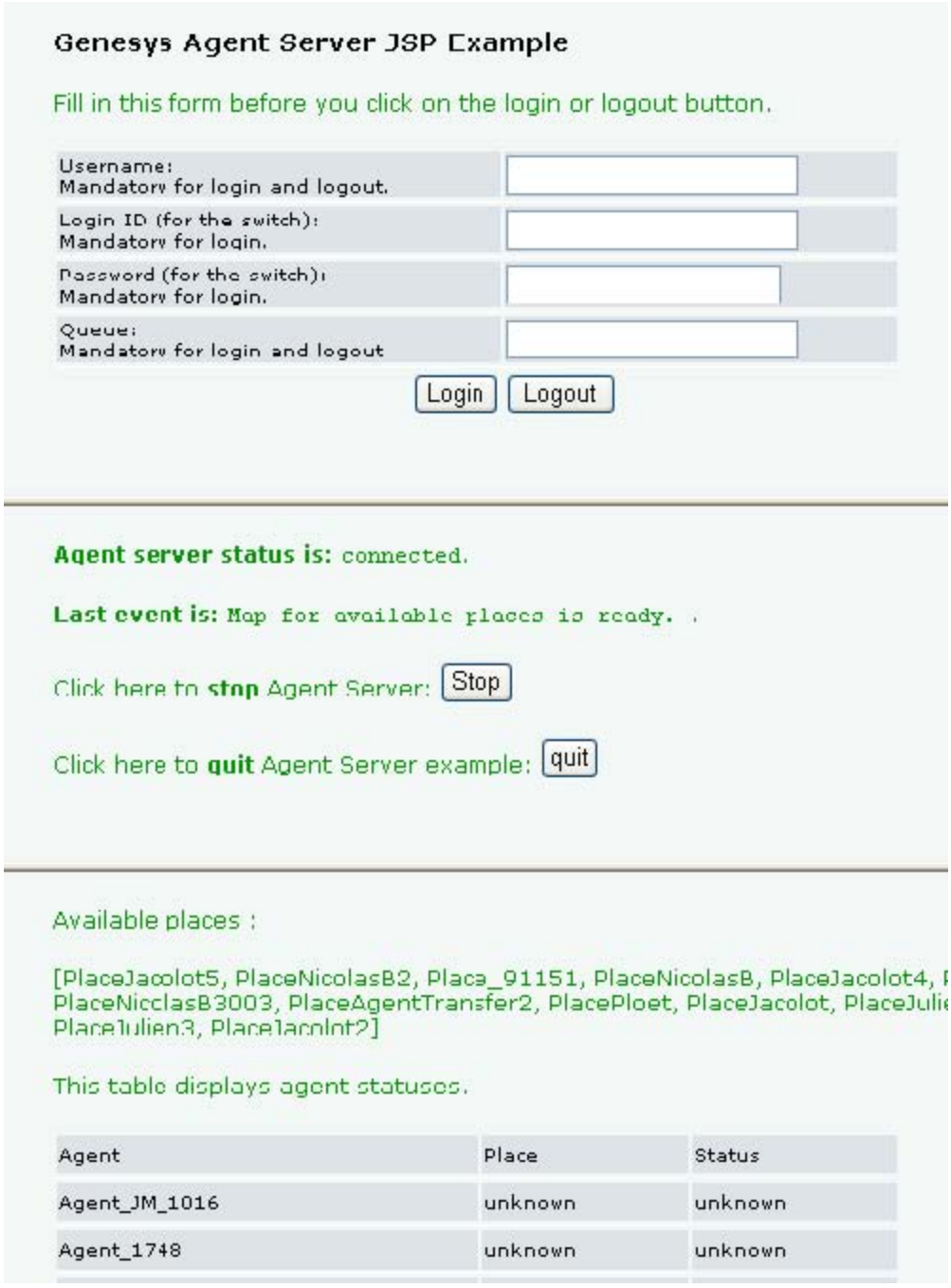
The `Agent Server` code example contains two types of files to be installed on a Tomcat server:

- The java source files, used to build the server application, as shown in the Architectural Overview of the Agent Server Code Example.
- The JSP files, which compose the servlet part of this example. They provide clients with a GUI and manage both client sessions and requests for the server.

When the user loads the `main_frame.jsp` page in a browser, a form appears to launch the server if it is not started, as shown in Agent Server at Startup.

Agent Server at Startup

When the Agent Server is running, it provides the JSP client application with agents' monitoring status. The JSP client application displays these status, registers for monitoring changes, and includes a frame that can send login and logout requests to the Agent Server, as shown in Agent Server is Started.



Agent Server is Started

Because the `AgentServer` example uses classes of the `com.genesyslab.ail.monitor` package to monitor agent status, this example does not deal with `PlaceListener` or `AgentListener` classes.

Connect to AIL

The Agent Server example uses an extended `Connector` application block to perform AIL connection. This extension consists in adding a few lines of code to the inner `ServiceListenerAdapter` class of the `Connector` application block, in order to start monitoring agent status as soon as the statistic service is available.
The following code snippet show the source code added to the `ServiceListenerAdapter.handleServiceStatusChanged()` method.

```
/// Added source code specific to AgentServer example
if(service_type == ServiceStatus.Type.STAT && agentServer != null)
{
   if(service_status == ServiceStatus.Status.ON)
      agentServer.setMonitorListener(true);
   else
      agentServer.setMonitorListener(false);
}
```

> **Important**
>
> The extended `Connector` application block corresponds to the `agentserver.Connector` class of the `Agent Server` code example.

Implement Multi-Threading

The `AgentServer` class implements three inner threads to handle Agent Interaction (Java API) events:

- `GetPlaceInfoThread` to collect place data.
- `StartMonitorThread` to start monitoring agent status.
- `NotifyThread` to handle monitor events.

Collect Place Data

At the server's startup, the `AgentServer()` constructor retrieves two types of data:

- The list of available agents to build a map that client application will display.
- The list of default places for further login actions.

```
//Get agent summaries to build the status map
buildStatusMap();
//Get default places for future login actions
GetPlaceInfoThread p = new GetPlaceInfoThread();
p.start();
```

To get the list of available agents, the `buildStatusMap()` method retrieves agent summaries by calling the `AilFactory.getAgentSummaries()` method. This method returns light objects and is not time-consuming.

```
statusMap= new HashMap();
Iterator itAgents = factory.getAgentSummaries().iterator();
while(itAgents.hasNext())
{
   String id = ((PersonSummary) itAgents.next()).getId();
   statusMap.put(id, new String[]{"unknown","unknown"});
}
```

Because there is no method to get a list of places in the Agent Interaction (Java API), the `AgentServer()` constructor runs a **GetPlaceInfoThread** thread to get an `Agent` instance for each agent of the status map and retrieve the associated default place name, if it exists. This operation is time-consuming because the AIL library has to build numerous `Agent` and `Place` instances, which are not light objects.

```
class GetPlaceInfoThread extends Thread {
   public void run()
   {
      //...
      Iterator itAgents = statusMap.keySet().iterator();

      while(itAgents.hasNext())
      {
         String agentId = (String) itAgents.next();
         Agent myAgent =(Agent) factory.getPerson (agentId);
         Place p = myAgent.getDefaultPlace() ;
         if(p!= null)
            placeVector.add(p.getId());
      }
   }
}
```

Start Monitoring Agent Status

The `ServiceListenerAdapter.handleServiceStatusChanged()` method calls the `AgentServer.setMonitorListener()` method to start monitoring agent statuses if the statistic service is available.
This method creates a `StartMonitorThread` thread which retrieves person summaries and registers an `AgentMonitorListener` for each agent, as shown here:

```
Iterator itAgents = factory.getAgentSummaries().iterator();
monitorListener = new AgentMonitorListener();

while(itAgents.hasNext())
{
   PersonSummary itAgentSummary = (PersonSummary) itAgents.next();
   try
   {
      monitorManager.subscribeStatus(IdObject.ObjectType.PERSON, itAgentSummary.getId(),
monitorManager.getChangesBasedNotification(1), monitorListener);
   }
   catch(RequestFailedException __e)
   {
      System.out.println(__e.getMessage());
   }
}

dataAvailable = true;
notifyListeners("Agent Server is monitoring agent information.");
```

For further details about the `AgentMonitorListener` implementation, see below.

Handle Monitor Event

The AgentServer class implements the MonitorListener interface to handle MonitorEvent events that occur on monitored agents, as shown here:

```
class AgentMonitorListener implements MonitorListener
{
   public void handleMonitorEvent(MonitorEvent event)
   {
      System.out.println("Event: "+event.getClass().toString());
      if(event instanceof MonitorEventAgentStatus)
      {
         MonitorEventAgentStatus agentEvent = (MonitorEventAgentStatus) event;
         NotifyThread th = new NotifyThread(agentEvent);
         th.start();
      }
   }
}
```

When a MonitorEvent occurs, the AgentServer instance creates a NotifyThread which updates the agent status map and notifies all registered clients with this event, as shown in this code snippet:

```
class NotifyThread extends Thread {
   MonitorEventAgentStatus agentEvent;
   public NotifyThread(MonitorEventAgentStatus m_agentEvent)
   {
      agentEvent = m_agentEvent;
   }
   public void run()
   {
      MonitorEventAgentStatus.AgentStatus agentStatus = agentEvent.getStatus();
      HashMap status = getAgentStatusMap(0);
      status.put(agentEvent.getUserName(), new String[]{agentEvent.getPlaceId(),
agentStatus.toString()});
      notifyListeners(agentEvent.toString());
   }
}
```

The AgentServer.notifyListener() method parses the content of the listenerVector vector which contains the listeners that registered to get notified of this event.

```
Iterator itListeners = listenerVector.iterator();
while(itListeners.hasNext())
{
   AgentServerListener listener = (AgentServerListener) itListeners.next();
   listener.handleEvent(msg);
}
```

Submit Login Requests

The AgentServer class does not requires a place name to perform login actions. It uses the list of default places built at startup by the GetPlaceInfoThread thread to perform a login action (see Collect Place Data).

```
String availablePlace = (String) placeVector.get(0);
String log_message = new String( );
```

```
Place m_place = factory.getPlace(availablePlace);
m_agent.login(m_place,loginId, password,
queue, Dn.Workmode.MANUAL_IN,null,null);
placeVector.remove(availablePlace);

notifyListeners(availablePlace+" is no longer available.");
```

Wrapping up

Previous sections list how the Agent Server class implements Agent Interaction (Java API) to manage data and events. Now, let's see how client applications, that is, servlets made of JSP files, interact with the `AgentServer` instance.
The main page of the JSP client application is `main_frame.jsp` . If the server is not started, it loads the `startServerForm.jsp` page which displays a form to be filled in (refer to Agent Server at Startup). When the user clicks on the `Submit` button, the `start.jsp` page creates a new `AgentServer` instance, as shown here:

```
//source from start.jsp
AgentServer agentServer = new AgentServer(configServerHost,configServerPort,
userName,userPassword,applicationName);
application.setAttribute("agentServer",agentServer);
```

Then, the `main_frame.jsp` page can create a client session to connect to the `AgentServer` instance, and loads four JSP pages in frames.

```
//source from main_frame.jsp
<frame src="loginForm.jsp"/>
<frame src="stopServerForm.jsp"/>
<frame src="displayStatus.jsp"/>
<frame src="pullJob.jsp"/>
```

The `loginForm.jsp` page displays a GUI to fill in before submitting a login or logout request. According to the clicked button, it runs the`login.jsp` or `logout.jsp` page, which submits a login or logout request to the server.

```
//source from login.jsp
String msg = agentServer.login(username,loginID,password,queue);
```

The`StopServerFrom.jsp` page displays event information and includes buttons to quit the client application or to stop the Agent Server. When the user clicks the `Stop` button, the `stop.jsp` page is loaded an stops the Agent Server, as shown here:

```
//source from stop.jsp
agentServer.stopAgentServer();
```

The displayStatus.jsp page is in charge of displaying agent monitoring information. Because the agent and place information is not available when the server starts, the servlet tests whether it can retrieve data by calling the`agentServer.isDataAvailable()` method as shown in the following code

snippet:

```
// source from DisplayStatus.jsp
HashMap agentStatusMap = null;
Vector placeVector = null;
if(agentServer.isDataAvailable())
{
   agentStatusMap = agentServer.getAgentStatusMap(index);
   placeVector = agentServer.getPlaceVector();
}
```

To get events (including monitor events), the client application registers an `AgentServerListener` by calling the `AgentServer.addListener()` method.

```
// source from DisplayStatus.jsp
AgentServerListener listener = (AgentServerListener)
session.getAttribute("agentServerListener");
if(listener == null)
{
   listener = new AgentServerListener();
   agentServer.addListener(listener);
   session.setAttribute("agentServerListener", listener);
}
```

The `pulljob.jsp` page is in charge of pulling events every second, as shown here:

```
// source from pulljob.jsp
while(! event)
{
   //...
   AgentServerListener listener = (AgentServerListener)
session.getAttribute("agentServerListener");
   if(listener != null)
   {
      event = listener.gotEvent;
      String msg = listener.getEvent();
      if(msg != null)
      {
         session.setAttribute("event",msg);
      }
   }else
      break;
      Thread.sleep(1000);
   }
}
```

When the `pulljob.jsp` page pulls an event, the application reloads the `main_frame.jsp` page to refresh all frames.

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).

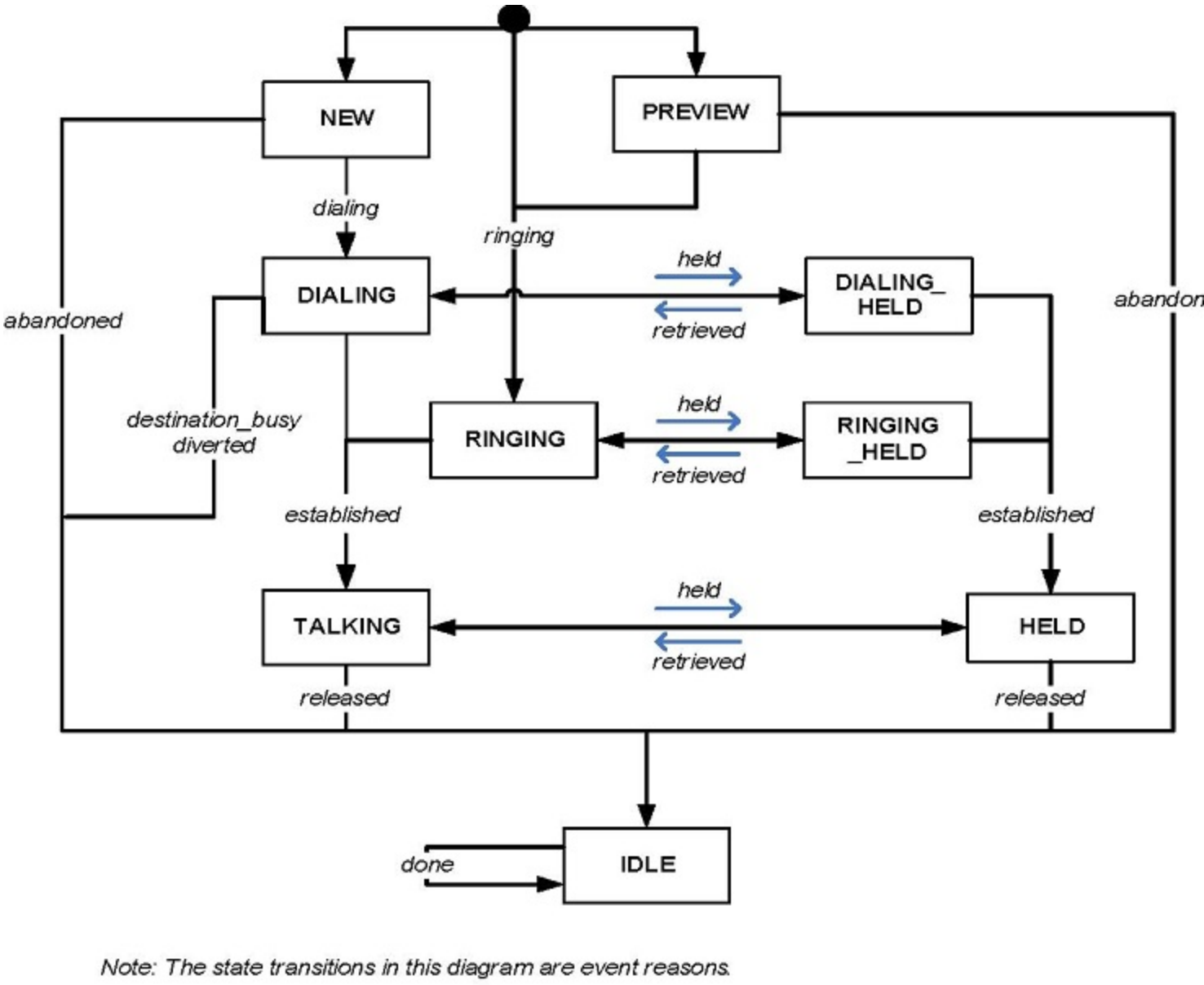
> **Important**
>
> Do not rely on event reasons to update your application; instead, use a refreshed list of possible actions. Event reasons, while they may change the value of what is possible, are primarily for information purposes.

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:

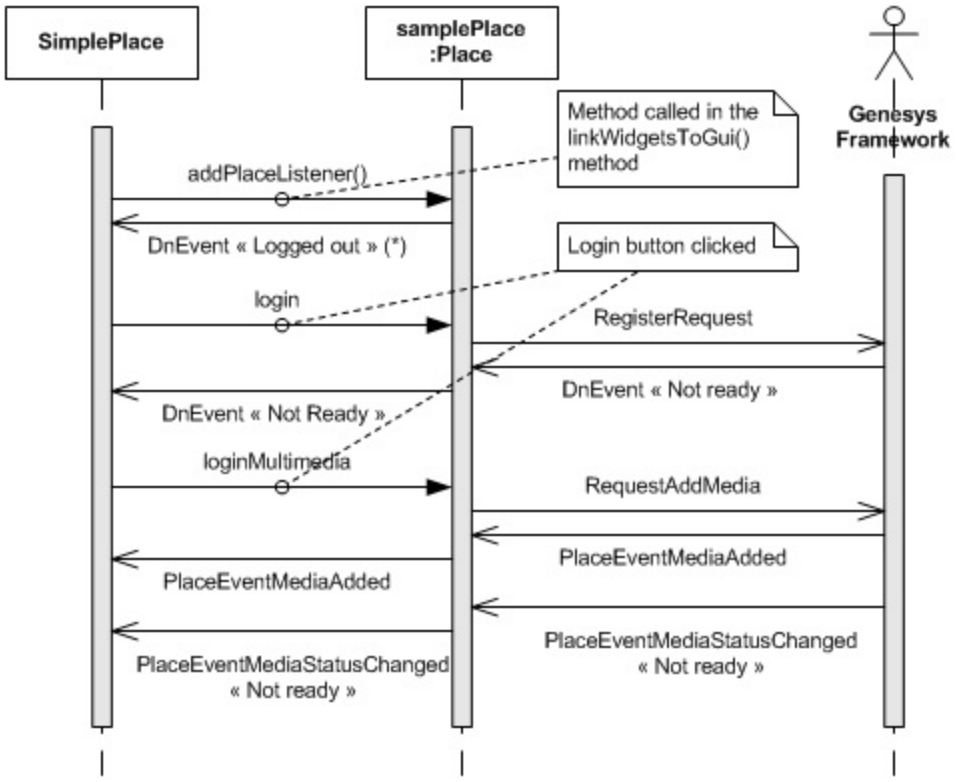
```
// THIS IS AN IMPORTANT STEP:
// As the status changes, enable or disable the buttons
setPlaceWidgetState();
setInteractionWidgetState();
```

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.

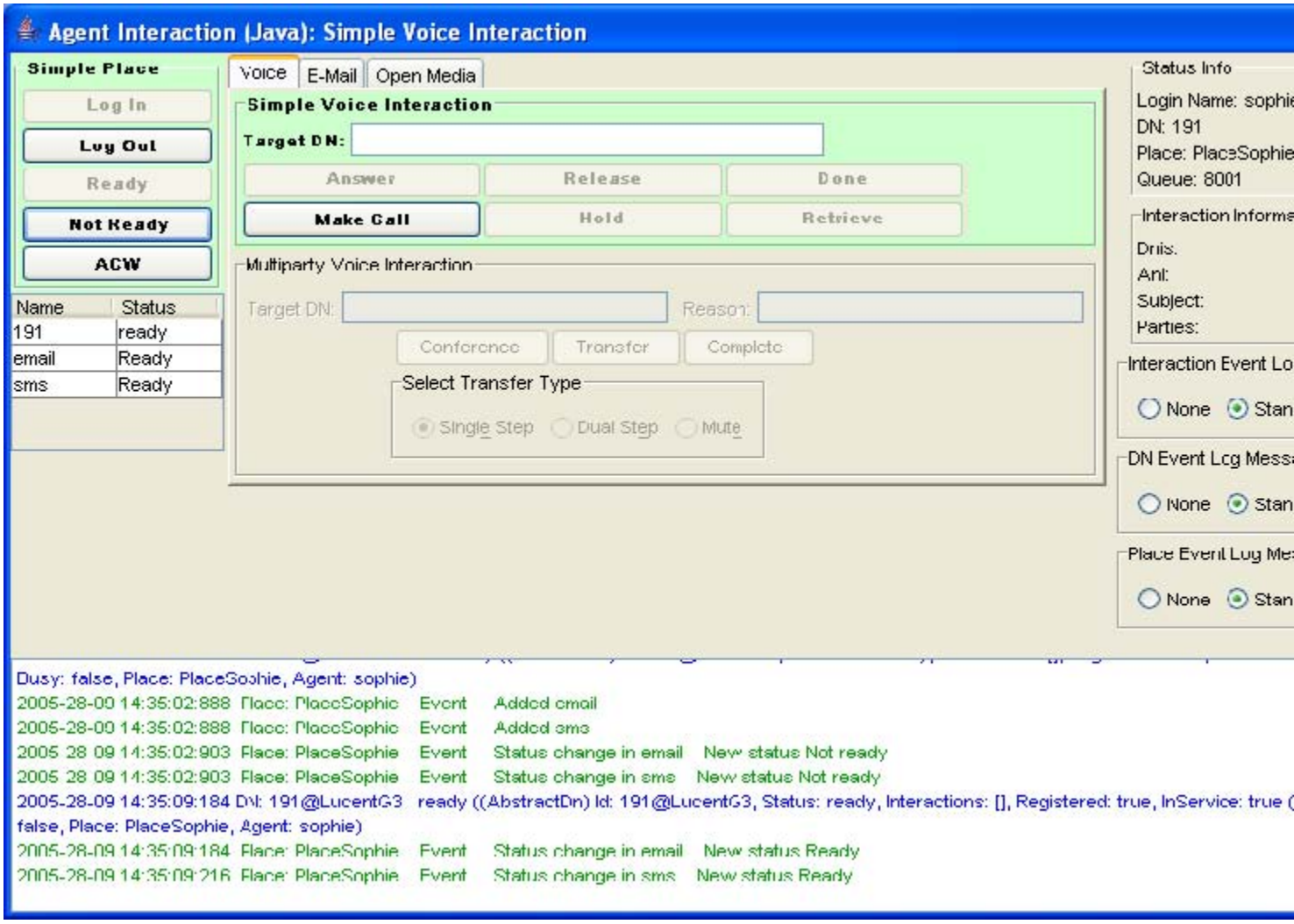
```
// Start the application
agentInteractionGui.mainGuiWindow.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE);
agentInteractionGui.mainGuiWindow.addWindowListener(connector);
agentInteractionGui.mainGuiWindow.pack();
agentInteractionGui.mainGuiWindow.setVisible(true);
```

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("");
}
```

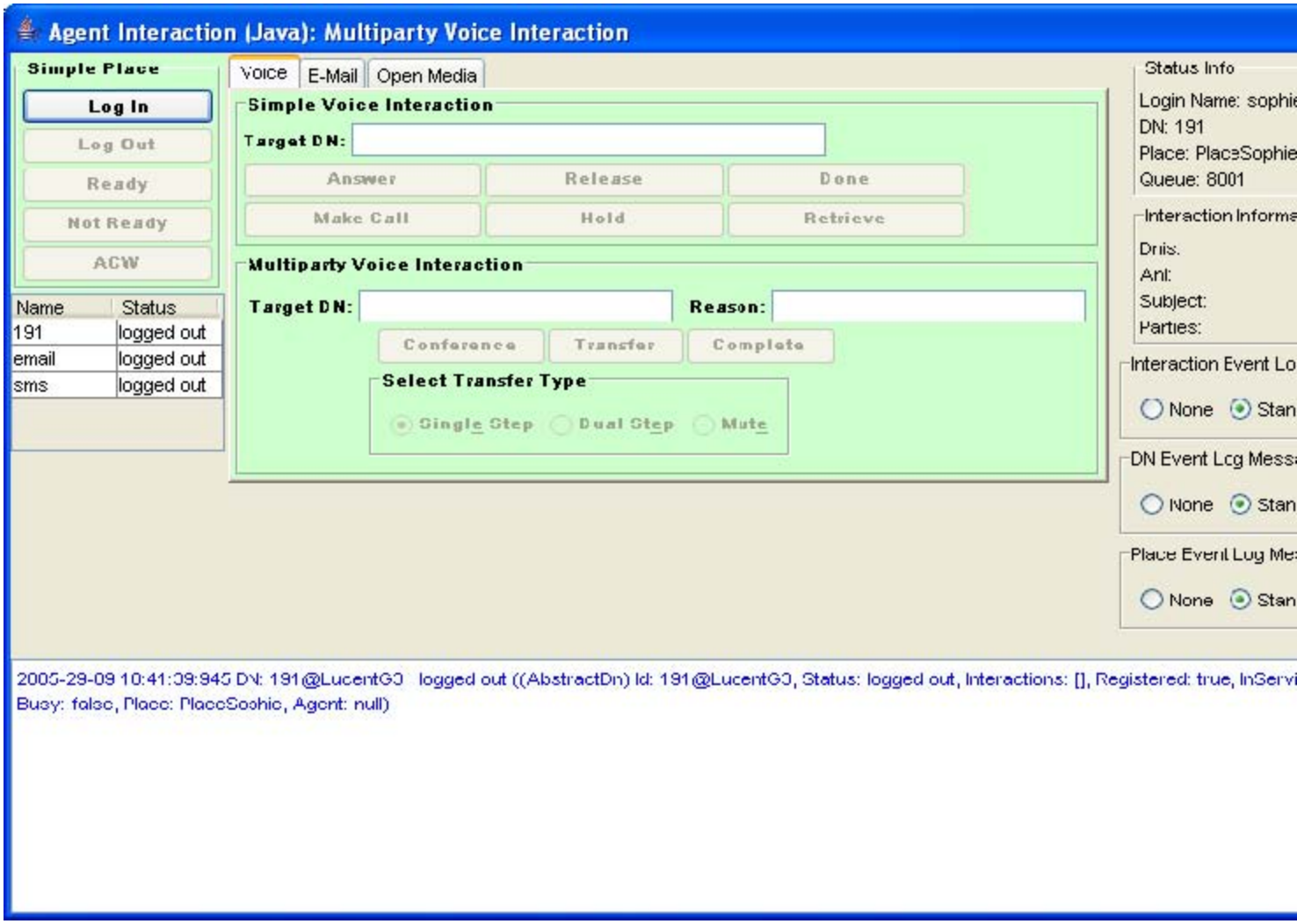
4. Finally, the GUI must be updated to keep in sync with the state of the application:

```
 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.

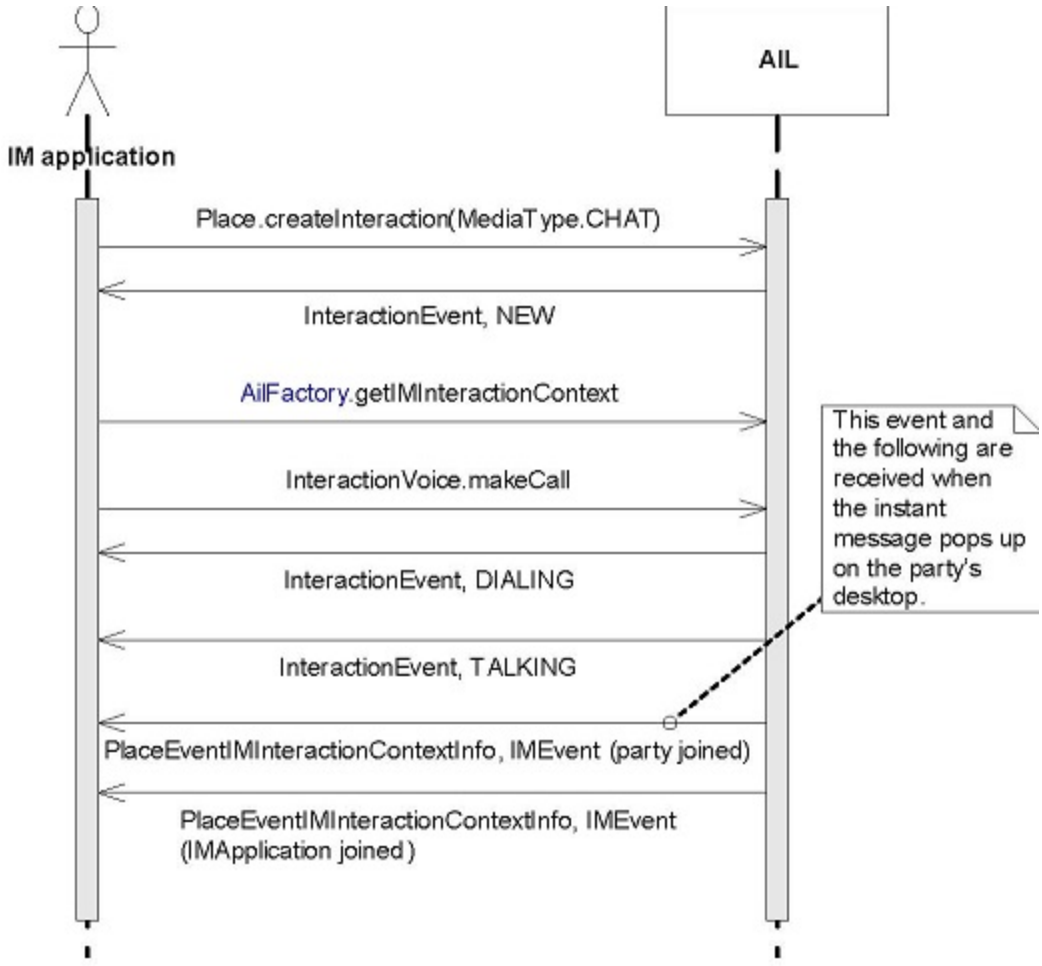
```
Iterator it = sampleContext.getTranscript().iterator();
while (it.hasNext())
{
        IMEvent ev = (IMEvent) it.next();
        // Process the event
        if( ev instanceof IMMessage)
        {
                IMMessage msg = (IMMessage) ev;
                IMParty party = msg.getParty();
                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.



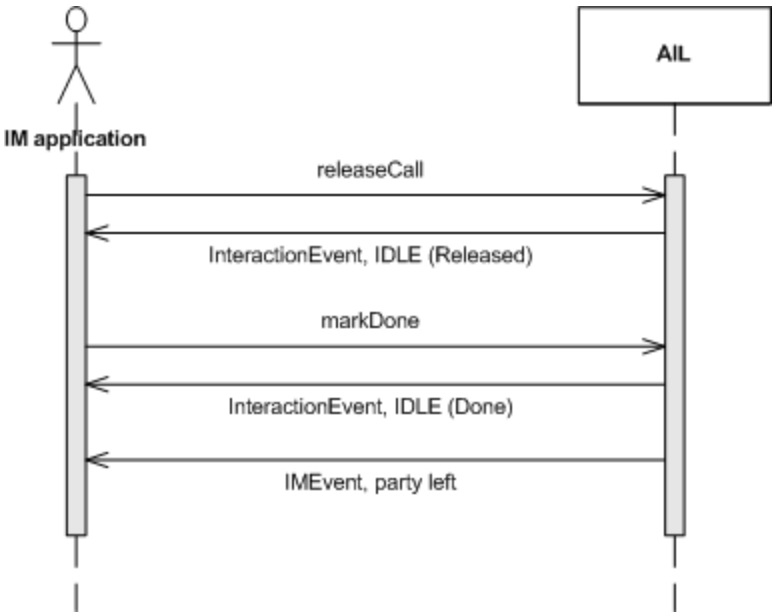
New Instant Messaging Session

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).

Switch Facilities

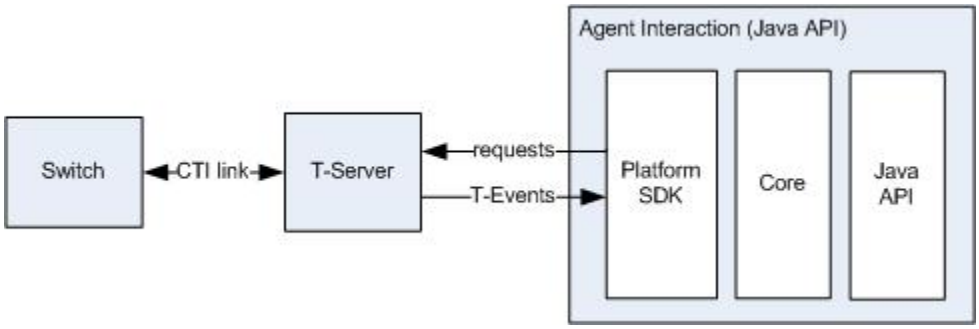
This chapter describes the switch facilities provided by the Agent Interaction (Java API).

Switch Design

The Agent Interaction (Java API) provides easy access to available switch features and data, and facilitates development when different switches are involved.
To ensure both data persistence and consistency, the core library manages connections with Genesys components and deals with incoming information by maintaining corresponding objects.
So forth, the Agent Interaction (Java API) uses a voice state model to manage components related to voice features (such as outbound, callback, and so on), and CTI objects corresponding to the use of particular switches across their T-Servers.
As far as they are able, state machines guarantee that the voice state model is coherent with other Genesys components (for example, multimedia solutions) across supported switches. This facilitates the integration of these services and applications. In contrast, developers using the Platform SDK must build their own state machines, which increases the integration complexity.

T-Server Connections

The AIL library core does not directly connect to any switches, but rather to the T-Servers that manage the switches. The AIL core library integrates with the Platform SDK to communicate with the T-Server that drives the switch. Therefore, the AIL's CTI features are limited to T-Server CTI features: The AIL library provides you only with what the T-Servers can perform.



AIL Driving the T-Server via the Platform SDK

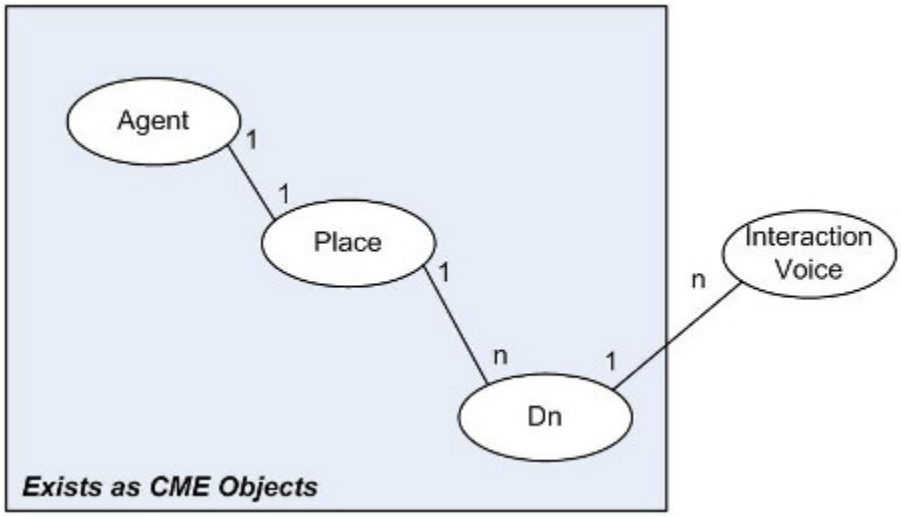
Through the Platform SDK, the AIL core library sends requests to the T-Server in order to perform agent actions (See Calls Mapping.) The T-Server manages its connection with the switch and generates requests to drive the switch
Then, the T-Server notifies the AIL core library with TEvents. The AIL core library updates its model, then notifies listeners with an event built from information contained in the TEvents.

> **Warning**
> The AIL library makes no assumption about the success or failure of any request processed through the API to the T-Server and only updates with TEvents.

Voice Model

The voice model is consistent with the models defined for other media. Prior to any voice manipulation, your application must log in an agent on a voice media (a DN) which is available in a `Place`. Through this logged DN, you get interactions that enable voice actions. Managing the interaction on the switch involves manipulating the `Dn` and `InteractionVoice` interfaces.

The voice model that relies on the main AIL classes—`Agent`, `Place`, and `Dn`—is presented in The Voice Model.



The Voice Model

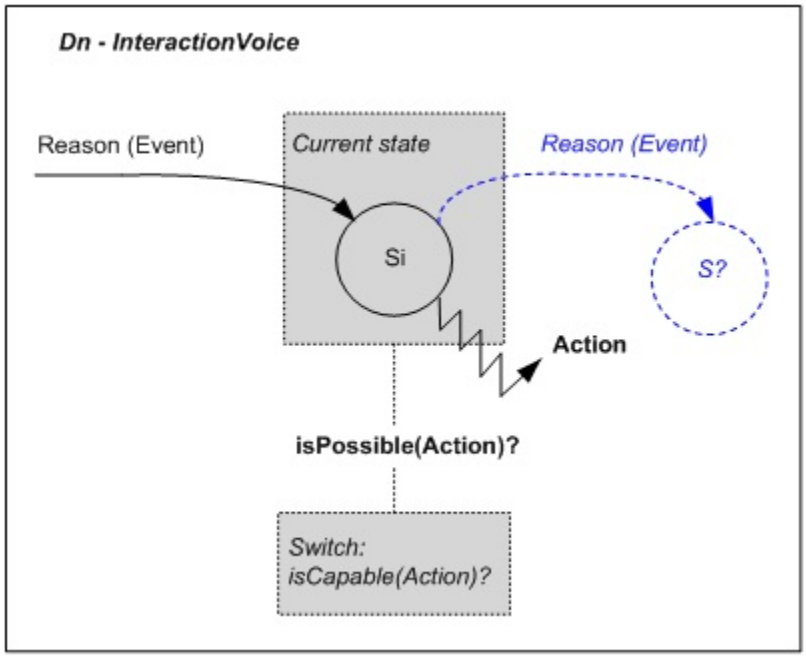
Voice State Model

The `Switch` interface accesses general information, that do not change at runtime, in contrast to the `Dn` and `InteractionVoice` instances that agent actions affect. For these two last classes, the AIL library `provides you with a voice state model, defined as follo` ws:

- `States`: These are the results of actions. Your application is notified of these actions as the switch performs them. The current state affects the range of actions that your application can perform.
- Transitions: These depend on the success of the last action. They are performed according to the last notified TEvent and are labeled according to the TEvent name.

States

Internally, the AIL core library manages both `Dn` and `InteractionVoice` objects through their own state machines. These and other objects inherit from the `Possible` interface, which offers the `isPossible()` method to test whether an action can be performed from the current state.

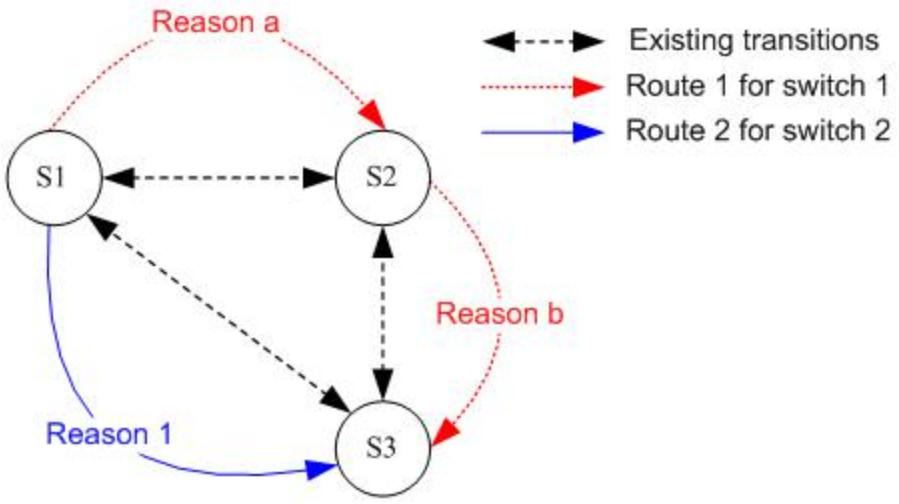


Possibility of Performing an Action on the Switch

This figure shows the `isPossible()` method dependencies. This method takes into account the current state of the object, and whether the switch is capable of a particular action, to determine the action's possibility with respect to current state. For details, see Determine Availability of CTI Features.

Transitions

Because the TEvent is an *acknowledge* action coming from the T-Server, the AIL core library first updates its model, then it builds the corresponding `DnEvent` or `InteractionEvent` events that notify your client application. These events' reason correspond to TEvent names.
The T-Server call flow differs from one switch to another: it is affected by switches, switch settings, and T-Server settings. Therefore, the library cannot impose uniformity on the transition sequences. The routes in state models are switch-dependent, as illustrated in Reasons and Routes.



Reasons and Routes

The figure shows that from one switch to another, the TEvent sequence is different and is associated with different reasons and routes in the state models. To cite a concrete example, an application monitoring a `Routing Point` has a state sequence that is different on Avaya G3 versus Nortel Meridian switches.

> ### Important
> You should design your applications with respect to object status rather than event reasons. The state model ensures coherence and reliability, whereas event reasons are strongly switch-specific.

> ### Warning
> Do not make any assumptions about incoming events. 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).

Switch and DN Management

Now that you have been introduced to the switch implementation in Agent Interaction (Java API), it is time to outline the consideration you will need to work with switches.
First, in order to log in properly on DNs, you must learn about the DN Consolidation, that explains the

DN consolidation through the Dn interface. It presents how you identify and access consolidated DNs defined for a given switch.
Then, before you make calls to voice features, you check whether these calls are available, as explained in Determine Availability of CTI Features.
After you checked the feature is available, you can perform a method call. This call is mapped with the Platform SDK, as detailed in Calls Mapping.
As the Agent Interaction Java API makes no assumption about action result, to get acknowledge of successful actions, you listen to events, as explained in Event Flow.

DN Consolidation

Regardless of the DN types that the switch handles, the DN consolidation model provides a single Dn interface with a unique DN ID, that a voice interaction reaches through its callable number, as detailed in the following subsections.

DN Types

Switch DNs' types—for example, ACD position or extension—are mostly switch-specific. This affects the use of such features as login, logout, ready actions, and so on.

The AIL library provides consolidation of the Dn object and its model so that you do not have to write the code to manage this in your application. In all cases, the library exposes a single Dn, even if the configuration of a place requires more DNs in the Configuration Layer according to the underlying switch.

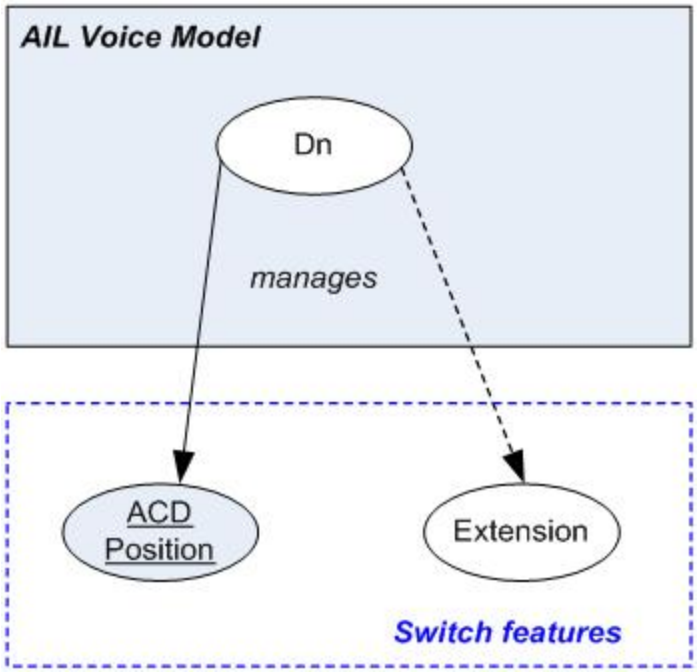
To find the required configuration for each switch, and thus find which DN type is visible, refer to the *Interaction SDK 7.6 Java Deployment Guide.*

> ### Warning
> You must respect this DN consolidation model if your applications are to handle DNs properly.

For example, to work in regular mode with an A4400 switch, the AIL:

- Registers the ACDPositio n and Extension.
- Exposes a single Dn object.

The Agent Interaction (Java API) presents a single DN to transparently manages the requests to hidden ACDPositions and Extensions. For A4400, the DN ID exposed in the API is the ACDPosition, as shown below.



A4400 - DN Consolidation

As another example, to work in substitute mode on an A4400 switch, your application must manage activities of an `extension` on a `Place` . When no agent is logged in, the extension is visible and the ACD position DN is not visible. When an agent is logged in, the ACD position DN is visible in the place and the extension is not visible in the place. For details about the corresponding events, refer to DN Events.

DN ID

The DN ID is used to retrieve the corresponding `Dn` instance through the `AilFactory.getDn()` method.
In this product's 6.5 releases, the DN ID used to be the Configuration Layer's DN callable number. This restriction did not allow two distinct switches to declare two identical DN numbers. To avoid any ambiguity and to ensure unique DN IDs, a DN ID is now defined in accordance with the following rule:`<DN_CME_Name>@<switch_name>`

> **Important**
>
> Set the `enabled` option to `false` in the `dn-at-switch` section to disable this rule and retrieve DN IDs as defined in 6.5 releases.

Callable Number

The callable number is the number that your application must use to reach a DN when, for instance,

making a call. Depending on the switch, the DN number declared in the Configuration Layer might not be the number to dial for call actions, such as make call, transfer, or conference. For example, you must remove some leading numbers to reach a Nortel DMS 100's DN.
The number to dial can also depend on the DN status. For example, if an agent is logged into the DN, you have to dial the agent ID.
Finally, because of the consolidation model, some DNs can be hidden but still be the real ones to dial instead of their visible counterpart. Once again, the callable number of a visible DN properly returns the hidden DN's number.
To determine the callable number of a `Dn` instance at runtime, call the `Dn.getCallableNumber()` method.

> **Important**
> Genesys recommends using callable numbers to access CTI features.

DN Activation

To perform telephony actions and receive telephony events for a DN on a T-Server, the library uses an internal `Dn` object, represented in the Agent Interaction (Java API) by the `Dn` interface.

Typically, when an agent uses your application to log in, the library creates a new `Dn` object for the agent's DN, but also creates a new monitoring session for the new `Dn` object. The library uses the monitoring session to register the `Dn` on its T-Server and then to monitor activity through the life of the `Dn` object. Usually, the DN's monitoring session is released at garbage collection time.

Typically, when developing a server application, it is possible that after one agent logs out, another agent may log in to work with the same DN. Your application can keep a `Dn` object alive in the time period between one agent's logout and another agent's login, but in that case, monitoring could cause unwanted interactions.

To kill a monitoring session for a DN, use the `Dn.unactivate()` method, which unregisters the DN on the T-Server and kills the monitoring session.

Later, at the time another agent is about to log in to start work with the same DN, the internal `Dn` object is still alive but inactive because it is not registered to its T-Server and it has no monitoring session. Before the agent can log in, your application must create a new session that re-registers the DN on the T-Server and lets the library perform actions and receive events. To create a new monitoring session, call the `Dn.activate()` method. This registers the DN on the T-Server, allowing the new login, and instantiates a new monitoring session for that `Dn` object.

> **Important**
> By default, registering and unregistering actions are handled by the library. If your client application calls the `activate()` or `unactivate()` method, your application is responsible for managing the monitoring session of the relevant DN.

The `activate()` and `unactivate()` methods are documented in the *Agent Interaction 7.6 Java API Reference* as members of the `AbstractDn` interface, which is a Superinterface for the `Dn` interface.

Determine Availability of CTI Features

After your agent has successfully logged in on a DN, your application will access CTI features only if these features are available on the DN's switch. Examples of such possibly unavailable features on a switch are `Transfer`, `Conference`, `Callback`, and `Holding`. Refer to your T-Server documentation to see which features are available on a given switch.
The Agent Interaction (Java API) is agent-oriented. This means that for a logged in agent at runtime, you are provided with all possible CTI features, and even methods to determine which features are available. These methods indicate switch *capabilities* and *possibilities,* as detailed in the following subsections.

Capability

You access capability through the `Switch interface`, which has the following characteristics:

- An instance of the `Switch` class is a static object.
- Access to a `Switch` object is given by the `Dn.getSwitch()` method, or `AilFactory getSwitch(java.lang.String name).`

The `Switch.isCapable()` method takes into account the switch's features and returns the switch's capability to perform the `InteractionVoice.Action` during the entire runtime session. The result is independent from the voice DN's current state.
Capability testing enables your application to determine what features are available on the switch and behave appropriately—for example, to show or not show buttons, menus, and so on.
For example, the `MultipartyVoiceInteraction` example tests transfer capabilities to determine whether it should display or not buttons for the transfer and conference features, as show in the following code snippet:

```
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);
```

Possibility

According to the current state of the objects (`Dn or Interaction`), a feature may be temporarily unavailable, although the switch is capable of performing the corresponding action. For instance, in most cases, if your agent is not logged in on a DN of the switch, your agent will not be able to create a new voice interaction with the corresponding `Dn` interface.
After the agent will be logged in, the Agent Interaction Java API will get an event that will change its

DN status, so that creating a voice interaction will become possible.
Possibility, managed through the `Possible` superinterface with the `isPossible()` method, is applied to actions that you wish your application to perform. To determine an action possibility, the `isPossible()` method concatenates:

- The object's current state, as calculated by the AIL.
- The switch capability to perform the action.

This is illustrated in Possibility of Performing an Action on the Switch.
Genesys recommends using this `isPossible()` method to determine feature accessibility each time an event occurs.

> **Warning**
>
> The `isPossible()` method does not reflect the availability of features that depend on DN type, T-Server options, or the switch environment (ACD, Genesys Routing, proprietary call distribution, and so on.)

A GUI application should use the `isPossible()` method to enable or disable buttons according to events. The following code snippet is extracted from this method and manages three radio buttons, enabling the agent to select the transfer to perform:

```
//The complete button should be enabled if a transfer or a conference
//can be completed
boolean complete = sampleInteraction.isPossible(InteractionVoice.Action.COMPLETE_TRANSFER)
   || sampleInteraction.isPossible(InteractionVoice.Action.COMPLETE_CONFERENCE);
completeButton.setEnabled(complete);
```

> **Important**
>
> Remember that the current state changes with events. Your event handlers should continually validate the availability of features.

Calls Mapping

The AIL core library handles switch specifics internally to allow easy, consistent management of voice features across all switches that AIL supports. Refer to the *Interaction SDK 7.6 Java Deployment Guide* for the list of supported switches.
When you make calls to voice methods, the AIL core library transparently takes over the required calls to Platform SDK functions, regardless of the switch type.

For Platform SDK users, the mapping of the Platform SDK features is straightforward because the naming convention is similar to that for Platform SDK functions. By contrast, building an application using the Platform SDK would require that you write code to deal with switch specifics in your function calls.

One example is the DMS100 switch and its specific implementation of a `makeCall()` method. Using

the Platform SDK, your application must first create a RequestMakeCall object and then the RequestAnswerCall object, as illustrated in the following code snippet. If it does not do both, the agent must manually go off hook to begin the call.

```
//requesting a MakeCall to the TServer using the Platform SDK
server.send( RequestMakeCall.create(
                _dn,
                _otherdn,
                _make_call_type,
                _location,
                _userdata,
                _reasons,
                _textensions);
//....
//This is a DMS100 switch: requesting an AnswerCall
server.send( RequestAnswerCall.create(
                _dn,
                NULL_CONNECTION_ID,
                _reasons,
                _textensions);
//...
```

The benefit of hiding these switch-specific interventions is that your code is less complex and your feature management is easier, available, and works for all switches supported by AIL. The AIL code corresponding to the previous Platform SDK code snippet is:

```
// Dial the call
voice.makeCall( myDn.getCallableNumber(), //callable number to dial
  _location, //location
  InteractionVoice.MakeCallType.REGULAR, //Type
  _userdata, //User data
  _reasons, //Reasons
  _textensions);//Textensions
//...
```

Although the Agent Interaction (Java API) simplifies switch management, you can still tune method calls with switch-specific parameters. Refer to Switch Tuning for further details.

Event Flow

The AIL core library makes no assumption of the method call's result. As explained in Voice State Model, after the request succeeds, the AIL core library will get a TEvent that will generate a DnEvent or InteractionEvent event (according to the called method.)
In case the method call fails, you should get an exception. Refer to the *Agent Interaction 7.6 API Reference* for further details about method exceptions.

> ### Important
>
> Because a Place reflects **all** Dn and Interaction events, Genesys recommends that you use a PlaceListener rather than using separate listeners for each Dn orInteraction.

DN Events

Through the `Place.handleDnEvent()` method, your application gets `DnEvent` events, that is information about successful login, logout, ready, not ready actions performed on the consolidated DNs of this place.
Through these events, you will get accurate information about DN status, in particular for some DNs that become invisible due to the consolidation (as explained in DN Consolidation).
As an example, to work in substitute mode on an A4400 switch, your application must manage activities of an `extension` on a `Place`. When an agent logs in successfully, the `Place.handleDnevent()` method gets two events:

- `dnRemoved` provides notification that the `extension` is no longer visible.
- `dnAdded` provides notification that a `Dn` of type `ACD position` is now visible.

Successful agent login generates a login event; the login event and all subsequent event and request activities occur with respect to the ACD position DN, not the extension.
When the agent logs out, logout is performed through the ACD position DN, and upon successful logout, the `Place.handleDnevent()` method gets four events:

- An event carrying notification of successful logout.
- A `dnRemoved` event carrying notification that the ACD position DN is no longer visible.
- A `dnAdded` event with notification that the extension is now visible.
- An event carrying notification of the extension status.

When no agent is logged in, the extension is visible and the ACD position DN is not visible. When an agent is logged in, the ACD position DN is visible in the place, and the extension is not visible in the place.

Single-Step Rollover to Mute Transfer

The Agent Interaction (Java API) provides you with a hidden mute transfer in the `singleStepTransfer()` method:

- If the switch is not capable of performing the single-step transfer, AIL transparently performs a mute transfer instead.
- The `isCapable(InteractionVoice.Action.SINGLE_STEP_TRANSFER)` call takes into account the hidden mute capability.
- The `isPossible(InteractionVoice.Action.SINGLE_STEP_TRANSFER)` call takes into account the hidden mute possibility.

Even `if true single-step transfer` is not available, the capability and possibility results indicate whether your application can perform the mute transfer instead. So, you can rely on this result to perform your transfer with the `singleStepTransfer()` method.

Switch Tuning

To fine-tune your application according to your switch, the Agent Interaction (Java API) provides you with TExtensions and workmodes.

> **Warning**
> Managing some switch-specific data implies a dependency on the corresponding particular switch.

TExtensions

`TExtensions` are `Map` data structures that take into account switch-specific features and information that cannot be described in a request parameter. The library transmits them as parameters in Platform SDK calls used to tune T-Server operations.
To get details about the list of extensions associated with a switch, refer to the corresponding T-Server documentation.
`TExtensions` exist for both `Dn` and `InteractionVoice` interfaces. You get them by calling the `getTExtensions()` methods. Refer to the *Genesys Platform SDK 7.6 Developer's Guide* for further information.

In Method Calls

To add TExtensions in a method call, you just need to create a `Map` of key-value pairs and pass it as parameter in your voice method call.
For example, if you are working on an application dedicated to the G3 switch, you may want to specify the `Trunk` TExtension for the `makeCall()` method, as shown in the following code snippet.

```
String myTrunk = "...";
HashMap myTExtension = new HashMap();
myTExtension.put("Trunk", myTrunk);
voice.makeCall( DN,
 null,
 InteractionVoice.MakeCallType.REGULAR,
 null,
 null,
 myTExtension);
```

> **Important**
> TExtensions that apply only to a given switch are described in the corresponding individual T-Server manual.

In Events

Both DnEvent and InteractionEvent events propagate TExtensions, that are switch-specific TEvent Extensions. These TExtensions are copied from the TEvent and are additional data provided for switch-specific interventions.
You can retrieve these TExtensions in a Map returned by calling InteractionEvent.getTEventExtensions() or DnEvent.getTEventExtensions(). As TExtensions are switch-specific, refer to your T-Server documentation to learn more about its TExtensions.
The following code snippet shows how to deal with TExtension Maps.

```
// Implementation of the Agent.HandleInteractionEvent() method
public void handleInteractionEvent(InteractionEvent _ie) {
        //...
        //Retrieval of the map containing the TExtensions

        Map myTExtensions = ie.getTEventExtensions();
        //Testing if there is TExtension attached in the event
        if (mTExtensions != null) {
                // Retrieving an iterator for the key set
                Iterator it = mTExtensions.keySet().iterator();
                while (it.hasNext() ) {
                        String key = (String) it.next();
                        System.out.println("Key: "+key+" Value: "+mTEventExtensions.get(key));
                }
        }
        //...
}
```

> **Warning**
>
> TExtensions can be optional in a TEvent, so the library propagates them only if they exist. Refer to your T-Server documentation for more information.

Workmodes

Like the Platform SDK, the Agent Interaction (Java API) provides you with dedicated methods to handle workmodes. Workmodes are used in agent events to provide more detailed information about an agent's actual state.
For example, the Manual in workmode for a log action indicates that the agent must validate the action manually on the phone. The After Call Work workmode indicates the agent is still working on the last call.
Workmodes are identified by the Dn.Workmode class. Refer to the Javadoc API Reference to get the list of workmodes taken into account by the API.
Workmodes can be specified in the parameters of dedicated methods related to Dn.Action. You can use them in the Agent, Place, and Dn interfaces for calls to their respective login(), logout(), ready(), and notready() methods.

> **Important**
>
> If you use a workmode in a method call performed from an `Agent` or `Place` object, all the associated voice DNs can be affected by the workmode that you specify.

Workmodes

To test available workmodes, the Agent Interaction (Java API) provides you with specific methods in the `Switch` and `Dn` interfaces:

- `Switch.isWorkmodeCapable(Dn.Workmode)` —`true` if the switch supports the DN workmode.
- `Dn.isWorkmodePossible(Dn.Workmode)` —`true` if the DN workmode is available for the next action on the `Dn` .
- `Dn.getPossiblebleWorkmodes()`: returns, in a table of `boolean` values, the availability of the different workmodes for the next action on the `Dn`.

You can use these possibility and capability tests in the same manner as the standard tests, which are described in Capability and Possibility.

> **Important**
>
> The provided workmode tests reflect only the T-Server's workmode availability. AIL does not take into account any action to perform.

The following code snippet shows how to use the `Dn.isWorkmodePossible()` method:

```
//...
if(mDn.isWorkmodePossible(Dn.Workmode.NO_CALL_DISCONNECT)==true) {
   mDn.ready(mQueue, Dn.Workmode.NO_CALL_DISCONNECT, null, null);
}
//...
```

> **Warning**
>
> This capability test is not processed by the T-Server and does not take into account any T-server or switch settings. Refer to your T-Server documentation to enable your workmodes.

After Call Work

The API provides additional `afterCallwork()` methods in the `Agent`, `Place`, and `Dn` classes specific to the After Call Work mode. You can call these methods without specifying any workmode, as shown in

the following code snippet:

```
myAgent.afterCallwork(null, null, null);
```

The `afterCallwork()` methods are equivalent to a `notReady()` call with the workmode `AFTER_CALL_WORK` passed as a parameter, as shown in this code snippet:

```
myAgent.notready(null, Dn.Workmode.AFTER_CALL_WORK,null,null);
```

The `afterCallwork()` methods enable you to put the concerned DNs into the associated status: `Dn.status.AFTERCALLWORK`. This status is an extension of the `NotReady` state: it means that the agent is not ready to receive another call, because he or she is working on another task.

> **Warning**
>
> If the `After Call Work` workmode is not supported by the underlying switch, `afterCallwork()` methods are unavailable.

PSDK Bridges

The Voice and Configuration PSDK Bridges are two distinct APIs that are integrated into the to the Agent Interaction Java API. They are intended to be used for implementing additional controls on objects that the Agent Interaction Java API does not handle.
Using this API requires an advanced level of coding experience with Agent Interaction Java API. Any custom code on top of these bridges requires requires testing in a production-like environment prior to any deployment.

Guidelines

As mentioned above, the PSDK bridges are particular APIs integrated into the Agent Interaction Java API. They provide access points to PSDK protocol instances in order to implement additional controls through the Platform SDK API:

- `com.genesyslab.platform.voice.protocol.TServerProtocol`
- `com.genesyslab.platform.configuration.protocol.ConfServerProtocol`

As explained in the following sections, the use of these classes is restricted to the Call Center objects that do not interfere with the AIL Core. For more details about these protocol classes, refer to the *Platform SDK 7.6* documentation.

Protocol Instances

When your application calls the `PsdkConfigBridge.getConfigServerProtocol()` or the`PsdkVoiceBridge.getTServer()` method, it retrieves the PSDK protocol instance that the AIL Core uses to communicate with the Configuration Server or with the specified switch.
Even when using the bridge, AIL manages connections, disconnections, HA and ADDP. (In any case, as a general guideline, your application should not deal with any of them through the AIL's PSDK bridges.)

> **Warning**
> Do not call the `close()` methods of the protocol instances that you get from the PSDK bridges. You would close the AIL's connections and make unavailable the server and the associated features until AIL reconnects.

Use special care when working with the Platform SDK since it exposes low-level APIs for the underlying servers. Because you deal directly with the PSDK protocol instances and not with some interfaces as for other Call Center objects (Agent, Place, DN, and so on), the AIL Core is not aware of requests that you make through PSDK bridges.

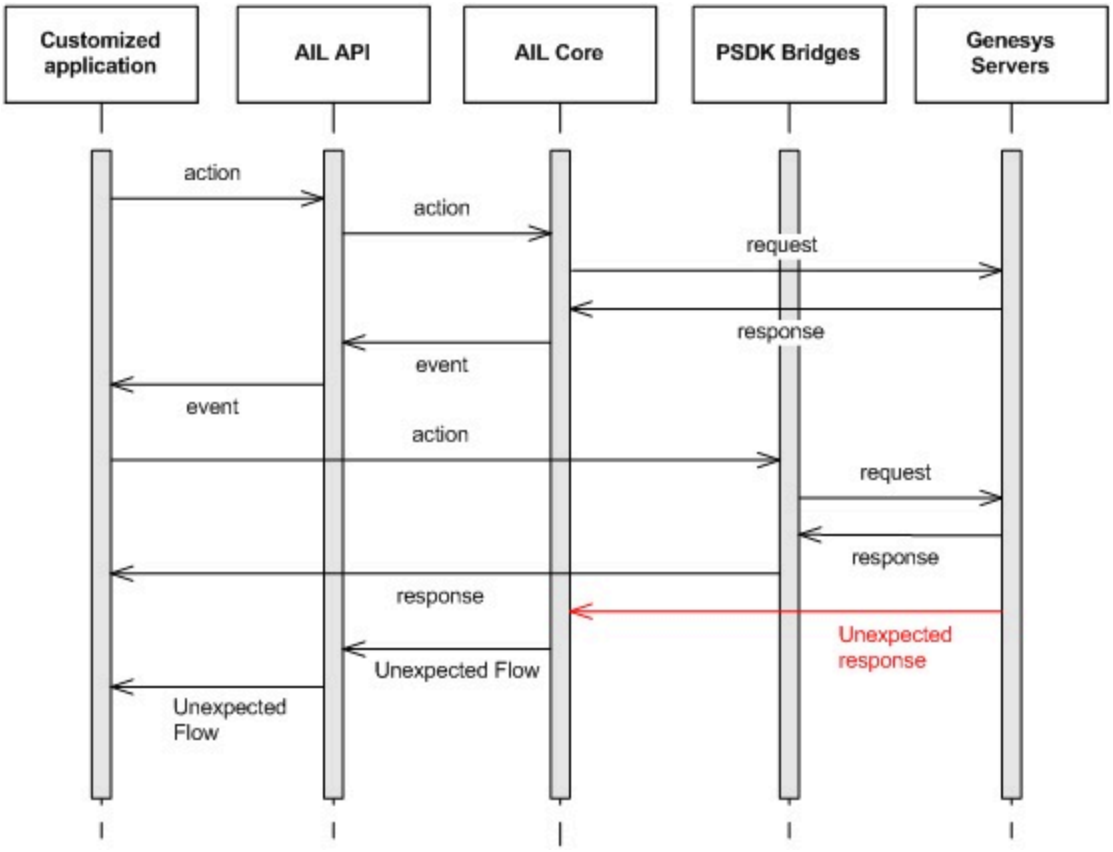
The PSDK Config Bridge' usage is restricted to monitoring objects or retrieving information. The PSDK Voice Bridge, in addition to monitoring features, supports some voice requests. The customization of

your application with the PSDK bridges can bring some overhead with the event flow or put AIL out of synchronization, as explained in the following sections.

> **Important**
>
> Prior to the deployment of this customization, you must perform unit tests on your application, and also load tests in production-like environments.

Message Flow

When you deal with the PSDK bridges, you create a new message flow using the `com.genesyslab.platform.voice.protocol.TServerProtocol` and `com.genesyslab.platform.configuration.protocol.ConfServerProtocol` classes.
The PSDK bridges' protocol instances enable you to send requests and, for those requests, if you get responses, AIL receives those too. As a result, AIL generates an event flow according to the responses received, as it does when no PSDK bridge is implemented. The difference is that you receive responses low-level requests that you might not otherwise. Usually AIL only handles messages that are responses to its own requests.
For this reason, AIL may generate unexpected events from an agent application's point of view. It is not possible to assume what the PSDK's additional message flow is, because it strictly depends on how your application makes use of the PSDK bridges.



Message Flow

PSDK Config Bridge

- **Description:** The PSDK Config Bridge is an access point to the Configuration Platform SDK which enables you to monitor objects in the Configuration Layer of your Genesys environment. Use special care when working with the Platform SDK since it exposes low-level APIs for the underlying servers.

For instance, you should avoid sending modification requests to the Configuration Server because the AIL Core would not be notified of these changes, and the AIL cache would be out of synchronization.

PSDK Voice Bridge

The PSDK Voice Bridge is an access point to the Voice Platform SDK which enables you to monitor voice interactions from a traditional or IP-based telephony device. In addition, you can also perform requests with the following packages:

- `com.genesyslab.platform.voice.protocol.tserver.requests.dtmf`
- `com.genesyslab.platform.voice.protocol.tserver.requests.special`
- `com.genesyslab.platform.voice.protocol.tserver.requests.queries`

- `com.genesyslab.platform.voice.protocol.tserver.requests.iscc`
- `com.genesyslab.platform.voice.protocol.tserver.requests.voicemail`

Use special care when working with the Platform SDK since it exposes low-level APIs for the underlying servers. Although AIL handles unsolicited telephony events, modifying devices or interactions may lead to data inconsistency, for instance, in connection with the result of the`isPossible()` method calls or in reference IDs that AIL creates for `Interaction` instances.

> **Warning**
> Do not modify DN registration with the PSDK Voice Bridge.

Managing PSDK Listeners

Even when using the bridge, AIL manages connections, disconnections, HA and ADDP. (In any case, as a general guideline, your application should not deal with any of them through the AIL's PSDK bridges.)

> **Warning**
> Do not call the `close()` methods of the protocol instances available through the PSDK bridges. That would close the AIL connection and make unavailable the server and the associated features till AIL reconnects.

In case of disconnections, the listeners (that your application registered to get protocol messages) are removed. So, your application must listen to configuration and telephony service statuses to get notified of disconnections. When AIL reconnects, the customer application must register its listeners again to get PSDK protocol messages. See also the section Steps for Integrating PSDK Config Bridge.

Steps for Integrating PSDK Config Bridge

This section offers a short implementation example to handle properly the notification of protocol messages for the PSDK Config Bridge.

1. Create a new class implementing the ServiceListener interface.

```
public class ConfigBridgeAdapter implements ServiceListener {
 AilFactory myFactory;
```

In the constructor, add your listener to the listener list of the `CONFIG` service which monitors the connection to the Configuration Server, as shown below.

```
 public ConfigBridgeAdapter(AilFactory _myFactory) {
    myFactory = _myFactory ;
    myFactory.addServiceListener(ServiceStatus.Type.CONFIG,this);
```

```
}
```

2. Then, implement the handler for the status changed events. When the connection to the Configuration Server is in ON status, you must register a PsdkConfigBridge.Listener instance to get protocol messages. To correctly process events, you should use two class variables, including a QueuedExecutor instance:

```
// Class variables
QueuedExecutor mQueue = new QueuedExecutor();
PsdkConfigBridge.Listener mListener;

public void serviceStatusChanged( ServiceStatus.Type service_type, java.lang.String
service_name, ServiceStatus.Status service_status) {

        // handle Service.Status on disconnections
        if(service_status.toInt() == ServiceStatus.Status.OFF_) {
                //...
                // TODO: implement what your application should do // when connection is
turned OFF
                //...
        } else if(service_status.toInt() == ServiceStatus.Status.ON_)
        {
                registerBridge();
        }
}

public void registerBridge()
{
        // Create the listener
        mListener = new PsdkConfigBridge.Listener() {
                public void handle( Message message ) {
                        mQueue.execute( new Runnable() {
                                public void run() {
                                        myHandler( message );
                                }} );
                }};

                PsdkConfigBridge.addConfigServerListener( mListener );
                // Your application now receives protocol messages
}

public void myHandler( Message message ) {
        /**
         * TODO Implement the message handling
         **/
}
} // End of ConfigBridgeAdapter class
```

As soon as you add your ConfigBridgeAdapter instance to the listener list of the CONFIG service, your class is notified of the connection status and is added to the protocol's listener list.

3. Now, you can retrieve the protocol instance and start customizing your application.

```
// Get the factory
AilFactory my_factory = AilFactory.getAilFactory();
// Listen to CONFIG service
ConfigBridgeAdapter myAdapter = new ConfigBridgeAdapter(my_factory);

// Get Protocol instance
com.genesyslab.platform.configuration.protocol.ConfServerProtocol
ailConfigProtocol = PsdkConfigBridge.getConfigServerProtocol() ;
```

```
    /**
    * TODO: implement customization
    */
```

E-Mail Interactions

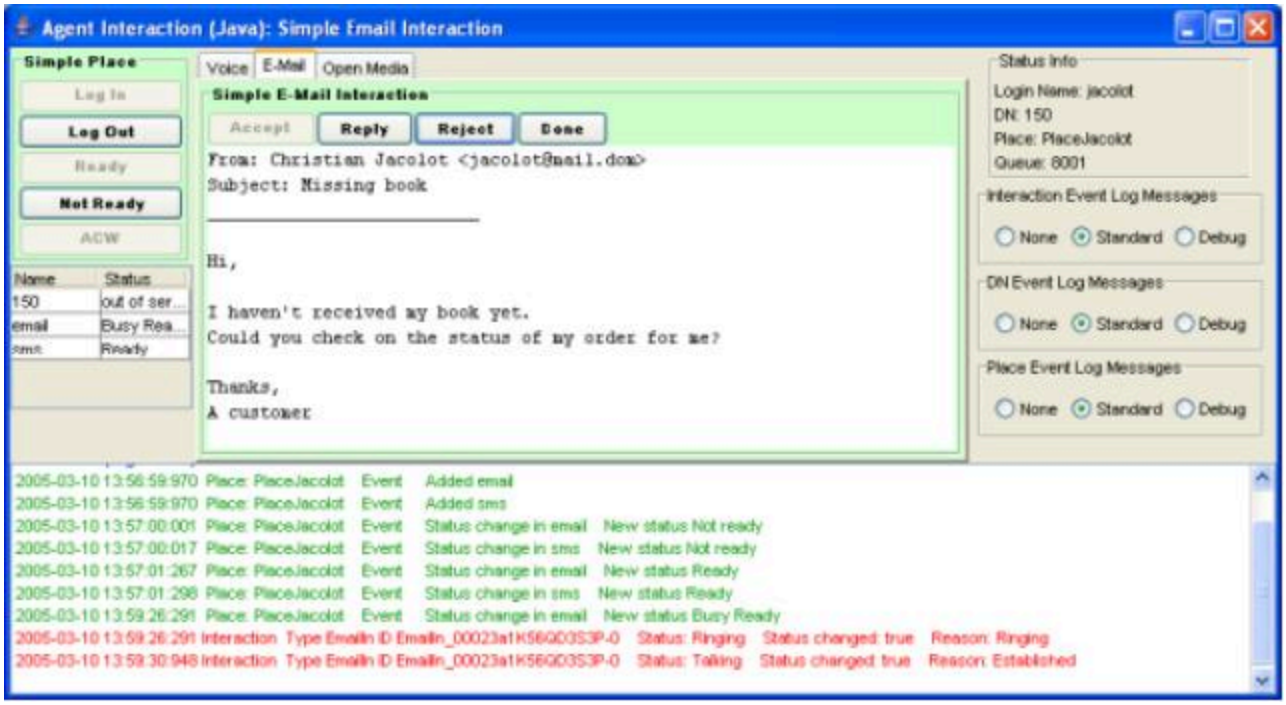
Multimedia interactions are interaction interfaces inheriting the `InteractionMultimedia` interface, that is, common e-mails, collaborative e-mails, chat interactions, and open media interactions. This chapter presents `SimpleEmailInteraction`, a code example that lets a user receive and answer e-mails. It also covers collaborative e-mails and workbin interactions.

SimpleEmailInteraction

This example is similar to `SimpleVoiceInteraction`, which was introduced earlier. It uses the same graphical user interface and the same internal structure, inheriting from `SimplePlace`.

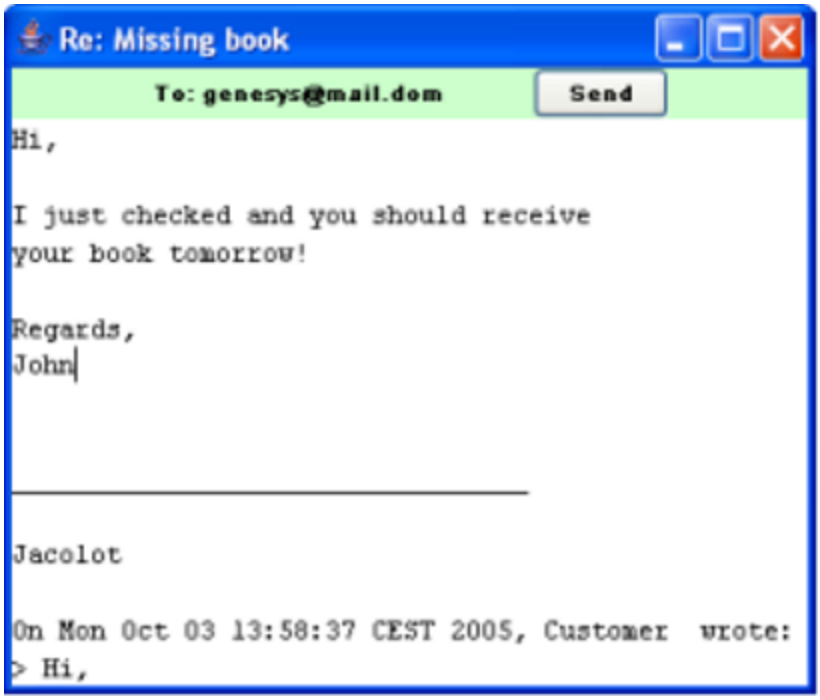
> **Important**
>
> For the sake of simplicity, this example is designed to handle one e-mail at a time. Set up a capacity rule limiting the agent to a single e-mail at a time in your routing strategy. For further details, see *Universal Routing 7.6 Documentation*.

When an e-mail arrives for this agent, the `Accept` button will be enabled. At this point, the agent can accept the e-mail. This displays the text of the incoming e-mail, and also enables the `Reply` button, as shown in Accepting an E-Mail.



Accepting an E-Mail

If the agent clicks the `Reply` button, a new window is created for the agent to type and send a response, as shown in The E-Mail Reply Window.



The E-Mail Reply Window

After the agent sends the e-mail, the processing of the inbound e-mail is finished and the example releases all the associated interactions.
As in the `SimplePlace` and `SimpleVoiceInteraction` examples, there are only six steps to follow in writing this application. As some of these steps have been done by `SimplePlace`, which is the superclass for this example, they will not be repeated here.

Set Up Button Actions

After calling the superclass method and setting up the tab, the `linkWidgetsToGui()` method is ready to set up the buttons. The `Accept` button is very simple. All it does is "answer the call" when an e-mail comes in:

```
sampleEmailIn.answerCall(null);
```

This changes the status of the incoming e-mail from `RINGING` to `TALKING`.
The `Reply` button issues a call to the `createReply()` method of `SimpleEmailInteraction`. This method first creates an outbound reply e-mail:

```
// Getting a queue to which to send the reply email.
String outboundQueue = getOutboundQueues();
sampleEmailReply = sampleEmailIn.reply(outboundQueue, false);
```

Then it makes call to an `agentInteractionGui` method, as shown here:

```
// Create a dialog box to enable the user to enter the reply text.
// The subject, the addresses, and part of the message are
// already created in the interaction.
agentInteractionGui.createReplyWindow( sampleEmailReply.getSubject(),
sampleEmailReply.getFromAddress().toString(),
sampleEmailReply.getMessageText());
```

This method creates a new window allowing the agent to enter a reply to the incoming e-mail in the widget and include a Send button. As for the`linkWidgetsToGui()` method, the `createReply()` method links widgets for managing the sending of the e-mail response.

```
// Linking widgets
sendButton = agentInteractionGui.sendButton;
replyWindow = agentInteractionGui.replyWindow;
responseTextArea = agentInteractionGui.responseTextArea;
```

When the agent has finished the reply, he or she can click the Send button in the reply window, which will do several things, as described here.
First, you will need to get the name of a queue. This is because you are about to send an e-mail interaction, and e-mail interactions must have a queue to go into. After getting a queue name, which will be explained in a moment, you must set the text of the outgoing e-mail based on the text entered by the agent:

```
//Get the queue
String outboundQueue = getOutboundQueues();
// Set the message text to the reply
sampleEmailReply.setMessageText(responseTextArea.getText());
```

Finally, you can send the response and close the reply window.

```
// Send the response
sampleEmailReply.send(outboundQueue);
replyWindow.dispose();
```

As mentioned above, you must supply a queue for the outgoing e-mail. There are several ways to do this. In this example, the available queues are read in and the first one is chosen, as shown in the next stretch of code. You may need to use more sophisticated means to perform this task.

```
String queueId = "";
Collection availableQueues = sampleEmailIn .getAvailableQueuesForChildInteraction();
Iterator iterator = availableQueues.iterator();
Queue queue;
while (iterator.hasNext()) {
   queue = (Queue) iterator.next();
   queueId = queue.getId();
   break;
}
return queueId;
```

That is a lot more work that you had to do in the previous examples, but there is not much left to do now.

Add Event-Handling Code

The `handleInteractionEvent()` method in this example is different from the corresponding method in `SimpleVoiceInteraction`. The significant difference is in how it processes two different types of e-

mail interactions: sampleEmailIn for an incoming e-mail and sampleEmailReply for the response.

> **Important**
>
> As explained in Threading section, you should write short and simple event handlers to avoid delaying the propagation of events.

In this purpose, the `SimpleEmailInteraction` uses `EmailInteractionEventThread` class to process `InteractionEvent` events. All the event processing is performed in the `run()` method of the thread.

```
public void handleInteractionEvent(InteractionEvent event) {
   super.handleInteractionEvent(event);
   EmailInteractionEventThread p = new EmailInteractionEventThread(event);
   p.start();
}
```

When an e-mail comes in, it will have a status of `RINGING`. At this point, if the `sampleEmailIn` interaction associated with the example is `null`, the example gets the event's e-mail interaction and displays a few details in the GUI, as shown here:

```
if (sampleEmailIn == null
   && event.getSource() instanceof InteractionMailIn
   && event.getStatus() == Interaction.Status.RINGING) {

   sampleEmailIn = (InteractionMailIn) event.getSource();

   //Displaying From and Subject fields
   String emailText = "From: "
      + sampleEmailIn.getFromAddress().toString()
      + "\nSubject: " + sampleEmailIn.getSubject();
   inboundEmailTextArea.setText(emailText);
}
```

At this point, if the received event provides notification of a change of status for the `sampleEmailIn` interaction, the `handleInteractionEvent()` method checks for two cases:

- If the e-mail is in `TALKING` status, the agent agreed to process it; `sampleEmailInteraction` reads information from the incoming e-mail and places it in the GUI by calling the displaySampleEmailIn() method.
- If the e-mail is in `IDLE` status, it marks it done (if necessary) and removes it from the example.

```
//The event involves the inbound email being processed
if(sampleEmailIn!= null
&& event.getSource().getId() == sampleEmailIn.getId())
{
   // The inbound email is in talking status, and can be displayed
   if (event.getStatus() == Interaction.Status.TALKING)
   {
      sampleEmailIn = (InteractionMailIn) event.getSource();
      displaySampleEmailIn();
   }
   // The interaction is processed,
```

```
   // the sample no longer needs to handle it
   else if (event.getStatus() == Interaction.Status.IDLE )
   {
      if(!sampleEmailIn.isDone())
      sampleEmailIn.markDone();
      sampleEmailIn = null;
   }
   setInteractionWidgetState();
}
```

Then, the event can also provide notification of a change of status for the sampleEmailReply interaction. In this case, the method checks whether or not this e-mail is released. If `true`, this interaction and its widgets are removed from the example, as shown here.

```
else if(sampleEmailReply!= null && event.getSource().getId() == sampleEmailReply.getId())
{
        if (event.getStatus() == Interaction.Status.IDLE )
        {
                if(!sampleEmailReply.isDone())
                        sampleEmailReply.markDone();
                sampleEmailReply = null;
                sendButton = null;
                replyWindow = null;
        }
        setInteractionWidgetState();
}
```

Synchronize the Widgets

As said previously, the standalone code examples use two similar methods to synchronize their user interface widgets with the application state: `setPlaceWidgetState()` and `setInteractionWidgetState()` .
`SimpleEmailInteraction` overwrites both methods:

- `setPlaceWidgetState()` to update login buttons in the `Simple Place` panel, according to the e-mail media's possible actions.

```
loginButton.setEnabled(sampleEmail.isPossible(Media.Action.LOGIN));
```

- `setInteractionWidgetState` to update the interaction buttons in the `SimpleE-Mail Interaction` panel, according to the actions available on the e-mail inbound interaction.

```
acceptButton.setEnabled(sampleEmailIn.isPossible(InteractionMailIn.Action.ANSWER_CALL));
```

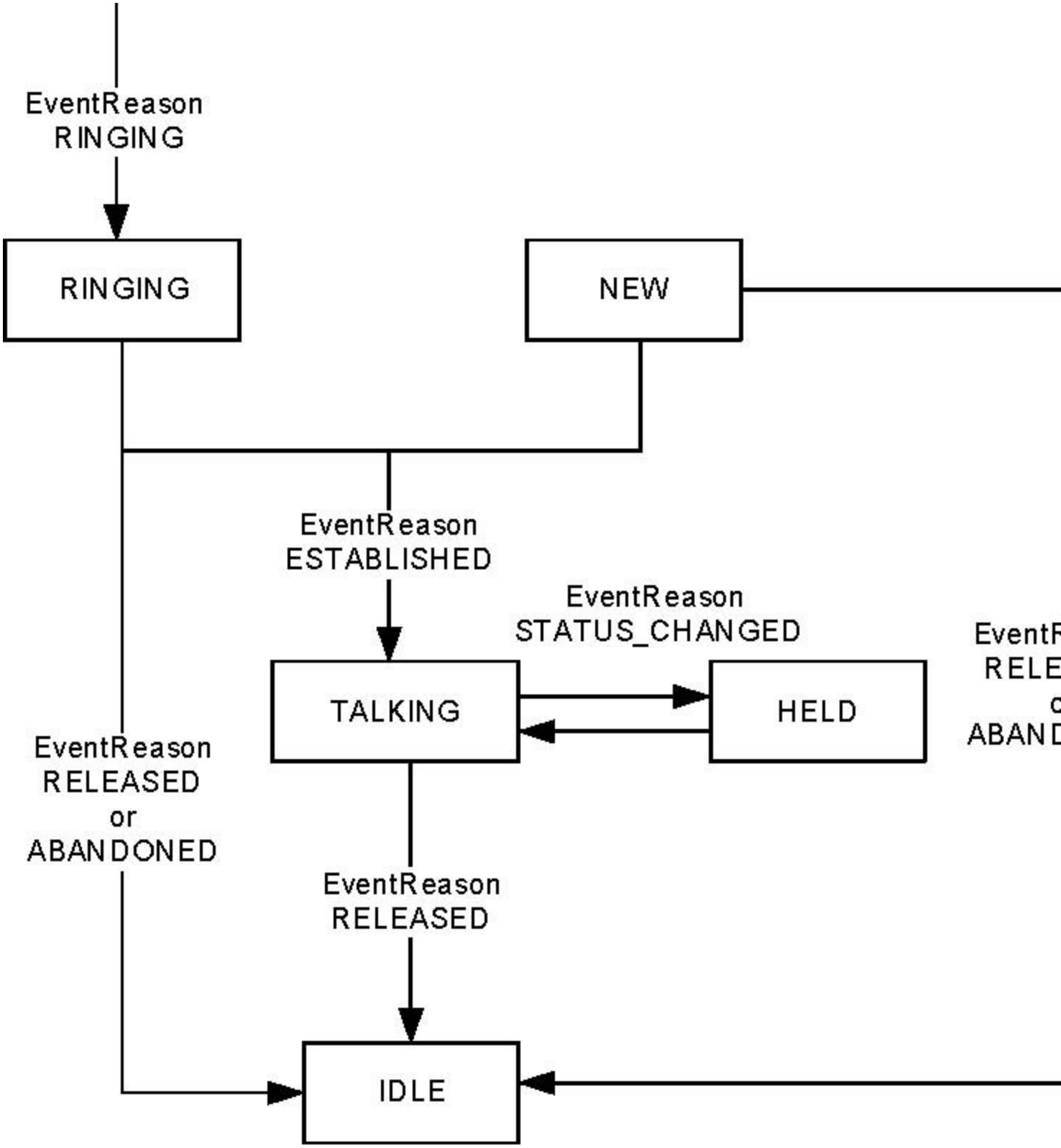
As you can see, the work you do to reply to an e-mail is not that different from what you do to handle a call. The following sections go into more detail about e-mail interactions.

Handling an E-Mail Interaction

To access e-mail interactions, an agent must log into the e-mail media of his or her place. Due to consolidation of the interactions, some method names and events applied to the manipulation of e-mail are drawn from telephony terminology.

E-Mail State

The e-mail state diagram below, E-Mail State, shows the life cycle of an e-mail interaction (inbound or outbound). The workflow is similar to that of a voice phone call, but without pure voice operations. States are `Interaction.Status` values and transitions are `InteractionEvent.EventReason` values.


EventReason
RINGING
RINGING
NEW
EventReason
ESTABLISHED
EventReason
STATUS_CHANGED
TALKING
HELD
EventReason
RELEASED
or
ABANDONED
EventReason
RELEASED
IDLE

E-Mail State

Sending an E-Mail

To send an e-mail, create an outgoing e-mail interaction, that is, an `InteractionMailOut` instance of type `Interaction.Type.EMAILOUT`.

Creating an e-mail is similar to creating a phone call. You first create an interaction with the `Agent.createInteraction()` (or `Place.createInteraction()`) method with type `EMAIL`, as shown in the following code snippet:

```
AilFactory factory = AilLoader.getAilFactory();
Agent agent = (Agent) factory.getPerson( AgentId );
Place place = (Place) sAF.getPlace(PlaceId);

HashSet myMedia = new HashSet();
myMedia.add(MediaType.EMAIL.toString());
agent.loginMultimedia(place, myMedia, ActionCode.Type.LOGIN.toString(),
"Login e-mail" );

//...

InteractionMailOut mailOut =
(InteractionMailOut) agent.createInteraction(MediaType.EMAIL, null, Queue);
```

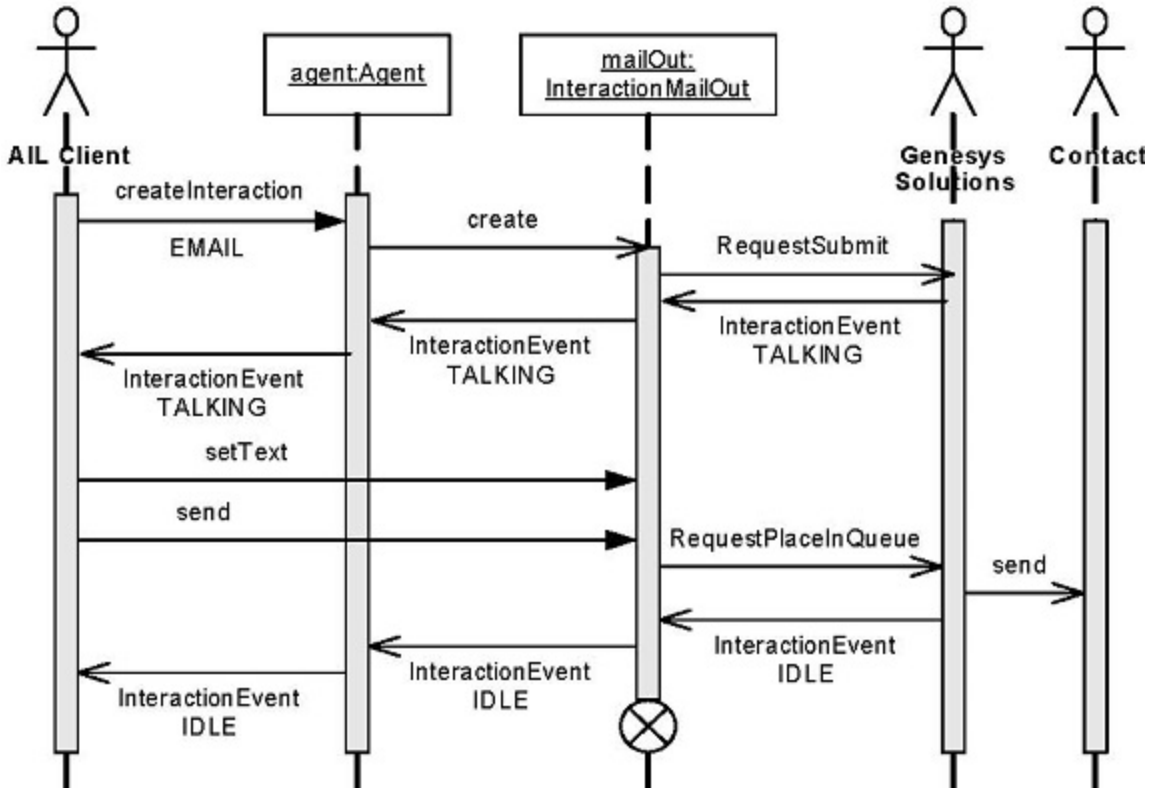
Then, you immediately receive an `InteractionEvent` event, setting your interaction to the `Interaction.Status.TALKING` . You are now ready to fill in the e-mail with the methods of `InteractionMailOut` .
To set addresses, you must create and fill an `EmailAddress` object for each e-mail address, as shown here:

```
EmailAddress[] myAddresses = new EmailAddress[1];
myAddresses[0] = sAF.createEmailAddress("My Contact","myContact@company.com");


mailOut.setToAddresses( myAddresses );
mailOut.setSubject( "Your Subject" );
mailOut.setMessageText( "Your BodyText" );
```

Once finished, you call the InteractionMailOut.`send()` method to send your e-mail to the Genesys servers.

```
mailOut.send( Queue ); //you have to specify the queue
```

Invoking `send()` automatically releases your interaction. You receive an `InteractionEvent` event propagating the interaction status change to `Interaction.Status.IDLE`.
The following diagram presents event flow for sending an e-mail.



Sending an E-Mail

Receiving an E-Mail

When your agent receives an incoming e-mail, your application receives an `InteractionEvent` event, giving an interaction of type `Interaction.Type.EMAILIN` with the status `RINGING` . This interaction corresponds to the incoming e-mail and is available through the `InteractionMailIn` interface.

```
void handleInterationEvent( InteractionEvent event ) {
InteractionMailIn mailIn =
(InteractionMailIn) event.getSource();
// ...
}
```

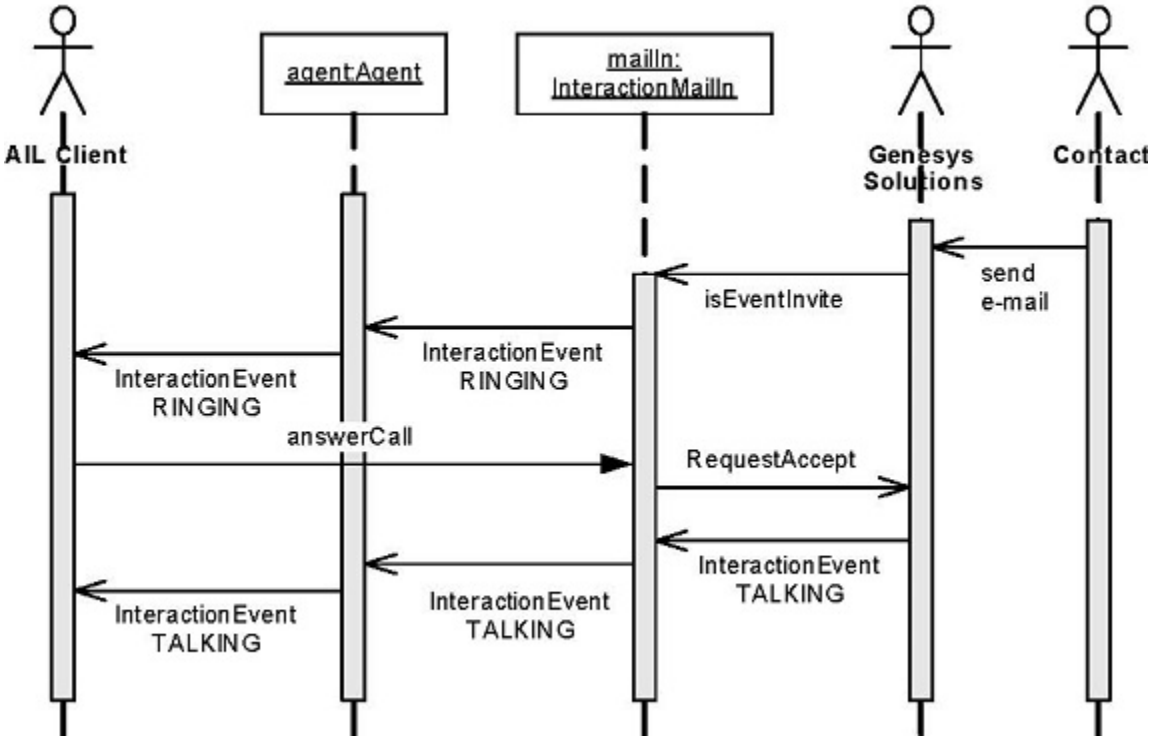
After invoking the `answerCall` () method to acknowledge the handling of the interaction, you receive another `InteractionEvent` event propagating the status of the interaction changed to `TALKING`.

```
mailIn.answerCall( null );
```

You can now access the incoming e-mail content.

```
System.out.println( "\nFrom: "+ mailIn.getFromAddress().toString() +"\nSubject: "+
mailIn.getSubject() +"\nText: "+ mailIn.getMessageText());
```

The following sequence diagram presents event flow for sending an e-mail.



Answering an E-Mail

Responding to an E-Mail

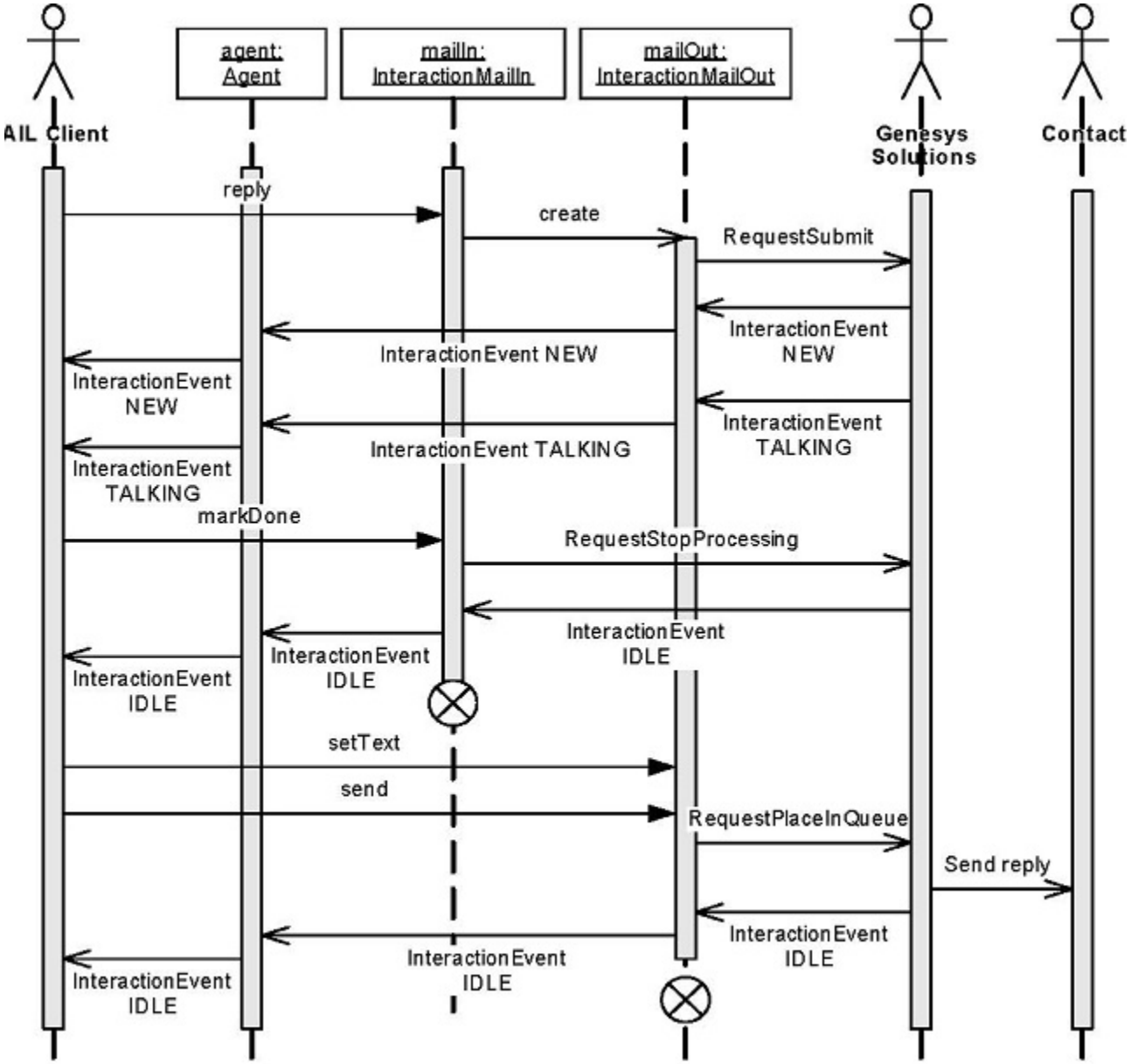
Responding to an e-mail is as straightforward as sending one. You can call the `InteractionMailIn.reply()` method to initiate a reply if the `InteractionMailIn.Action.REPLY` is available; you test this by invoking `isPossible()`. In the following code snippet, the Agent Interaction Java API creates a new interaction of type `Interaction.Type.EMAILOUT_REPLY` in `TALKING` status, and sends it to you through an event `InteractionEvent`, or as a result of the method.

```
//Create a reply, with auto mark done of the inbound e-mail
InteractionMailOut mailOutReply = mailIn.reply( Queue, false, true);
```

Above, the `InteractionMailIn` instance is automatically marked done, so it is no longer available. If you wish to keep alive the `InteractionMailIn` instance for making further replies, you set the `auto-mark-done` parameter to `false`. Later, to terminate this interaction, your application marks it done, as shown here:

```
//Create a reply, with no auto mark done of the e-mail in
InteractionMailOut mailOutReply = mailIn.reply( Queue, false, false);

//...

mailIn.markdone();
```

The `InteractionMailOut` reply is handled exactly in the same way as sending an e-mail. As soon as your application calls the `InteractionMailOut`.send() method, AIL places the request in the specified queue and terminates the `InteractionMailOut` reply. Your application does not need to mark done this interaction.



Responding to an E-Mail

Handling Collaborative E-Mail Interactions

When an agent is working on an outgoing e-mail, he or she can request the collaboration of other agents to elaborate the e-mail content.
A collaboration session involves several types of collaborative interactions. A collaborative interaction is an e-mail interaction that manages additional collaboration data.
During a collaboration session, your application can:

- Initiate a collaboration session on an outgoing e-mail.
- Participate in a collaboration session.

When an agent initiates the collaboration, he or she sends invitations to the participants. By periodically refreshing the status of the invitations, your application can monitor the participants' collaboration activity.
If the agent is a participant, your application manages `InteractionEvent` events for `Interaction.Status` changes and performs collaboration actions both on the invitation and on the reply that is sent as a result of the collaboration.
Classes and interfaces dedicated to the collaboration are part of the `com.genesyslab.com.ail.collaboration` package.

Types of Collaborative E-Mail Interactions

Types for collaborative e-mail interactions are accessed with the inherited `Interaction.getType()` method. The following table presents the types of collaborative e-mail interactions that your application can deal with.

Types of Collaborative E-Mail Interactions

Interactions	Interaction.Type	Interface	Description
Parent invitation	`COLLABORATION_INVIT_OUT`	`InteractionInvitationOut`	Interactions for sending invitations to participants.
Invitation seen by the parent	`COLLABORATION_INVIT_IN`	`InteractionInvitation ParentIn`	Interactions for invitations sent by the agent (or parent) who requested collaboration.
Incoming invitation	`COLLABORATION_INVIT_IN`	`InteractionInvitationIn`	Interactions for incoming invitations received by the participant (or child).
Reply to invitation	`COLLABORATION_REPLY_OUT`	`InteractionReplyOut`	Interactions for collaborative

Interactions	Interaction.Type	Interface	Description
			replies sent by participants (or children).

Outgoing Invitations

As mentioned in the Agent Interaction SDK (Java) API Reference, the collaborative `InteractionInvitationOut` interaction is used to set a list of participants in the collaboration session. This interaction creates and sends incoming invitation interactions to each participant on this list. You need a single `InteractionInvitationOut` interaction to send invitations to several participants.

> **Warning**
> As soon as invitations are sent, the outgoing invitation is no longer available. Nor is it stored in the database. Do not keep any reference on this object.

Invitation

Invitation interactions are inbound e-mails of type `COLLABORATION_INVIT_IN` . Depending on your application's role in the collaboration (child or parent), the application provides you with two interfaces to handle invitations:

- `InteractionInvitationIn` —child invitation:
  - Each participant in the collaboration session receives an incoming child invitation.
  - This interaction informs the participant of the collaboration request.
  - For information about child invitation management, see Participating in a Collaboration Session.
- `InteractionInvitationParentIn` —parent invitation:
  - For each invitation sent to a participant, the agent who initiates the collaboration can access the corresponding invitation interaction.
  - The agent uses parent invitations to monitor the collaboration and the participants' replies.
  - Each invitation is a child interaction of the initial outgoing e-mail interaction.
  - Use `InteractionMailOut.getSentInvitations()` to retrieve parent invitations.
  - For information about parent invitation management, see Handling a Collaboration Session.

Collaborative Reply

The collaborative reply is an outgoing e-mail replying to a child incoming invitation. It is created by calling the `InteractionInvitationIn.reply()` method. As for an outgoing replying e-mail, some e-mail fields are already filled in and you just have to set new text with

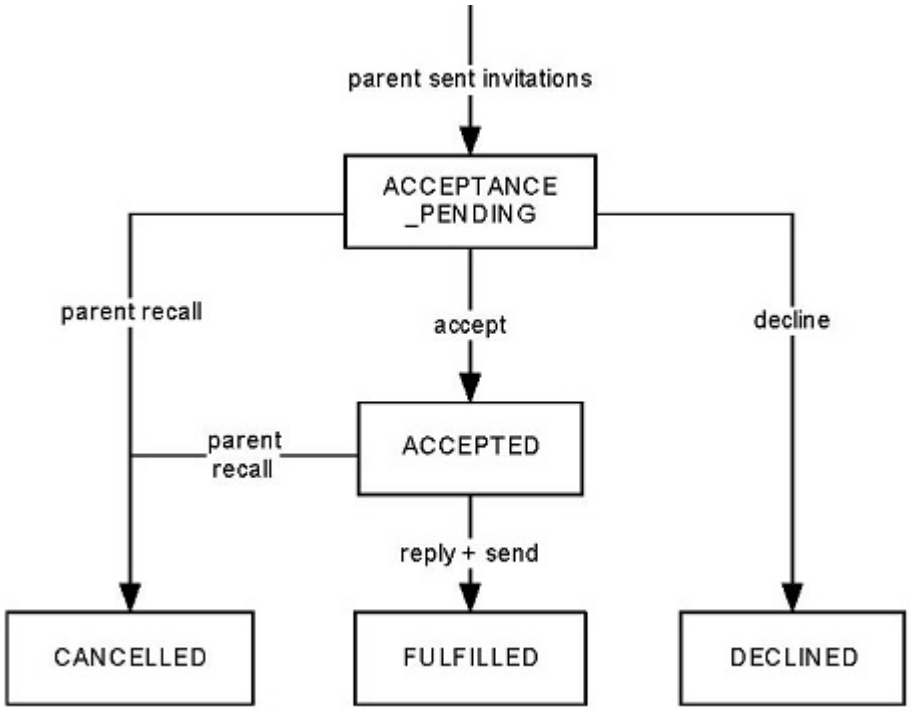
`theInteractionReplyOut.setMessageText()` method.
Use a collaborative reply in the same way that you would use a reply outgoing e-mail interaction. For further details, see Participating in a Collaboration Session.

Collaboration Status

Collaborative interactions all have a collaboration status—described in the `InteractionInvitationIn.Status` inner class—available through the `getCollaborationStatus()` method.
Applications that initiate the collaboration have specific interests in the collaboration status of their parent invitations.When a parent invitation gets a `FULFILLED` collaboration status, the application can get the reply corresponding to the invitation by calling the `getCollaborativeReply()` method.

> **Important**
> Collaboration status changes do not launch additional `InteractionEvent` events. There is no notification of modifications through the API. Periodically read the collaboration status to check any status change.

The following state diagram shows non-exhaustive transitions that exist between collaboration statuses of an incoming invitation.



Transitions Between Statuses of an Incoming Invitation

> **Warning**
> This figure is provided as an informative example. It does not include all possible statuses and transitions.

You must take into account an invitation's collaboration status to perform collaboration actions.

Handling a Collaboration Session

From the parent's point of view, the collaboration feature has been designed as a set of invitations that are children of the outgoing e-mail interaction that requests a collaboration session.
This section discusses first the sending of invitations to participants, then the management of the parent invitations.

Sending Invitations

First, create a draft of the relevant outgoing e-mail interaction. You need a single InteractionInvitationOut interaction to send invitations to several participants.
Use the `InteractionMailOut.createCollaborationInvitation()` method as shown in the following code snippet:

```
InteractionInvitationOut myInvitationsToSend =
myInteractionMailOut.createCollaborationInvitation();
```

You have to specify the reason for this collaboration session. Use `setSubject()` and `setMessageText()` to add information.
You must add participants to the collaboration session. First, create each participant with the `InteractionInvitationOut.createParticipant()` method, then add those participants to the outgoing invitation with the `InteractionInvitationOut.addParticipant()` method, as shown in the following code snippet:

```
Participant myParticipant0 = myInvitationsToSend.createParticipant();
// Setting type and name of the participant that is agent 0.
myParticipant0.setType(QUEUE);
myParticipant0.setName("agent0");

// Adding agent0 to the list of participants
myInvitationsToSend.addParticipant(myParticipant0);
```

To determine if those participants have been correctly added to the outgoing invitation, you can implement a `ParticipantsListener` listener and add it with the `InteractionInvitationOut.addParticipantsListener()` method.
Once all participants have been added to the outgoing invitation, you can send them invitations. Depending on the method called to send invitations, your application activates a specific mode:

- `send()` —Sends the invitations to the participants in pull mode. The child invitations are available in workbins. See Putting Interactions in Workbins.
- `transfer()` —Transfers the invitations to the participants in push mode. Each participant receives the invitation as an incoming interaction.

The following code snippet uses the `transfer()` method:

```
// Use transfer in the push mode
myInvitationsToSend.transfer();
```

In push mode, on the child side, each participant receives a `RINGING InteractionEvent` for the corresponding `InteractionInvitationIn` . See Participating in a Collaboration Session.

Managing Parent Invitations

For each participant to the collaboration session, you manage a corresponding `InteractionInvitationParentIn` interaction. You can get the set of parent invitations by calling the `InteractionMailOut.getSentInvitations()` method of the outgoing e-mail interaction involved in the collaboration session.

Parent Actions

You can perform parent-specific actions on the invitations, that is, reminding participants about invitations or recalling invitations.
Those parent actions are available in the`InteractionInvitationParentIn.Action` inner class. To determine if the `RECALL` and `REMIND` actions are available, test the collaboration status of the invitation as shown in the following code snippet:

```
if((myInteractionInvitationParentIn.getCollaborationStatus() ==
InteractionInvitationIn.Status.ACCEPTED) ||
(myInteractionInvitationParentIn.getCollaborationStatus() ==
InteractionInvitationIn.Status.ACCEPTANCE_PENDING)
{
   // The parent can take the REMIND or RECALL action on the
   // interaction.
   myInteractionInvitationParentIn.remind(myPlace);
} else {
   // Collaboration status is DECLINED, FAILED, or FULFILLED
   // REMIND or RECALL are not available
}
```

Monitoring Participant Activity

`InteractionInvitationParentIn` interactions are not used as incoming e-mails; they should be used to monitor the participant activity on the invitation.
Test the collaboration status periodically to take changes into account. To get the collaboration status of an invitation, call the `InteractionInvitationParentIn.getCollaborationStatus()` method. When the collaboration status is `InteractionInvitationIn.Status.FULFILLED`, you can get the response of the corresponding participant by calling the `getCollaborativeReply()` method.

```
if(myInteractionInvitationParentIn.getStatus() == InteractionInvitationIn.Status.FULFILLED)
```

```
{
   InteractionReplyOut myCollabReply =
myInteractionInvitationParentIn.getCollaborativeReply();
   System.out.println("The reply is: "+myCollabReply.getMessageText() );
}
```

> **Important**
>
> You cannot start a collaboration session on a collaborative reply. You cannot reply to a collaborative reply.

Participating in a Collaboration Session

In push mode, from the participant (or child) point of view, the agent receives an `InteractionEvent` for a `RINGING` incoming e-mail interaction of type `COLLABORATION_INVIT_IN`. See Putting Interactions in Workbins for details about pull mode. The following code snippet implements the `handleInterationEvent()` method for an `AgentListener`:

```
void handleInterationEvent( InteractionEvent event ) {
Interaction myInteraction = (Interaction) event.getSource();

//...
if(myInteraction.getType() == Interaction.Type.COLLABORATION_INVIT_IN)
{
   //Management of the collaborative invitation
}
   //...
}
```

Cast the interaction associated with the event to `InteractionInvitationIn`, as shown in the following code snippet:

```
InteractionInvitationIn myInvitation = (InteractionInvitationIn) event.getSource();
```

You can manage this incoming interaction as a common e-mail by answering to it. See also Responding to an E-Mail.

Once the interaction is in `TALKING` status, the agent can either accept the collaboration by calling the `acceptInvitation()` method or decline the invitation by calling the `declineInvitation()` method.

If the agent accepts, he or she can reply to the invitation, as shown in the following code snippet:

```
//Getting the interaction for the reply
InteractionReplyOut myReply = (InteractionReplyOut) myInvitation.reply("Place0") ;

//Setting the collaboration message
myReply.setMessageText("My reply is...");

//Sending the message
```

```
myReply.send();
```

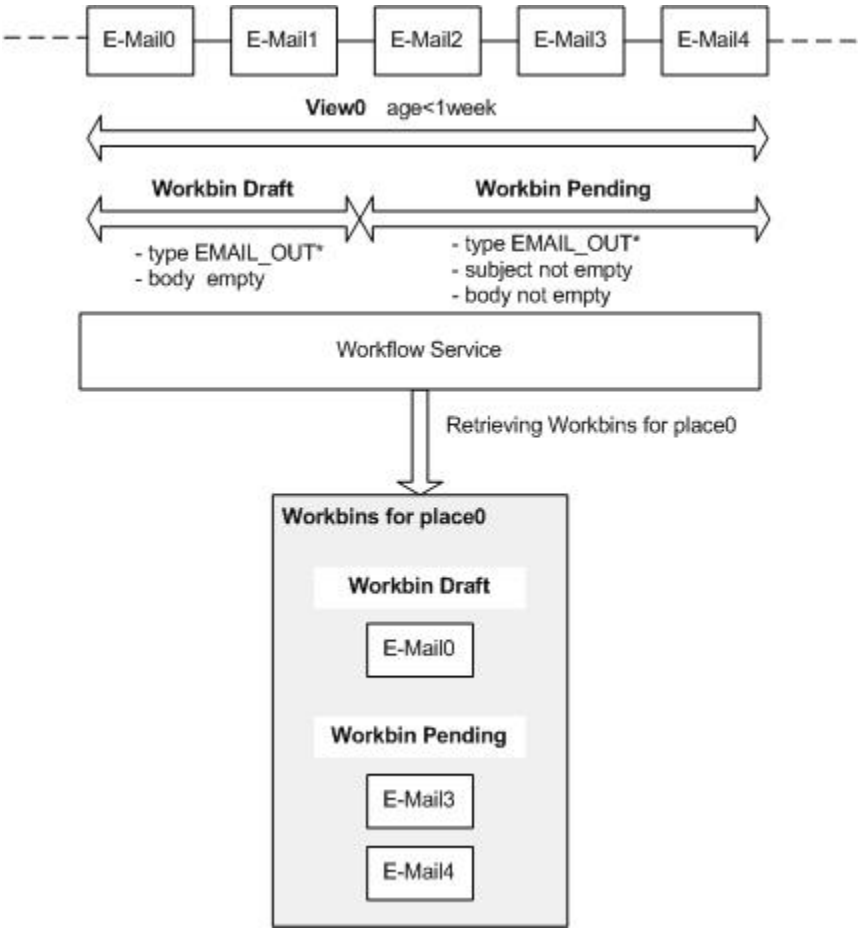
Once the reply is sent, the collaboration status of the corresponding invitation interaction becomes `FULFILLED` and the agent who initiates the collaboration can access the reply. Both the invitation and reply interactions are released.

Handling Workflow

Workflow management is provided with the `WorkbinManager` interface and classes of the`com.genesyslab.ail.workflow` package.

The `WorkbinManager` interface accesses the workbins of a place. From an agent's point of view, a workbin is a sort of interaction directory from which your application can pull, or into which it can put, interactions.

To define workbins more precisely: a queue contains interactions, and a view filters a queue's interactions according to a set of criteria. A workbin filters a view's interactions according to a further set of criteria. The following diagram shows an example of a view and some workbins defined for a queue.



Example for Workbins, Views, and Queues

The above figure shows a queue containing e-mail interactions. For this queue, `View0` lets your application see only e-mail interactions that are no older than a week.

In this view, two workbins coexist: one for draft e-mail interactions, and one for pending e-mail interactions. The `WorkbinManager` interface can use those filters to retrieve a set of interactions organized in workbins for a particular place.

Here, the `WorkbinManager` interface retrieves interactions no older than a week and available for `place0`. `E-Mail1` and `E-Mail2` are not retrieved, as they should not be processed in `place0`.

Use the Configuration Layer to define views and workbins. For further details, refer to your Configuration Layer documentation.

> **Important**
> Workbins can contain multimedia (non-voice) interactions only.

Workbins enable pull mode for multimedia interactions. Interaction status is `IDLE` in a workbin. You have to pull an interaction to change the interaction status and execute actions on the interaction. Use the `Workbin` and `WorkbinManager` interfaces to:

- Display workbins and their filtered interactions.
- Enable an agent to put an interaction in a workbin.

Getting the Workbin Manager

The workbin manager is available on the `Place` interface. Invoke `Place.getWorkbinManager()` on your agent's place to get the workbin manager.

```
WorkbinManager myWorkbinManager = place0.getWorkbinManager();
```

Use the `WorkbinManager` interface to get `Queue` and `Workbin` instances available for the agent. For example, the following code snippet gets the `Workbin` instance corresponding to Draft (see the diagram example for Workbins, Views, and Queues).

```
Workbin myDraftWorkbin = myWorkbinManager.getWorkbin("Draft");
```

Workbin Content

The `Workbin` interface is designed to manage the contents of a workbin as a set of `InteractionMultimediaSummary`. It provides the following methods:

- `getContent()`—retrieves interaction summaries for this particular workbin.
- `getContentForAll()`—retrieves interaction summaries for all agents or places defined for this workbin.

- `getSortedContentForAll()`—retrieves interaction summaries for all agents or places defined for this workbin, sorted by agent or place.

The `InteractionMultimediaSummary` interface is a summary description of an interaction available in a workbin. The corresponding interaction's status is `IDLE`. To execute actions on workbin interactions, you first have to pull those interactions. You cannot work with summaries or interactions retrieved from summaries.

The following code snippet displays the contents of the `draft` workbin for this place.

```
Collection myDrafts = myDraftWorkbin.getContent();
Iterator itDrafts = myDrafts.iterator();
while(itDrafts.hasNext())
{
   InteractionMultimediaSummary myDraft = (InteractionMultimediaSummary) itDrafts.next();
   System.out.println("Type: "+myDraft.getType().toString() + " Subject:"+myDraft.getSubject()
     + " Date:"+myDraft.getDateReceived().toString()+"\n");
}
```

The `Workbin` interface has methods to display information about the workbin itself—for instance, its name with `getName()`, the name of the associated queue with `getQueue()`, and its type with `getType()`.

Putting Interactions in Workbins

You can put an `InteractionMultimedia` into a workbin by calling the `Workbin.put()` method, as shown here:

```
/// Creating an outgoing e-mail interaction
InteractionMailOut mailOut = (InteractionMailOut) agent.createInteraction(MediaType.EMAIL,
null, Queue);
// Putting the interaction in the draft workbin
myDraftWorkbin.put(mailOut);
```

Two types of workbins coexist:

- Agent workbin—The `AgentWorkbin` interface allows your application to put interactions into the same workbin defined for other agents, or to get the contents of this workbin for another agent.
- Place workbin—The `PlaceWorkbin` interface allows your application to put interactions into same workbin defined for other places, or to get the contents of this workbin for another place.

You can cast the workbin according to its type, as shown here:

```
if( myDraftWorkbin.getType() == Workbin.Type.AGENT)
{
   AgentWorkbin myDraftAgentWorkbin = (AgentWorkbin) myWorkbinManager.getWorkbin("Draft");

   ///....
   // Creating an e-mail
   InteractionMailOut mailOut = (InteractionMailOut) agent.createInteraction(MediaType.EMAIL,
null, Queue);
   // Getting Agent2 interface
   Agent agent2 = (Agent) myAilFactory.getPerson("Agent2");
```

```
   // Putting the interaction in the draft workbin of agent2
   myDraftAgentWorkbin.put(mailOut, agent2);
}
```

During a collaboration session, the inviting agent can choose to use pull mode when sending his or her invitations. When setting the participant list of an `InteractionInvitationOut`, he or she can activate the pull mode. When the parent sends the outgoing invitation by calling `InteractionInvitationOut.send()`, participants of type `Participant.Type.AGENT` get their invitation in their workbin. They have to pull the invitation interaction to process it.

```
InteractionInvitationOut myInvitationsToSend =
myInteractionMailOut.createCollaborationInvitation();

// Setting type and name of the participant that is agent 0.
myParticipant0.setType(AGENT);
myParticipant0.setName("agent0");

// Adding agent0 to the list of participants
myInvitationsToSend.addParticipant(myParticipant0);

// Sending invitations: participant0 receives his or her invitation
// in his or her workbin
myInvitationsToSend.send();
```

Pulling Interactions

You can pull an interaction from its `InteractionMultimediaSummary` to a place by calling `pullInteractionMultimedia()`.

You can pull an interaction with the corresponding identifier by calling `Agent.openInteraction()` or `Place.openInteraction()`.

The `InteractionMultimedia` interaction is pulled on the agent or the place. Your application receives an `InteractionEvent` to update the `Interaction.Status` status which changes from `IDLE` in the workbin to `TALKING` once pulled.
Further processing of pulled interactions does not differ from the use cases described in the earlier sections.

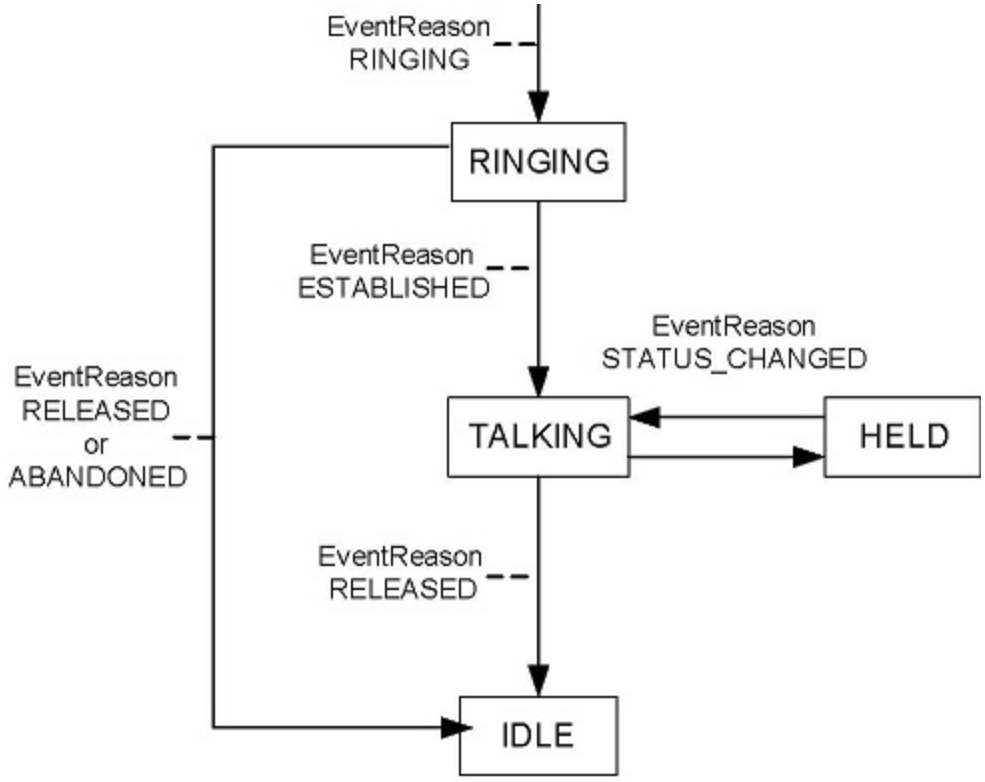
Chat Interactions

Chat interactions are a type of multimedia interactions, that make use of the `InteractionChat` interface, and inherit functionality from `InteractionMultimedia` interface. Other multimedia interactions are discussed in E-Mail Interactions and Open Media Interactions.
This chapter discusses chat interactions. It also presents `SimpleChatInteraction`, a new example that allows user to join chat sessions, send messages, and use the CoBrowse feature.

Chat Interaction Design

Chat State

Because `InteractionChat` inherits the `InteractionMultimedia` interface, each `InteractionChat` object has a status, described in the `Interaction.Status` inner class. The following diagram shows the possible states of a chat interaction. The only chat interactions are incoming.



Chat State Diagram

For a chat interaction, the interaction status can change due to a common`Interaction.Action`, that is, a call to the corresponding method. For example, a successful `Interaction.Action.ANSWER_CALL` action changes the interaction status from `Interaction.Status.RINGING` to `Interaction.Status.TALKING`. The corresponding method is `InteractionChat.answerCall()`.

Not all `Interaction.Action` actions are available on chat interactions. `InteractionChat` inherits `Possible`. Test if an action is possible on a chat interaction by calling`InteractionChat.isPossible(Interaction.Action)`.

If the chat interaction has a `TALKING` status, the chat interaction is active in the chat session and you can take chat-specific actions on the interaction, by using methods such as:

- `sendMessage()`
- `conferencePlace(), conferenceAgent()`
- `clearCall()`

Details about these and other methods are provided in the Handling a Chat Interaction section.

CoBrowse Interactions

The Agent Interaction (Java API) provides a CoBrowse feature through the `InteractionCoBrowse`

interface used as a container to store URLs that your application can share with a customer at runtime. The stored URLs are then intended to be used in contact histories. For further details about the History feature, see Contact History.
The CoBrowse feature does not include any web management and is fairly simple to use. You create an `InteractionCoBrowse` instance by calling a standard `createInteraction()` method; most of the time, you will need a CoBrowse interaction when you are handling another interaction (of any type, that is e-mail, chat, voice, or open media.) To link your `InteractionCoBrowse` instance to an existing interaction you specific a parent interaction in parameters at its creation.

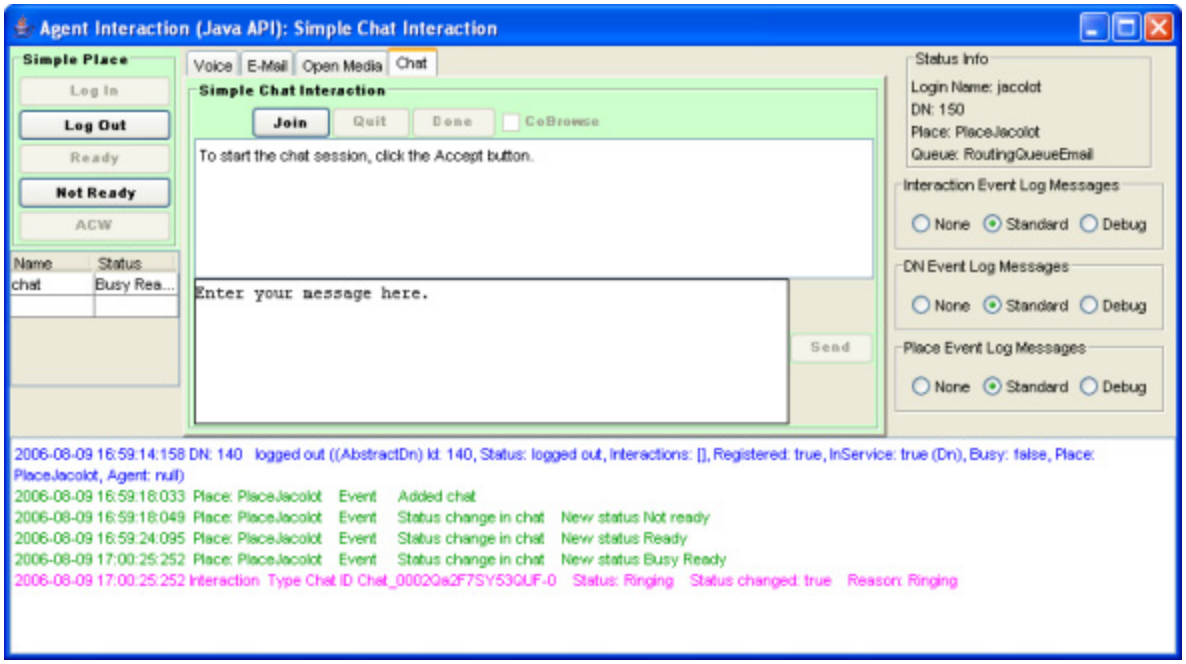
As this interaction is a used as a container, this `InteractionCoBrowse` instance has no status to monitor by handling events. Then, all your application is responsible is saving the `InteractionCoBrowse` instance.

The `SimpleChatInteraction` example includes a CoBrowse feature. For further details, refer to Add CoBrowse-Handling Code.

SimpleChatInteraction

This example is similar to `SimpleVoiceInteraction`, which was introduced earlier. It uses the same graphical user interface and the same internal structure, inheriting from `SimplePlace`. When you launch this example, which is in the `StandAloneExamples` directory, you will see the user interface presented in the screen shot below.

> ### Important
> For the sake of simplicity, this example is designed to handle one chat session at a time. Set up a capacity rule limiting the agent to a single chat interaction at a time in your routing strategy. For further details, see *Universal Routing 7.6 Documentation*.



Join a Chat Session

If the user checks the `CoBrowse` checkbox, `SimpleChatInteraction` opens a dialog box which displays the content of the child CoBrowse interaction created for the example. Then, `SampleChatInteraction` parses all the received chat messages, adds any URL found in the chat text to the `CoBrowse` interaction, and finally saves this interaction when the chat interaction terminates.

Set up Button Actions

By inheritance, `SimplePlace` takes care of all agent buttons located in the `Simple Place` panel. `SimpleChatInteraction` is in charge of chat buttons designed to manage the chat session, that is, `Join` a ringing chat session, `Quit` an established session, `Done` to mark the corresponding chat interaction as `Done` . The Send button enables the agent to send a chat message entered in the message panel.
The GUI program `AgentInteractionGui` has created buttons for each of these actions, but at this point they do nothing. `SimpleChatInteraction` rings these buttons to life by overloading the `linkWidgetsToGui()` method.
The following code snippet shows how to implement the Send button. The corresponding button action uses the contents of the chat message text area to send a message, then, it clears this text area.

```
// Add a send button for the chat session
sendButton = agentInteractionGui.sendChatMsgButton;
sendButton.setAction(new AbstractAction("Send") {

public void actionPerformed(ActionEvent actionEvent) {
 try {
   String msg = chatMsgTextArea.getText();
   sampleChatIxn.sendMessage(msg);
```

```
   chatMsgTextArea.setText("");
 } catch (Exception exception) {
   agentInteractionGui.writeLogMessage(exception.getMessage(),"ErrorEvent");
 }
}});
```

Add Event-Handling Code

`SimpleChatInteraction` is designed to handle chat 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.
In this purpose, the `SimpleChatInteraction` uses two threads:

- `ChatSessionInteractionEventThread` to handle `InteractionEvent` events sent for the `InteractionChat` used to process the chat session.
- `InteractionChatEventThread` to handle `ChatEvent` events sent for processing messages during the chat session.

The event-handling code in `ChatSessionInteractionEventThread` is similar to the event-handling code in `VoiceInteractionEventThread`. It checks several statements to handle status changes in the processed `InteractionChat` instance (that corresponds to the chat session.) For further details, see Add Event-Handling Code.

The event-handling code in `InteractionChatEventThread` is in charge of displaying chat messages and information about users in the chat panel.

```
if(chatEvent.getEventType() == InteractionChatEvent.Type.MESSAGE_RECEIVED)
{
        displayInteractionChatMessage(chatEvent.getParty(), chatEvent.getText());
        if(sampleCoBrowseIxn != null)
        {
                checkURLs(chatEvent.getText());
        }
}
else if(chatEvent.getEventType() == InteractionChatEvent.Type.DISCONNECTED)
{
        agentInteractionGui.writeChatMessage(" ", "You are disconnected !",
AgentInteractionGui.ELSE_STYLE);
}
else if(chatEvent.getEventType() == InteractionChatEvent.Type.USER_JOINED)
{
        agentInteractionGui.writeChatMessage( chatEvent.getDate().toGMTString(),
chatEvent.getParty().getNickName() + " joined", AgentInteractionGui.ELSE_STYLE);
}
else if(chatEvent.getEventType() == InteractionChatEvent.Type.USER_LEFT)
{
        agentInteractionGui.writeChatMessage( chatEvent.getDate().toGMTString(),
chatEvent.getParty().getNickName() + " left", AgentInteractionGui.ELSE_STYLE);

}
else if(chatEvent.getEventType() == InteractionChatEvent.Type.USER_REENTER)
{
        agentInteractionGui.writeChatMessage( chatEvent.getDate().toGMTString(),
chatEvent.getParty().getNickName() + " reentered", AgentInteractionGui.ELSE_STYLE);
}
```

Chat events do not affect the status of the chat interaction. That's why interaction widgets don't update on chat events.

Add CoBrowse-Handling Code

In this code example, CoBrowse is started when the user checks the CoBrowse checkbox. At that moment, `SimpleChatInteraction` calls its `startCoBrowse()` method which creates an `InteractionCoBrowse` instance for the current chat session, as shown here:

```
try {
//1. Create a new InteractionCoBrowse
  sampleCoBrowseIxn = (InteractionCoBrowse) sampleAgent.createInteraction(MediaType.COBROWSE,
sampleChatIxn, sampleChatIxn.getQueue());
} catch (RequestFailedException e) {
   agentInteractionGui.writeLogMessage(e.getMessage(), "ErrorEvent on CoBrowse creation");
}
```

Then, when the application gets a chat message, it parses the message content by calling the `checkURLs()` method. If this method finds a URL in the text, it adds the URL to the `InteractionCoBrowse` instance.

```
test[i] = "http:"+test[i];
agentInteractionGui.coBmodel.addElement(test[i]);

try {
  sampleCoBrowseIxn.addURLs(new String[]{test[i]});
} catch (RequestFailedException e) {
  agentInteractionGui.writeLogMessage(e.getMessage(),"ErrorEvent");
}
```

When the chat interaction is terminated, that is, in `IDLE` status, the associated CoBrowse interaction is saved and marked as done.

```
sampleCoBrowseIxn.save();
sampleCoBrowseIxn.markDone();
```

Handling a Chat Interaction

Chat interactions are multimedia interactions that allows an agent to manage, or participate in, a chat session. You need a single chat interaction to let your agent take part in the chat session. The chat interaction receives events for message exchanges.
Chat interactions are available in the `InteractionChat` interface of the `com.genesyslab.ail` package. The following sections detail how to use this interface.

Entering a Chat Session

You can participate in a chat session if your `Agent` object has successfully logged into a `CHAT` media in its place. Through a registered `AgentListener` of your `Agent` object, you are notified of a chat session

request by receiving an `InteractionEvent` event about an `InteractionChat` object, in`Interaction.Status.RINGING`.

```
void handleInteractionEvent( InteractionEvent event ) {
  // ... Check if it is the awaited interaction
  InteractionChat myChatInteraction = (InteractionChat) event.getSource();
  // ...
}
```

To take part in the chat session, invoke the `answerCall()` method. If the action is successful, your application receives an `InteractionEvent` event, showing that the interaction is now in state `TALKING`. See Processing a Chat Interaction.

The agent is now one party to the chat session and the chat interaction is active in the chat session.

> **Important**
>
> Assign a nickname to the agent with the `InteractionChat.setNickName()` method before answering the interaction. If you do not assign a nickname, the nickname is the agent's user name.

Chat Parties

The parties of the chat session are available through the `InteractionChat.getParties()` method. A `ChatParty` object describes the nickname and visibility of each party.

Handling Chat Events

To handle discussion, chat interactions send text messages and receive `InteractionChatEvent` event with a registered `InteractionChatListener`. These events also propagate chat errors and party changes during the chat session. The `InteractionChatEvent`.Type inner class lists the possible `InteractionChatEvent` types.

The following code snippet implements an `InteractionChatListener` class:

```
class ExampleChatListener implements InteractionChatListener {
  public void handleInteractionChatEvent (InteractionChatEvent chatEvent)
  {
      /// Management of the chat event
  }
}
```

To receive `InteractionChatEvent` events, register your `InteractionChatListener` on your `InteractionChat`. When registering, you can get all the events exchanged during the chat session before you are connected, as shown in the following code snippet:

```
InteractionChatEvent[] allPreviousEvents = myChatInteraction.addChatListener(new
ExampleChatListener(), true); // previous events are returned.
```

> **Important**
>
> You can also get all these events after registration, by calling the `InteractionChat.getEvents()` method.

Handling Chat Messages

To send a message, call the `sendMessage()` method of the `InteractionChat` interface. The message is sent to all parties of the chat session.

```
myChatInteraction.sendMessage( "This is a chat message" );
```

Incoming `InteractionChatEvent` events of type `InteractionChatEvent.Type.MESSAGE_RECEIVED` correspond to chat messages. To read the message, use the `getText()` method of the `InteractionChatEvent` that is sent to your `InteractionChatListener`.

```
void handleInteractionChatEvent(InteractionChatEvent chatEvent ) {
        // Testing if the event is a chat message
        if(chatEvent.getEventType() == InteractionChatEvent.Type.MESSAGE_RECEIVED)
        {
                // Displaying the message
                String message = chatEvent.getText();
                String sender = chatEvent.getParty().getNickName();
                System.out.println("From: "+sender+"\n>"+message+"\n");
        }
}
```

> **Important**
>
> You can also access all received messages of the session by calling the `InteractionChat.getMessages()` method.

Handling Typing

To notify the parties that the user is typing a message, call the `InteractionChat.typingStarted()` method of the `InteractionChat` interface. The `InteractionChatEvent.START_TYPING` event is sent to all parties of the chat session.

```
myChatInteraction.typingStarted();
```

Incoming `InteractionChatEvent` events of type `InteractionChatEvent.Type.TYPING_STARTED` correspond to that typing notification. To get the name of the party who is typing, use the

getParty() method of the InteractionChatEvent.

```
void handleInteractionChatEvent(InteractionChatEvent chatEvent ) {
        // Testing if the event is a chat message
        if(chatEvent.getEventType() == InteractionChatEvent.Type.TYPING_STARTED        )
        {
                // Displaying the message
                String sender = chatEvent.getParty().getNickName();
                System.out.println(sender+" is typing...");
        }
}
```

If the user stops (without submitting the message), invoke the InteractionChat.typingStopped() method to notify other parties. Parties will receive the InteractionChatEvent.Type.TYPING_STOPPED event.

Push URL

Your application can now push a URL to the chat applications of other parties by calling the InteractionChat.pushURL() method.

```
myChatInteraction.pushURL("http://genesyslab.com");
```

The chat application for each participant then receives the InteractionChatEvent.Type.PUSH_URL event, which contains the pushed URL, which can be retrieved from the InteractionChatEvent.getText() method.

```
void handleInteractionChatEvent(
                InteractionChatEvent chatEvent ) {
        // Testing if the event is a chat message
        if(chatEvent.getEventType() == InteractionChatEvent.Type.PUSH_URL)
        {
                // Getting the URL
                String message = chatEvent.getText();
                // ... Display the URL ...
        }
}
```

Conferencing

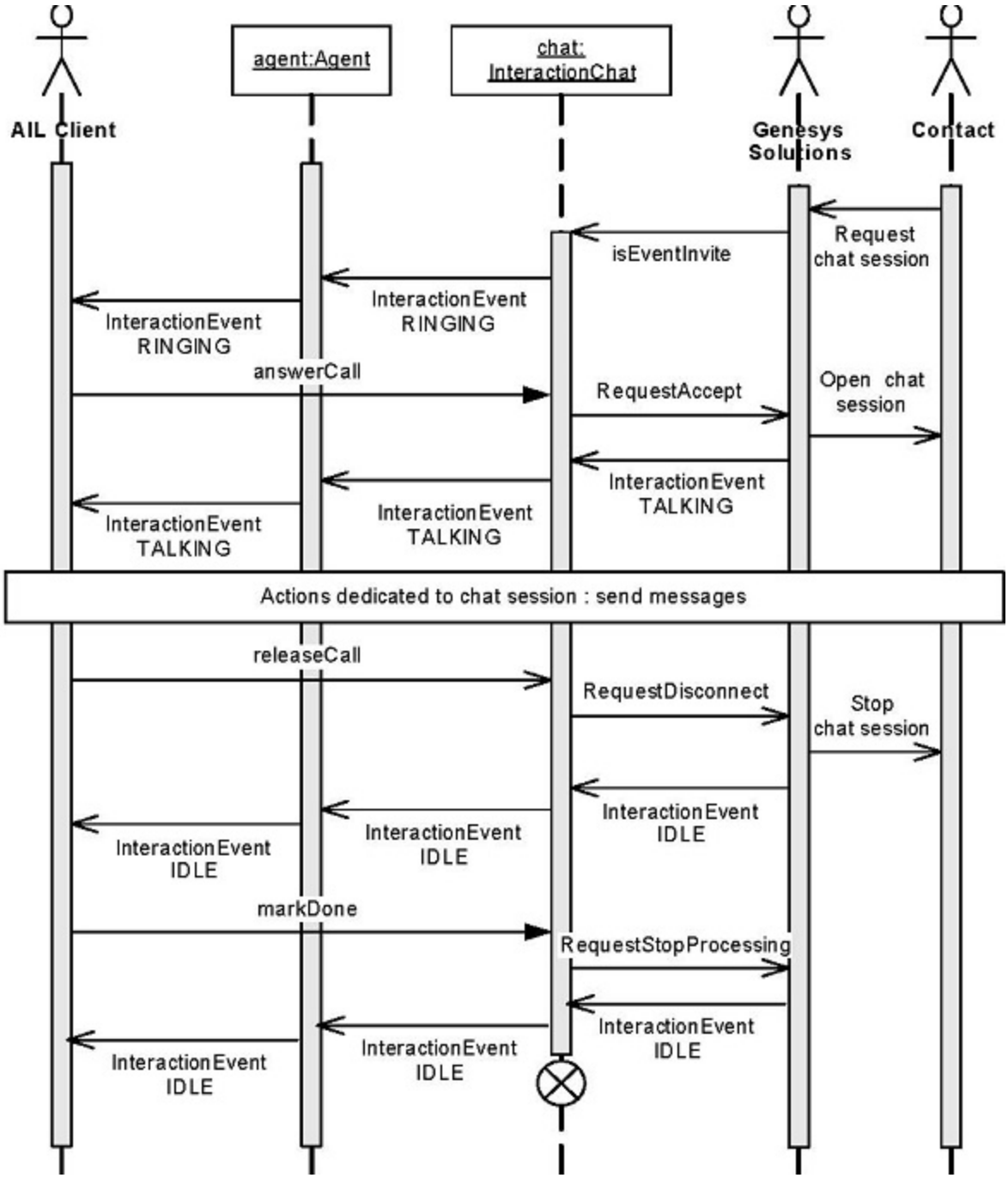
The conference feature allows an agent to invite another agent to join the chat session by using the InteractionChat.conferenceAgent() or InteractionChat.conferencePlace() method.
The following code snippet creates a chat conference with the agent agent1.

```
myChatInteraction.conferenceAgent("agent1", //agent who should join.
ChatParty.Visibility.INT,        // only agents can see him or her.
"reason for joining"); //a string reason.

The agent agent1 receives an InteractionEvent for a InteractionChat in
Interaction.Status.RINGING. If this agent answers the chat interaction, he joins the chat
session.
```

Terminating the Chat Session

To disconnect the chat session, invoke the `InteractionChat.releaseCall()` method. After receiving the `InteractionEvent` for `Interaction.Status.IDLE`, use the `markDone()` method to properly save and clean the interaction.

The e-mail server can have a strategy to send the transcript of the chat interaction to the contact. In this case, to disconnect, use `clearCall()` or `transferToQueue()` instead of `releaseCall()`. The transcript is automatically sent to the contact.
Chat event flow is shown in Processing a Chat Interaction.



Processing a Chat Interaction

Open Media Interactions

Open media interactions are multimedia interactions, that is, the `InteractionOpenMedia` interface inherits `InteractionMultimedia` interface. Other multimedia interactions are discussed in E-Mail Interactions and Chat Interactions.
This chapter presents `SimpleOpenMediaInteraction`, a new example that receives open media interaction.

Open Media Design

Contact center agents routinely work with applications that are separate from traditional CRM (Customer Relationship Management) or agent desktop applications. Genesys Open Media allows you to create custom media types that integrate this agent activity into the contact center workflow.
To use open media, you will need to define new interaction types in the Configuration Layer. After you do this, you can use the Media Interaction SDK to build client and server applications that manage them within the Genesys platform. Finally, you will use the Agent Interaction SDK to allow agents to process these interactions.
This section will provide an overview of the kinds of situations that lead application developers to use open media interactions. It will then outline some of the things you can do with open media using the Agent Interaction (Java API). Finally, it will give more in-depth descriptions of a couple of real-world open media use cases.

> **Important**
>
> For more information on the Media Interaction SDK, see *Media Interaction SDK 7.6 Java Developer's Guide*. For more information on multimedia interactions, see *Multimedia 7.6 User's Guide* and read the *Basic Interaction Models* chapter from the *Genesys Events and Models Reference Manual*.

Bridging the Contact Center and the Enterprise

There are many occasions when contact center agents could carry out work they do not normally do. Open media makes this a lot more productive than it would otherwise be.
For example, in the wake of a natural disaster, insurance companies tend to get high call volumes as people file their claims. At this point, you might want to have claims adjusters and others available to expand the contact center workforce.

But after most of the calls come in, the claims adjusters will have a lot of work to do on these new claims. With open media, you can route calls to adjusters when call volumes are high, and then you can route routine work from the adjusters to the contact center when call volumes are low.

Another common scenario is for contact center agents to handle faxed requests when they are idle. In this case, you could set up an interaction type that includes the fax data for the agent to process.

When the interaction is routed to the agent's desktop, the agent can process the data as appropriate and mark the item as done.

There are several ways you could use open media in these situations. In the simplest cases, you might want to use open media interactions merely to route work to an agent. But in some cases, it might be even better to process information directly in the agent desktop, using the open media functionality of the Agent Interaction SDK. There will be an example of each of these scenarios later in this section, but first, here is a look at some of the things you can do with open media.

Basic Capabilities

Agent Interaction SDK makes it possible for you to perform actions like the following on open media interactions:

- Open an interaction (using an existing interaction as the parent interaction).
- Create a new interaction.
- Respond to a received interaction.
- Thread interactions.
- Forward an interaction.
- Transfer an interaction to another queue or place.
- Add an interaction to an existing history.
- Retrieve any open, pending, or terminated interaction—of any open media type—from a contact history database.
- Manage an interaction's workflow.

> **Important**
>
> For more information on the open media–related functionality provided by the Agent Interaction SDK, see the Agent Interaction SDK API Reference's entry on the InteractionOpenMedia class.

Routing Rejected Orders to an Agent

This example will show how you might use open media interactions simply as a routing mechanism, while having an agent continue to work in an application that is separate from his or her desktop. Agents for a telephone company might use many applications, including an order management application. When a customer calls in to order DSL (a form of high-speed Internet service), an agent must enter the customer information and submit the order. If the agent makes a mistake on the order form, the system will reject the order and send it back to the agent for correction.

Because the order management system is not linked to the contact routing platform, the incorrect order will sit at the agent's desktop—perhaps for hours—until there is a lull in inbound activity that

will allow the agent to address the problem.

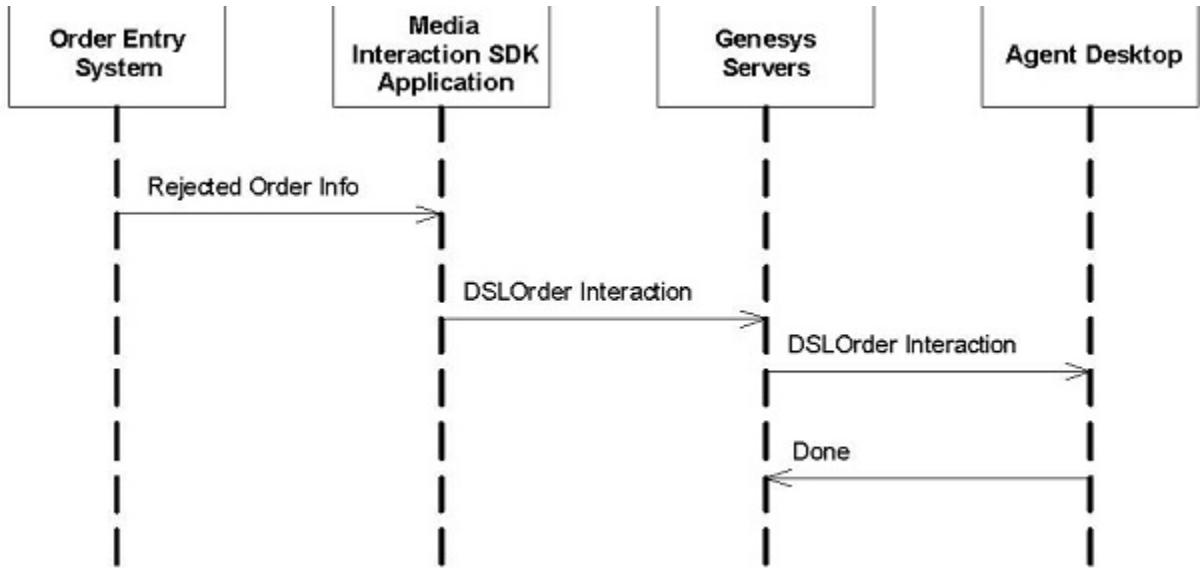
With the addition of Genesys Open Media, the scenario is different.

If the agent makes a mistake on an order, the rejected order can be submitted to the Genesys platform as a new activity to be queued and sent back to the agent. Depending on the priority that you set, the agent may be taken off the phone queue immediately and routed back to the order to make corrections.

Here is how you could use Genesys Open Media to create this kind of solution:

1. Define a new interaction type, DSLOrder, in the Configuration layer.
2. Set up the appropriate routing for the new interaction type.
3. Use the Media Interaction SDK to write an application that can receive information about rejected orders from the order management system.
4. Use the Agent Interaction SDK to write agent desktop functionality that allows the agent to mark the `DSLOrder` interaction as done. (Note that in this scenario, the agent is doing the order management work in the order management application. All that is required from the Agent Interaction SDK is the ability to mark the interaction as done.)

When an order is rejected, the order entry system will send information about the order to the new Media Interaction SDK application, which will create a new interaction of type `DSLOrder`. The interaction will be sent to the Genesys servers for processing and they will route the interaction to the appropriate agent, as shown below. When the agent has corrected the order, he or she will mark the interaction as done.



Alerting an Agent to Process a Rejected DSL Order

See SimpleOpenMediaInteraction for a code example that shows how to write an agent desktop application that can mark an open media interaction as done.

Working on a CRM Case

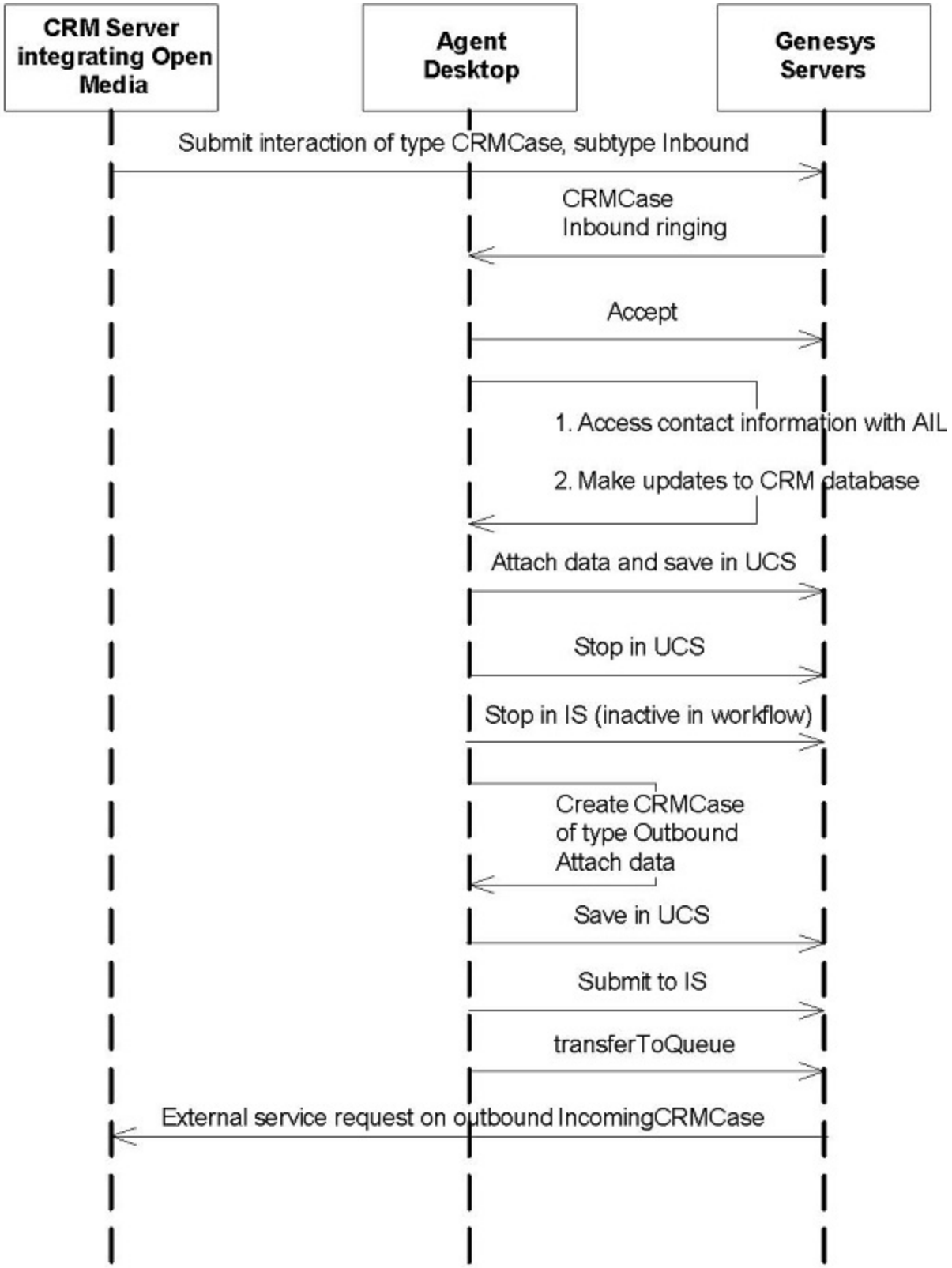
Sometimes it is not enough to use open media as a routing and notification mechanism. There are many cases where it makes more sense to write detailed interaction-handling functionality directly into the agent desktop.

For example, you can use open media interactions to allow contact center agents to handle incoming fax data for use by a Customer Relationship Management system. This could be done in a way that is very similar to the preceding example, but in this case, you might prefer that the interactions you create would also carry data for the agent to process right in his or her agent desktop application.

As in the example above, you would need to define a new interaction type, perhaps called `CRMCase`. You would also need to set up the appropriate routing and write an application that uses the Media Interaction SDK. This application would attach CRM data to the `CRMCase` interaction.

But the Agent Interaction SDK functionality you write would be a bit different, as it would do more than just mark the interaction `Done`. Here is the process the interaction would follow, as shown in Handling a CRM Case Using Open Media:

- The CRM server submits a `CRMCase` interaction of type `Inbound` to the Genesys servers.
- The interaction rings at the agent's desktop.
- The agent accepts the interaction.
- The agent accesses contact information from the Universal Contact server, using the Agent Interaction SDK.
- The agent desktop updates the CRM database.
- The agent desktop saves the interaction in the UCS database, using the Agent Interaction SDK.
- The agent desktop stops the inbound interaction in UCS.
- It also stops the interaction in Interaction Server.
- The agent desktop creates a reply to the inbound interaction. This reply is a `CRMCase` interaction of subtype `Outbound`.
- The agent desktop attaches data to the outbound `CRMCase` interaction, using data from the inbound transaction, as appropriate.
- The agent desktop submits the outbound interaction (to make it active in the Interaction Server workflow).
- The interaction is transferred to a queue.


CRM Server integrating Open Media
Agent Desktop
Genesys Servers
Submit interaction of type CRMCase, subtype Inbound
CRMCase Inbound ringing
Accept
1. Access contact information with AIL
2. Make updates to CRM database
Attach data and save in UCS
Stop in UCS
Stop in IS (inactive in workflow)
Create CRMCase of type Outbound Attach data
Save in UCS
Submit to IS
transferToQueue
External service request on outbound IncomingCRMCase

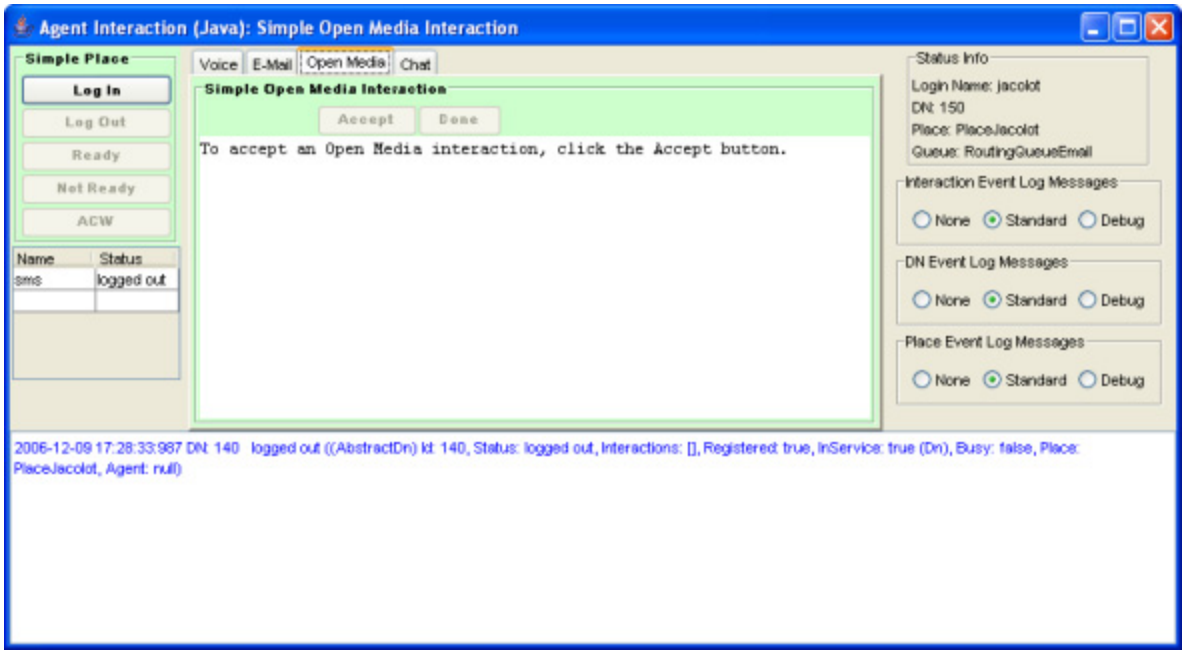
Handling a CRM Case Using Open Media

SimpleOpenMediaInteraction

This example extends `SimplePlace` and is very similar to `SimpleEmailInteraction`, which was explained earlier in this chapter.

> **Important**
>
> This example processes open media interactions. In order to run it, you will need to create your own open media interactions using the Media Interaction SDK. You must also set up a business process and routing strategy to get these interactions to your agent. The example is packaged with the business process and routing strategies that were used to develop it. If you use this business process and strategies, you will have to modify server names and Configuration Layer objects. For more information, refer to the Universal Routing documentation.

When you launch the application, there is a panel with two buttons: `Accept` and `Done`, as shown below. These buttons will let the agent accept an open media interaction and mark it done, respectively. When the agent accepts the interaction, information about it will be displayed in the GUI.



SimpleOpenMediaInteraction at Launch Time

Here are the steps you must take to create this application. As before, the steps that have already been done by `SimplePlace` will not be discussed here.

Set up Action Buttons

The `linkWidgetsToGui()` method is very similar to corresponding methods discussed earlier in this chapter. The two buttons are also very simple. Here is the action code for the `Accept` button, which is just like the `Accept` button for `SimpleEmailInteraction`:

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

Likewise, the `Done` button simply marks it done:

```
sampleInteraction.markDone();
```

Note that the `markDone()` method tells both Interaction Server and Universal Contact Server to stop processing the interaction.

These two sections of code are almost all that is new in this section of the example. Since the `setInteractionWidgetState()` code—used to synchronize the user interface—is very similar to what you have seen in previous examples, the only other thing to do is set up the event-handling code, which is also fairly simple.

Add Event-Handling Code

This example uses event-handling code that is largely similar to what you saw in `SimpleVoiceInteraction`.

> **Important**
>
> As explained in Threading section, you should write short and simple event handlers to avoid delaying the propagation of events.

As in `SimpleVoiceInteraction`, you check for a status of `TALKING` and for the correct media type. Then you read in the information and display it in the GUI:

```
//Checking that the event reports a change
//on an open media interaction
if(event.getSource() instanceof InteractionOpenMedia)
{
        //...
        // if the event involves the interaction to process,
        // it tests the interaction status
        if (sampleInteraction != null
                        && event.getInteraction().getId()==sampleInteraction.getId())
        {
                // When the open media interaction is in talking status,
                // it means that the agent or place is owner of the interaction
                // this is when you can process this interaction.
                if (event.getStatus() == Interaction.Status.TALKING) {

                        /// You can process the interaction.
                        /// In this example, processing the interaction corresponds
                        /// to displaying the text of the open media interaction
                        sampleInteraction=(InteractionOpenMedia) event.getSource();

                        // You can now access the Open Media interaction's content.
                        String someText = "Media type: "
                                + sampleInteraction.getOpenMediaType()
                                + "\nInteraction type: "
                                + sampleInteraction.getOpenInteractionType()
                                + "\nSubject: " + sampleInteraction.getSubject()
                                + "\nText: " + sampleInteraction.getText()
                                + "\nDate: "
+sampleInteraction.getDateCreated().toGMTString();

                        openMediaTextArea.setText(someText);
                }
        }
}
```

Contact

Contact Information

A contact is a customer with whom the agent may interact through a media type.
Each contact has an ID, which is a unique system reference used in the Genesys Framework. The Universal Contact Server (UCS) stores the contact data, which includes names, e-mail addresses, phone numbers, and other information. This server also stores the history of a contact, that is, processed interactions.
`ContactManager` is an interface that lets you:

- Get and set contact information.
- Search for contacts.
- Add new contacts and new contact information.

To get the `ContactManager`, call the `AilFactory.getContactManager()` method as shown in the following code snippet:

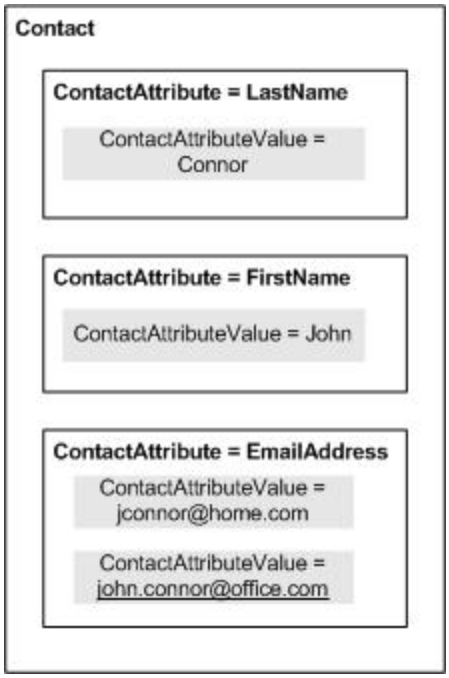
```
ContactManager myContactManager= myAilFactory.getContactManager();
```

Getting Contact Information

With the `ContactManager` , you can retrieve `Contact` objects. The `Contact` interface describes the data of a contact. To get a `Contact` instance describing a particular contact, you need the corresponding contact identifier. This identifier is available, for example, in interactions. Then, you can use the `ContactManager.getContact()` method to retrieve the contact. The following code snippet shows how to retrieve the `Contact` interface of an interaction:

```
public void handle(InteractionEvent event){
        //....
        Interaction myInteraction = event.getSource();
        String myContactID = myInteraction.getContactId();
        Contact myContact = myContactManager.getContact(myContactId();
        //....
}
```

The `Contact` interface offers access to contact attributes. A contact attribute is contact data that can have several values. For example, if a contact has one or several e-mail addresses, the e-mail address is an attribute and each e-mail address is contained in a `ContactAttributeValue` object as illustrated in Example of Contact Information.



Example of Contact Information

Default Attributes

First name, last name, phone number, e-mail address, and title are default attributes available for each contact. Those default attributes have `get` and `set` methods in the `Contact` interface for accessing directly their values. The following code snippet displays the last and first names of a contact:

```
System.out.println( myContact.getFirstName() +" "+myContact.getLastName());
```

Attributes and Metadata

The Universal Contact Server defines metadata for each type of attribute. (Attributes are defined in the Configuration Layer under `Business Attributes`.) For example, the last name is a type of contact attribute specified by metadata. For the last name attribute, the metadata specifies that the attribute name is `LastName`, the type of the attribute value is a string, the display name is `Last name`, and so on.

A single metadata is available for each type of attribute; it has a unique system identifier and a unique name. For example, a single metadata is available for all the existing last names' attribute values. The metadata is independent from the contact attributes' values.

The metadata interface is `ContactAttributeMetaData`. Call the `ContactManager.getContactAttributeMetaDataById()` or `ContactManager.getContactAttributeMetaDataByName()` to retrieve a

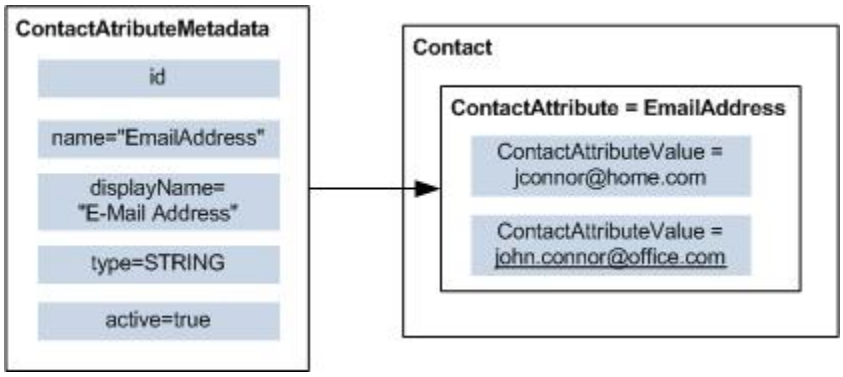
`ContactAttributeMetadata` interface. The following code snippet retrieves the metadata for e-mail addresses:

```
ContactAttributeMetaData myHomeAddressMetadata =
myContactManager.getContactAttributeMetaDataByName("HomeAddress");
```

For default attribute metadata, you can use a `ContactManager.get*Attribute()` that returns the attribute corresponding to*, as in the following example:

```
ContactAttributeMetaData myEmailAddressMetadata = myContactManager.getEmailAddressAttribute();
System.out.println(myEmailAddressMetadata.getDisplayName() );
```

For a particular contact, each `ContactAttributeMetaData` object is associated with a set of `ContactAttributeValue` objects that are available through the `Contact` interface. Each `ContactAttributeValue` object has a unique system identifier and contains a value of the contact attribute.



Attributes' Metadata and Values

> **Important**
>
> In the previous figure, the primary attribute value is underlined for each type of attribute.

Use the `Contact.getAttributeValues()` method to retrieve the contact attribute values of a contact.

```
// Getting all the contact e-mails
Collection myEmailAddresses = myContact.getAttributeValues(myEmailAddressMetadata, false);

// Displaying the string value for each ContactAttributeValue
Iterator itEMails = myEmailAddresses.iterator();
```

```
while(itEMails.hasNext())
{
   ContactAttributeValue _email = (ContactAttributeValue) itEMails.next();
   System.out.println(_email.getStrValue() );
}
```

Primary Attributes

A contact's primary attribute value is one of the attribute values marked as primary. For example, if a contact has several e-mail addresses, the e-mail address at work can be the primary e-mail attribute. For default attributes, the `Contact` interface provides you with `get/setPrimary*()` methods. For instance, the following code snippet displays the contact's primary e-mail address.

```
System.out.println(myContact.getPrimaryEmailAddress() );
```

The `ContactAttributeValue` interface includes an `isPrimary()` method that returns `true` if the value is a primary one.

> **Important**
>
> You cannot have two primary values for a given attribute. Your application must manage itself primary values.

Fast Contact Management

By calling the `ContactManager.findOrCreateContact()` method, you can specify key-value pairs for searching contacts, where the key is the string name of a `ContactAttribute` and the value is a string value. The method returns the IDs of matching contacts, or, if no such contact exists in the database, it creates a contact for these new parameters.

Through this method, your application can also take advantage of the Contact Server Custom Lookup algorithm (which accelerates the contact search) by adding a `MediaType` key with the `voice` or `email` value.

The following code snippet activates this algorithm and makes a fast search that creates a contact if the e-mail address is unknown.

```
// Creating the map of attribute values of the contact:
// tsmith@myCompany.com
Collection myFastContactSearch = new HashMap();

// Adding the last name attribute
myFastContactSearch.put("EmailAddress", "tsmith@myCompany.com");

// Adding the key-value pair that activates the CSCL algorithm
myFastContactSearch.put("MediaType", "email");

// Fast search
Collection result = ContactManager.findOrCreateContact(
   myFastContactSearch
```

```
);
```

Advanced Search Feature

The AIL library includes an advanced-search feature for contacts. With the `ContactManager` interface, your application can search contacts according to several attribute values and their associated metadata ID. The matching results can be sorted or truncated by sizing the returned array of results. This subsection details the steps to build an advanced search.

Creating a Contact Search Template

To search contacts, you first create a SearchContactTemplate object. Call the `ContactManager.createSearchContactTemplate()` method to get an empty instance:

```
SearchContactTemplate mySearchTemplate = myContactManager.createSearchContactTemplate();
```

Building a Filter Tree

Contact filter trees correspond to search requests that approximate SQL requests to the *Universal Contact Server* (UCS). Those filter trees are equivalent to assignment and logical expressions. For example,`(LastName="B*" and FirstName="A*")` searches for any contact whose first name begins with A and whose last name begins with B, and `(EMailAddress="ab*@company.com")` searches for any contact whose e-mail address begins with ab and finishes with `@company.com` .
Your application must build a `FilterTreeElement` object to fill the contact template. Without a filter tree, searching for a contact is not possible. A `FilterTreeElement` can be either a `FilterNode` or a `FilterLeaf` instance.
The following subsections detail the creation of filter leaves and nodes, then list the best practices for filling these elements.

Filter Leaves

A filter leaf contains a terminal expression that defines a search value for a contact attribute, such as `LastName=”B*”` or `primary Lastname=”B*”`. This means that your application can use the wildcards defined in the `FilterLeaf.LeafWildcard` enumerated type.
As the Agent Interaction Java API is able to find any occurrence, even if it includes the * character, your application creates a normalized string, as shown in the following code snippet:

```
//Creating the string B*
String leaf1Value = mySearchTemplate.normalizeSearchValue("B") ;
leaf1Value.concat(FilterLeaf.LeafWildcard.ANY.toString() );
```

For additional details about wildcards, see the Agent Interaction SDK 7.6 Java API Reference. Your application can create a filter leaf with an instance of the `FilterLeaf` class, which associates a metadata ID with a contact attribute value.
The following code snippet implements a `FilterLeaf` object for the `primary LastName=”B*”` expression:

```
// Getting the metadata
```

```
ContactAttributeMetaData myLastNameMetadata = myContactManager.getLastNameAttribute();

// Creating the leaf
FilterLeaf myLeaf = mySearchTemplate.createFilterLeaf();

// Setting the expression: "primary LastName=B*"
myLeaf.setPrimaryOnly(true)
myLeaf.setContactAttribute(myLastNameMetadata);
myLeaf.setOperator(FilterLeaf.LeafOperator.EQUAL);
myLeaf.setValue(leaf1Value);
```

To restrict the search to the primary value of the attribute, set the `primaryOnly` flag to `true` by calling the `FilterLeaf.setPrimaryOnly()` method. If your application calls the SearchContactTemplate.`setSearchPrimaryValueOnly(true)` method (see Filling the Search Template), the search does not take into account the `primaryOnly` flag of the `FilterLeaf` object.

> **Important**
>
> Before you set attributes for filter leaves, see details in Filling the Search Template.

Filter Nodes

A filter node contains a non-terminal expression that defines an operation for several non-terminal (nodes) or terminal (leaves) expressions. For example:

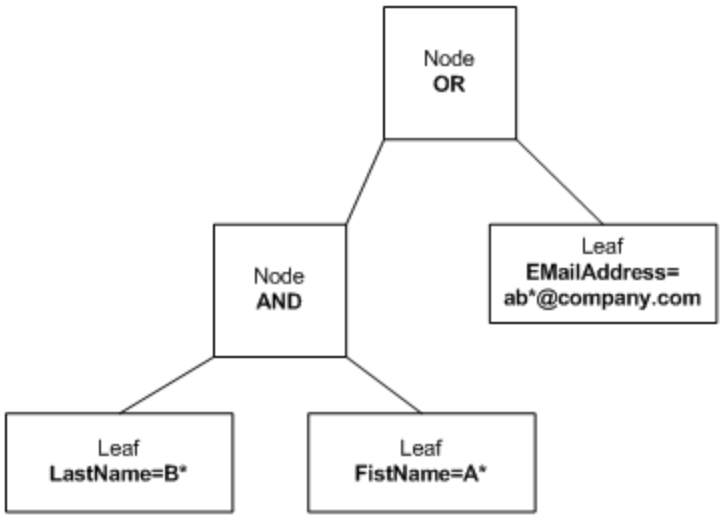
```
a or b or c

a and b and c

a or b

a and b
```

—where a, b, and c can be other filter nodes or leaves, and operators are `or` and and.

Your application can create a filter node with an instance of the `FilterNode` class, which associates an operator with a set of contact filter nodes or leaves.

The following figure presents a filter node containing the following expression:

```
(LastName="B*" and FirstName="A*") or (EMailAddress="ab*@company.com")
```



A Contact Filter Node

The following code snippet creates a `FilterNode` for the object for the `LastName="B*" AND FirstName="A*"` expression:

```
// Creating the leaf for FirstName="A*"
// Getting the metadata
ContactAttributeMetaData myFirstNameMetadata = myContactManager.getFirstNameAttribute();

// Creating the leaf
FilterLeaf myLeaf2 = mySearchTemplate.createFilterLeaf();

// Setting the expression: "FirstName=A*"
myLeaf2.setContactAttribute(myFirstNameMetadata);
myLeaf2.setOperator(FilterLeaf.LeafOperator.EQUAL);

//Creating the string A*
String leaf2Value = mySearchTemplate.normalizeSearchValue("A") ;
leaf2Value.concat(FilterLeaf.LeafWildcard.ANY.toString() );

myLeaf2.setValue(leaf2Value);
myLeaf2.setPrimaryOnly( true );

// Creating the Node
FilterNode myNode = mySearchTemplate.createFilterNode();

// Setting the operator AND
myNode.setOperator(FilterNode.NodeOperator.AND);

// Adding the leaf for "LastName=B*" (see above)
myNode.addFilterLeaf(myLeaf);

// Adding the second leaf
myNode.addFilterLeaf(myLeaf2);
```

Best Practices for Filling Your Filter Tree

The filter tree's definition determines the processing times of your search requests. There are several aspects to take into account to fit your application needs and fine-tune the building of your filter tree.

is-searchable

Genesys recommends that your application uses attributes marked as `is-searchable`. This ensures that you make calls to the appropriate UCS search algorithms and receive the most timely responses. To set up `is-searchable` attributes, open the targeted `Attribute Value` object in the `Contact Attributes` list of your Configuration Manager. In the `Annex` tab of the attribute object, open `settings` and set to `true` the `is-searchable` option.

In the Agent Interaction Java API, call the `isSearchable()` method of the `ContactAttributeMetadata` instance to determine whether the associated attribute values are searchable.

If your application searches for any attributes, regardless of whether they are marked as searchable, the process will be time consuming, and will slow down your application. In particular, if your application is a server, this method is inappropriate to ensure the most timely performance performances for your application and the system.

primary

If you set the `primaryOnly` flag to `true` by calling the `FilterLeaf.setPrimaryOnly()` method, you restrict the search to the attributes' primary values. The more you search for primary values, the less the SQL request is complex, and thus, UCS requires less time to return a result.

Also, for an even quicker search, you can set up the `SearchPrimaryValueOnly` flag to `true` by calling the SearchContactTemplate.`setSearchPrimaryValueOnly(true)` method. The search is restricted to attributes' primary values only, regardless of the definition of `FilterLeaf` objects in the filter tree.

Number of Attributes

The number of attributes used in the filter tree to refine your search impacts the request's processing time. The more attributes you set up, the finer your search is, but the longer the request takes.

Additionally, if you set up a wide search with few attributes or vague values, that returns a large collection of instances, and this increases the processing time as well, because it impacts the network activity. The problem is similar if you set up a great number of attributes to be returned with each matching instance: The more instances that match, the more data is returned, and this slows your response.

Filling the Search Template

A SearchContactTemplate object defines a contact search and the array of returned matching contacts. Once you have created the filter tree, fill the template by setting:

- The list of attributes to retrieve for each matching contact by passing their metadata.
- The list of attributes to use for sorting the matching contacts.
- The `SearchPrimaryValueOnly` flag.
- The filter tree.
- The number of returned results.
- The index of the first item in the matching results.

As explained in the Best Practices for Filling Your Filter Tree section, the filter tree definition impacts the time processing for receiving results, but the impact does not stop there. The list of attributes to use for sorting the matching contacts is important too. According to the number of matching results, if you set up a great number of attributes to be returned with each matching instance, you can face some network issues: the more instances that match, the more data is returned, and this slows your response.
The following code snippet fills the search template with the filter tree created in the previous section. It searches only for primary values.

```
// Results are sorted by names
mySearchTemplate.addSortAttribute(myLastNameMetadata, false);

// Contacts are returned with their last names, // their first names, and all their e-mails
mySearchTemplate.addRetrieveAttribute(myLastNameMetadata, false);
mySearchTemplate.addRetrieveAttribute(myFirstNameMetadata, false);
mySearchTemplate.addRetrieveAttribute(myEmailAddressMetadata, false);

// The results have to match the filter tree
mySearchTemplate.setFilter(myNode);

// Activating quick search
mySearchTemplate.
setSearchPrimaryValueOnly
(true);

// The first 10 contacts are returned
mySearchTemplate.setIndex(0);
mySearchTemplate.setLength(10);
```

Then, call the `ContactManager.` searchContact() method to request the contact search. It returns the matching contacts in a `Collection` . The following code snippet uses the above search template and displays the primary attributes of the matching contacts.

```
// Requesting a search for this template
Collection myMatchingContacts = myContactManager.searchContact(mySearchTemplate);
Iterator itContacts = myMatchingContacts.iterator();
while(itContacts.hasNext())
{
  Contact _Contact = (Contact) itContacts.next();
  System.out.println(_Contact.getFirstName()
   +" "+ _Contact.getLastName()
   +" "+ _Contact.getPrimaryEmailAddress());

}
```

Advanced Contact Management

With the ContactManager interface you can create contacts, and for each contact, the Contact interface allows you to add (or set) new values to attributes, remove some attribute values, merge information from another contact, or even remove the contact from the database.

Creating Contacts

You can create a contact with the ContactManager.createContact() method. If you only have default attributes to specify, you can create the contact in a simple call, as shown in the following code snippet:

```
Contact theCreatedContact = myContactManager.createContact("M.", "Terry", "Smith",
"tsmith@myCompany.com", "4153087723");
```

To create the contact with more attributes, you have to create ContactAttributeValue objects with the ContactAttributeMetaData.createValue() method for each attribute value of the created contact. The following code snippet shows how to proceed for the new contact Terry Smith .

```
// Creating the map of attribute values of the contact:
// Terry Smith, tsmith@myCompany.com
HashMap myNewContactAttributes = new HashMap();

// Creating the last name attribute
ContactAttributeValue myLastNameValue= myLastNameMetadata.createValue("Smith");
myLastNameValue.setPrimary(true);
myNewContactAttributes.put(myLastNameMetadata, myLastNameValue);

// Creating the first name attribute
ContactAttributeValue myFirstNameValue= myFirstNameMetadata.createValue("Terry");
myFirstNameValue.setPrimary(true);
myNewContactAttributes.put(myFirstNameMetadata, myFirstNameValue);

//...
//... Creating
// Creating the contact:
Contact theCreatedContact = myContactManager.createContact(myNewContactAttributes);
```

Adding Attribute Values

For any contact, you can add new values to an attribute (defined in the Configuration Layer) of the contact with the Contact.setAttributeValues() method. Create ContactAttributeValue objects with the ContactAttributeMetaData.createValue() method for each new attribute value. After a call to Contact.setAttributeValues() , propagate the contact changes in the database by calling the Contact.save() method, as shown below.

```
//Creating a collection of e-mail values
ArrayList myOtherEMails = new ArrayList();
ContactAttributeValue email1 =
myEmailAddressMetadata.createValue("terry.smith33@webmail.com");
myOtherEMails.add(email1);
ContactAttributeValue email2 = myEmailAddressMetadata.createValue("tsmith33@webmail.com");
myOtherEMails.add(email2);
```

```
// Setting the new values
theCreatedContact.setAttributeValues(myEmailAddressMetadata, myOtherEMails);
// Updating the database with changes
theCreatedContact.save();
```

If a ContactAttributeValue instance has a non-null identifier, the setAttributeValues() method updates the value corresponding to the identifier.

Contact History

The *Universal Contact Server* (UCS) stores contacts' data, including the contact history. The history manager gives access to a set of contact histories managed by the UCS. For each contact, its history contains a set of interactions involving the contact. Within the history manager, your application can retrieve summaries for a set of interactions.
To get an interface for the history manager, call the AilFactory.getHistoryManager() method as shown in the following code snippet:

```
HistoryManager myHistoryManager= myAilFactory.getHistoryManager();
```

For each contact, the HistoryManager can retrieve an History object. This History object contains a list of HistoryItem s that are interaction summaries.
First, you create and fill a SearchInteractionTemplate object. The history manager uses this template to:

- Set the size of the HistoryItem list.
- Set the index of the first item retrieved with this list.
- Set the interaction attributes to retrieve.
- Set the interaction attributes to sort interactions.

The HistoryManager interface includes three default interaction attributes that are only used for HistoryItem sorting: the interaction subject, the owner identifier corresponding to an agent ID (or username), and the start date of the interaction. Those attribute values are always available in the get methods of a HistoryItem .
For each default interaction attribute, you can get the corresponding InteractionAttributeMetaData interface with three dedicated methods of the history manager:

```
HistoryManager.getInteractionAttributeMetaDataForSubject()
HistoryManager.getInteractionAttributeMetaDataForOwnerId()
HistoryManager.getInteractionAttributeMetaDataForStartDate()
```

You can also get other InteractionAttributeMetaData with the InteractionManager.getInteractionAttributeMetaDataById() or InteractionManager.getInteractionAttributeMetaDataByName() methods.
Those metadata correspond to the interaction attributes defined in the Configuration Layer.

> **Important**
>
> If you add those metadata in the list of retrieved attributes, they are available in the attached data of the item.

The following code snippet retrieves the first ten history items of a contact history, sorted by subject:

```
// Creating the search template
SearchInteractionTemplate myTemplate = myHistoryManager.createSearchInteractionTemplate();
// Setting list characteristics
// The ten first history items are retrieved
myTemplate.setLength(10);
myTemplate.setIndex(0);
// Getting the subject metadata for sorting interactions
InteractionAttributeMetaData mySubjectMetaData =
myHistoryManager.getInteractionAttributeMetaDataForSubject();

// The history items are sorted by subject
myTemplate.addSortAttribute(mySubjectMetaData,false);

// Getting the history containing the archived interactions
// and fulfilling the template
History myHistory = myHistoryManager.getHistory(myContact, myTemplate, true);

//Displaying the History Content
List myHistoryItems = myHistory.getHistoryItems();
Iterator itItems = myHistoryItems.iterator();
while(itItems.hasNext())
{
   HistoryItem myItem = (HistoryItem) itItems.next();
   System.out.println(myItem.getSubject());
   System.out.println(myItem.getDateCreated());
}
```

Standard Responses

This chapter covers the standard responses, handled through the Standard Response Manager.

SRL Design

Agent Interacting (Java API) includes access to a self-learning categorization system to help agents by providing responses when they process a multimedia interaction.
The Standard Response Library system is self-learning: it “teaches” itself with new incoming messages, according to agents’ feedback. For further information about the SRL, refer to *Universal Contact Server* (UCS) documentation.
The following subsections detail how the SRL service interacts with the Standard Response Library database.

Standard and Suggested Responses

A standard response is a pre-written response stored in the Standard Response Library database. An agent may choose to reply to a customer with a response from the Standard Response Library. The provided standard response may have tags in its body that your application can automatically replace with contacts’ data.

When an agent processes an interaction, your application can display a tree of standard responses or the ones contained in the interaction’s suggested categories (if any).

The system selects this suggested categories according to categorization criteria. For details, see Category section below.

Your application can insert standard responses as replies into any e-mail or chat message, or it can display them so that an agent can read them to the contact during a voice interaction.

Category

A category is a group of related standard responses and categories that are available for similar interactions. For example, a company might define a `Defect` category, that contains standard responses to provide when customers report a product defect. In this `Defect` category, this company might define a category for each product. Each category defines a set of more-specific responses for the product’s identified defects.

Your application can display categories as trees, allowing the agent to select a category and a standard response with that category. Your application can also propose an interaction’s suggested category. For instance, if an e-mail interaction has suggested categories, they are available by calling the `InteractionMail.getSuggestedCategories()` method.

An agent can accept or reject the system’s choice of a selected category, in order to provide feedback to the Standard Response Library’s self-learning system.

SRL Manager

The SRL manager provides your application with access to categories and standard responses stored in the Standard Response Library database. This chapter's remaining sections cover the SRL Manager's main features:

- Accessing standard responses and categories.
- Managing agents' favorites.

Using the SRL Manager

SRL Management is provided with the `SRLManager` interface and classes of the `com.genesyslab.ail.srl` package.

The `SRLManager` interface accesses standard responses used when an agent processes an interaction. Your application can use the information retrieved by the SRL manager to display categories, and standard responses, in trees.

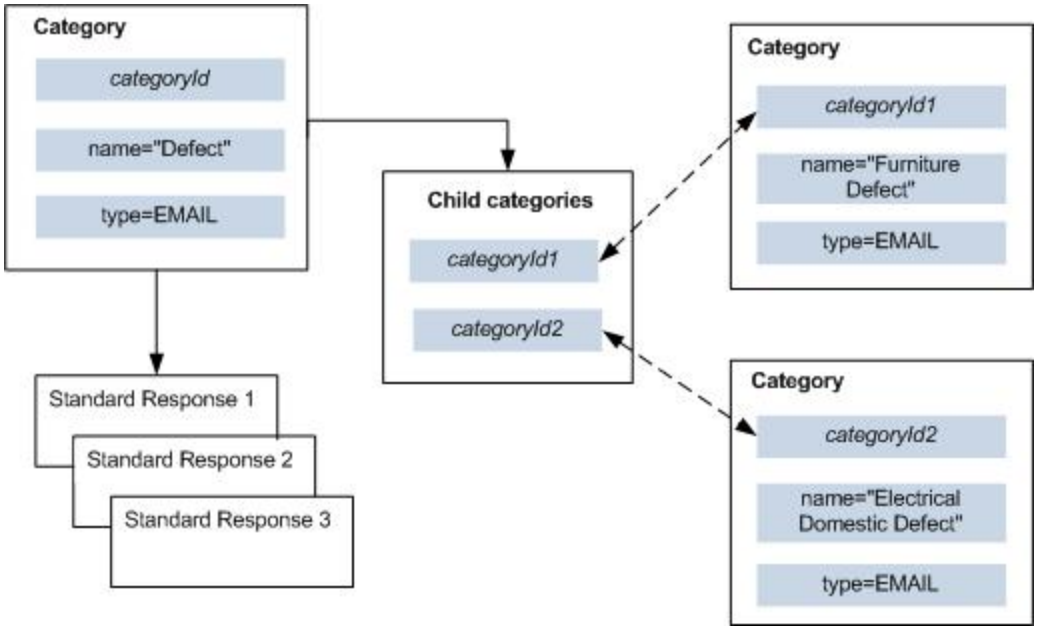
To get the SRL manager, call the `AilFactory.getSRLManager()` method as shown in the following code snippet:

```
SRLManager mySRLManager= myAilFactory.getSRLManager();
```

Getting Categories and Standard Responses

As Category in the SRL Database shows, in the SRL database, a category is composed of:

- Its own data, such as the category name, system ID, language and type.
- A set of standard responses that belong to the category.
- An array of its child category IDs.



Category in the SRL Database

Use the SRLManager interface to get a Category interface for a given system identifier. In the following code snippet, the SRLManager interface gets the Category instance corresponding to the category identifier for a multimedia interaction.

```
void handleInterationEvent( InteractionEvent event ) {

        //....
        InteractionMultimedia mail =
                (InteractionMultimedia) event.getSource();

        // ...
        // Getting the current category assigned // to the Multimedia Interaction
        String categoryID = mail.getCategoryId();

        // Getting the category instance with the SRLManager
        Category myCategory = mySRLManager.getCategory(categoryID);

        //Displaying the category content
        System.out.println(" Category Name: "+ myCategory.getName() +
               " Description: "+myCategory.getDescription());
        // ...
}
```

Each Category instance allows to access the standard responses and child categories assigned to the category, as shown here.

```
// Displaying the responses in this category
Collection myStandardResponses = myCategory.getStandardResponses();
Iterator itResponses = myStandardResponses.iterator();
while(itResponses.hasNext())
```

```
{
        StandardResponse myResponse = (StandardResponse) itResponses.next();

        System.out.println("Response Name: "+ myResponse.getName() + " Description:
"+myResponse.getDescription()+"\n");
        System.out.println("Body: "+ myResponse.getBody()+"\n");
}

// Displaying the child categories
Collection myChildCategories = myCategory.getChildrenCategories() ;
Iterator itChildren = myChildCategories.iterator();
while(itChildren.hasNext())
{
        Category myChild = (Category) itChildren.next();
        System.out.println("\tChild Category Name: "+ myChild.getName() + " Description:
"+myChild.getDescription());
}
```

Each category is a category tree and can belong to another category. Each child `Category` instance contains its own standard responses and child categories.

A category that does not belong to another category is a root category of a category tree. Call `SRLManager.getCategoriesRoot()` to retrieve the root categories as a collection of categories.
A standard response's body text can include tags that you can replace with interaction and contact's data by calling the `StandardResponse.getBody()` method as shown here:

```
System.out.println("Filled body text: " + myResponse.getBody(agent0, mail) );
```

Managing Agent's Favorites

The `SRLManager` interface manages agents' favorite standard responses. The `SRLManager` can:

- Add a new favorite standard response for an agent by calling `addStandardResponseFavorite()` .
- Get the favorite standard responses for an agent by calling `getStandardResponseFavoriteIds()` .
- Remove a favorite standard response for an agent by calling `removeStandardResponseFavorite()` .

Handling Suggested Categories

The `InteractionMultimedia` interface defines all the methods handling suggested categories. The self-learning system can suggest categories for multimedia interactions, for example, incoming e-mail interactions.
When receiving an e-mail interaction, your application can check if the self-learning system suggested other categories, as shown here.

```
Map suggestedCategories = mail.getSuggestedCategories();
```

The returned `Map` contains category identifiers and associated relevancy.

Your application must give feedback to the self-learning system.
If the agent uses a standard response of the interaction's suggested categories, approve this category, as shown here.

```
// Assigning one of the suggested categories to the interaction
mail.setCategoryId("myCategoryId");
// Approving the category
mail.setIsCategoryApproved(Boolean.TRUE);
```

If the agent disapproves all suggested categories for the interaction, he or she must select the suggested category with best relevancy and then disapprove it, as shown in the following snippet:

```
// Assigning the most relevant category to the interaction
mail.setCategoryId("myCategoryId");
// disapproving the category
mail.setIsCategoryApproved(Boolean.FALSE);
```

> **Important**
> This disapproves all the suggested categories for this interaction.

If the agent selects a standard response of a non-suggested category, he or she must add this category with a `null` relevancy to the suggested categories and then, approve it, as shown here.

```
HashMap mySelectedCategory = new HashMap();
mySelectedCategory.put("mySelectedCategory", null);

// Adding the satisfactory category
mail.addSuggestedCategories(mySelectedCategory);

// Assigning the satisfactory category to the interaction
mail.setCategoryId("myCategoryId");

// Approving the satisfactory category
mail.setIsCategoryApproved(Boolean.TRUE);
```

Outbound Service

Handling outbound information no longer requires handling a particular interaction type. Implementing outbound is now a matter of handling additional outbound information that the Agent Interaction (Java API) provides to interactions.
This chapter shows you how to deal with the outbound service.

Outbound Design

In the 7.6 release, you no longer deal with the `InteractionVoiceOutbound` interface. Instead, you manage interactions as usual, and you get additional outbound information for an interaction to be processed. These changes make it possible to handle both voice and multimedia interactions in outbound campaigns.

Outbound Information

To access outbound information, you deal with the `OutboundService` instance of the current `Place` in use. With `OutboundService` methods, you can register listeners for getting events about current campaigns, and access to `OutboundChain` objects for this `Place` .
Each `OutboundChain` instance contains customer outbound data, provided as a collection of `OutboundRecord` objects. For example, in the context of a voice outbound call, each record of the chain contains a phone number associated with the customer to be called. If the call with a given record fails, the agent can get a chained record to attempt a new call for this customer.
Regardless the campaign mode, the `Place` object receives `PlaceEventOutboundChainInfo` events for new or modified outbound chains that the agent should process.
To determine which outbound chain is associated with an interaction, call the `OutboundService.getOutboundChain(Interaction)` method.

Outbound Actions

`OutboundRecord` and `OutboundChain` interfaces provide you with outbound methods, that the agent calls to perform outbound actions, such as `cancel`, `do not call` , and `reschedule` .
These actions are independent from the outbound interaction used to process the outbound record or the outbound chain. They manage record information on the Outbound Server.

Campaign Dialing Modes

The campaign dialing modes determine how an agent, or a group of agents, participates in an outbound campaign. This affects the outbound event flow, and also agent actions to be performed when participating in the campaign.
The following table introduces these dialing modes.

Campaign Dialing Modes

Campaign Mode	AIL Events	Description
PREVIEW	`PlaceEventOutboundChainInfo`	In preview dialing mode, an agent requests one or several records from the OCS, previews the record(s), and decides to process one of them by creating a new outbound interaction.
PUSH_PREVIEW	`PlaceEventOutboundChainInfo` `InteractionEvent` (for `NEW` `InteractionOpenMedia`)	(Also called proactive) In push preview mode, the agent does not need to request the record to preview the record. He or she gets this and subsequent records through a new open media interaction. To process the record, the agent creates a new outbound interaction.
PROGRESSIVE	`PlaceEventOutboundChainInfo` `InteractionEvent` (for `DIALING` `InteractionVoice`)	The OCS dials a record in the list as soon as an agent is available.
ENGAGED_ PROGRESSIVE	`PlaceEventOutboundChainInfo` `InteractionEvent` (for `DIALING` `InteractionVoice`)	The OCS creates a voice interaction to dial a record in the list when an agent is available and engaged. Your agent application gets a dialing outbound voice interaction and an outbound chain.
PREDICTIVE	`PlaceEventOutboundChainInfo` `InteractionEvent` (for `DIALING` `InteractionVoice`)	The OCS predicts agent availability and dials a record. Your agent application gets a dialing outbound voice interaction and an outbound chain.
ENGAGED_ PREDICTIVE	`PlaceEventOutboundChainInfo` `InteractionEvent` (for `DIALING` `InteractionVoice`)	The OCS predicts when the agent is available and engaged, and creates a dialing outbound voice interaction to dial a record in the list. Your agent application gets a dialing outbound voice interaction and an outbound chain.
UNKNOWN	Unknown	Unknown campaign mode.

Important

According to the campaign mode, you may notice the following:

- In a non-engaged mode, the contact may be online before the agent.
- In an engaged mode, your application can receive a record after the agent accepted a ringing call. Test the `InteractionVoice.isASMCall()` to check whether a ringing call is part of an outbound campaign.

For further information, refer to the *Outbound Contact 7.6 Documentation*.

Steps for Writing an Outbound Application

Now that you have been introduced to the outbound feature's design, it is time to outline the steps you will need to work with its events and objects.
As specified in the previous section, outbound data do not interfere with interaction management. To handle campaign information and outbound records, modifications in your agent application consist in a few adds-in.
There are five basic things you will need to do in your AIL application:

- **Implement a CampaignListener listener** to get notified of changes in active outbound campaigns. Here is how a `SimpleExample` class would do this:

```
public class SimpleExample implements CampaignListener {
        //...
        public void handleCampaignEvent(CampaignEvent event)
        {

                // Testing whether it is a new active campaign
                if(event.getEventType()==CampaignEvent.Type.CAMPAIGN_ADDED)
                {
                        //The agent takes part in a new outbound campaign
                        //...
                }
        }
}
```

- **Get an outbound service** to register your campaign listener. For instance, you could modify one standalone code example by declaring a private `outboundService` variable, then by adding the following code snippet in the constructor method:

```
Class SimpleExample implements CampaignListener
{
        OutboundService outboundService;
        public SimpleExample()
        {
                //Retrieve the service
                outboundService = samplePlace.getOutboundService();
                //Add your campaign listener
                outboundService.addListener(this) ;
                //...
        }
```

```
}
```

- **Set up button actions** (or actions on other GUI components) tied to outbound features, according to the `OutboundService`, `OutboundChain`, and `OutboundRecord` objects' methods. For instance, to cancel a record, your agent needs an `Cancel` button to cancel the processing of the record.

```
// Add a cancel button for joining the chat session
JButton cancelOutboundRecordButton = new JButton(“Cancel”);
cancelOutboundRecordButton.setAction(new AbstractAction("Cancel") {
        public void actionPerformed(ActionEvent actionEvent) {
                try {
                        outboundRecord.cancel(null);
                } catch (Exception exception) {
System.out.println(exception.getMessage(),"ErrorEvent");
                }
        }
});
```

- **Modify the `PlaceListener.handlePlaceEvent()` method** to handle `PlaceEventOutboundChainInfo` events. Create a thread that manages `PlaceEventOutboundChainInfo` events to update your application with outbound information.

```
// For instance, display the event reason</tt>
System.out.println("Outbound chain event- reason is:
"+PlaceEventOutboundChainInfo.getReason().toString());
```

- **Check if interactions own outbound information** in the implemented `handleInteractionEvent()` methods. If a campaign is started, according to the campaign mode, your application may have to check if interactions in `NEW`, `DIALING`, and `RINGING` status are associated with outbound information.

See further sections for details.

Preview Outbound Interactions

In a Preview Outbound Campaign, the agent is active and requests new outbound records, then chooses either to process or not process the call.

Active Campaigns

As soon as the agent is logged in, his or her place receives events for active campaigns. To determine whether your agent participates in a Preview Outbound Campaign, you can implement the `CampaignListener` as described in the previous section. On `CampaignEvent` events of type CAMPAIGN_ADDED, retrieve the corresponding CampaignInfo object and test its type, as follows:

```
if(event.getEventType()==CampaignEvent.Type.CAMPAIGN_ADDED)
{
        OutboundCampaignInfo campInfo = event.getCampaignInfo();
        if(campInfo.getCampaignMode() == OutboundCampaignInfo.Mode.PREVIEW)
        {
                //...
        }
}
```

Another way to determine whether a preview outbound campaign is active is to retrieve outbound campaigns available for your place, by calling the `OutboundService.getCampaignInfos()` method, or, if you know the campaign ID, by calling the `OutboundService.getCampaignInfo()` method.

Start and Stop Preview

Your application should start and stop preview mode according to the optional setting `agent_preview_mode_start` defined in the OCS application object in Configuration Manager. If this option is set to `true`, your agent application must start preview mode method after an agent logs in, prior to any preview record request, in order to receive scheduled call records from OCS.

This setting is most often used to ensure that no prescheduled call records are sent to the place directly after the agent logs in.

> **Important**
>
> The Agent Interaction (Java API) does not include a method to test this option. This is your responsibility to determine whether you develop an agent application working with an OCS whose `agent_preview_mode_start` option is `true`.

To start preview mode, retrieve the `OutboundCampaignInfo` instance that corresponds to your campaign. Then, call the `startPreviewMode()` method.

```
OutboundCampaignInfo campaign = OutboundService.getCampaignInfo(campaignID);

campaign.startPreviewMode();
```

To stop preview mode (when the agent stops working in outbound campaigns), call the `stopPreviewMode()` method.

```
campaign.stopPreviewMode();
```

If the agent wants to participate in a preview campaign, preview mode must be started before requesting any preview record, else OCS ignores calls to get preview records.
When the option `agent_preview_mode_start` is set to `false` , OCS assumes that the agent is ready to receive any prescheduled call records. If a preview campaign is running when the agent logs in, he or she can request preview records at anytime.

Handle Regular Preview Calls

Your agent is now ready to participate in an active campaign. If preview mode mode is active (see above for details), the first step is to request a preview record.

> **Important**
>
> If your campaign mode is push preview, your application does not need to request the preview record.

To get a preview record, you can choose between calling the`getPreviewRecord()` method (which returns the record) or calling the `requestPreviewRecord()` method of your `OutboundCampaignInfo` interface.
If your application calls the `requestPreviewRecord()` method, and if a record is available, you get a `PlaceEventOutboundChainInfo` event through the `PlaceListener.handlePlaceEvent()` method.

Use the `PlaceEventOutboundChainInfo` event to inform the user that a new outbound record should be processed, as shown here:

```
public void SimplePreviewExample implements PlaceListener {
        OutboundService outboundService ;

        public void SimplePreviewExample()
        {
                //...
                boolean request = campInfo.requestPreviewRecord();
                if(request == true)
                        System.out.println("Request for preview record succeeded.");
        }

        public void handlePlaceEvent(PlaceEvent event)
        {
                if(event instanceof PlaceEventOutboundChainInfo)
                {
                        PlaceEventOutboundChainInfo eventInfo = (PlaceEventOutboundChainInfo)
event ;
                        OutboundChain outboundChain = eventInfo.getOutboundChain();
                        OutboundRecord outboundRecord = outboundChain
                        .getActiveRecord();
                        System.out.println("Outbound chain event, reason "+
eventInfo.getReason().toString());
                }
        }
```

To process the `OutboundChain`, create an interaction which uses the active record to fill in the `Interaction` data and methods' parameters. In the following code snippet, the interaction processes a voice call and uses record data to dial the call.

```
        // Method called when the agent wish to use the record
        public void processActiveRecord(OutboundChain outboundChainToProcess)
        {
                InteractionVoice outboundIxn =
                        (InteractionVoice) outboundChain.createInteraction
(MediaType.VOICE,null,agentInteractionData.getQueue());
                outboundIxn.makeCall(outboundRecord.getPhone(), null,
InteractionVoice.MakeCallType.REGULAR, null, null, null) ;
        }
}
```

> **Important**
> When your outbound interaction is released, close the outbound chain, as specified in section Close the Chain.

Handle Push Preview Interactions

In push preview mode, your application deals with open media interactions: it does not make preview requests, it receives the open media interaction and the chain which contains the record.
Attached data provided in the open media interaction contains additional preview information that your application uses to create the outbound interaction (for instance, voice or e-mail).
Push preview mode requires that the agent logs into a third party media (refer to the Configuration Layer for further details). Then, the setup is the same: you get an `OutboundService` from your `Agent`, and you register your listeners.

Push Preview Events

When the agent is logged into and ready on the third party media, you get the following events:

- a `PlaceEventOutboundChainInfo` event through the `PlaceListener.handlePlaceEvent()` method.
- an `InteractionEvent` event through the `PlaceListener.handleInteractionEvent()` method for an `InteractionOpenMedia` in `NEW` status, as shown in the following code snippet.

```
public void handleInteractionEvent(InteractionEvent event)
{
        if(event.getSource() instanceof InteractionOpenMedia
                        && event.getStatus() == Interaction.Status.NEW))
                        {
                                InteractionOpenMedia outboundPreviewIxn =
(InteractionOpenMedia) event.getSource();
                                OutboundChain outboundChain =
outboundService.getOutboundChain(ixn);
                                OutboundRecord outboundRecord =
outboundChain.getActiveRecord();
                                // Use the outbound chain to create the outbound interaction
                                // and use the InteractionOpenMedia to get additional data
                                //...
                        }
}
```

Process the Outbound Interaction

To process the `OutboundChain` , create an interaction by calling the OutboundChain.createInteraction() method (for instance, voice or e-mail) which uses the active record and the open media interaction to fill in the `Interaction` data and methods' parameters. In the following code snippet, the interaction processes a voice call and uses record data to dial the call.

```
// Method called when the agent wish to use the record
public void processActiveRecord(OutboundChain outboundChain, InteractionOpenMedia
```

```
outboundPreviewIxn)
{
        InteractionVoice outboundIxn =
                (InteractionVoice) outboundChain.createInteraction
(MediaType.VOICE,null,(String) outboundPreviewIxn.getAttachedData("queue");
                ixnVoice.makeCall(recordToProcess.getPhone(), null,
InteractionVoice.MakeCallType.REGULAR, null, null, null) ;
}
}
}
```

To end the outbound interaction, the agent releases it then marks it done. Finally, before your application closes the outbound chain (as explained in section Close the Chain), your application should also mark done the preview open media interaction.

```
outboundIxn.releaseCall();
outboundIxn.markDone();

outboundChain.markProcessed();
outboundPreviewIxn.markDone();
outboundChain.close();
```

> **Important**
>
> The agent can be in charge of performing the mark done of the open media interaction.

Predictive Outbound Interactions

Handling a predictive outbound is simpler than handling preview outbound interactions. The setup is the same: you get an `OutboundService` from your `Agent`, and you register yours listeners, but you do not have to request records. Outbound Server is in charge of distributing records and dialing interactions. Refer to the *Outbound Documentation* for further details.

For a predictive outbound campaign, your application just waits for `RINGING` interactions and outbound chains.

Active Campaigns

To determine whether your agent participates in a Predictive Outbound Campaign, test whether the campaign mode is OutboundCampaignInfo.Mode.PREDICTIVE. To do so, you can implement the `CampaignListener` as described in section Active Campaigns.

Handling a Predictive Outbound Interaction

If your agent participates in a predictive campaign, your application gets interactions in `DIALING` or `TALKING` status, to be processed as usual. For further details, see previous interaction chapters. When your application gets those interactions through interaction events, display outbound data. Each received outbound interaction is associated with an outbound chain, that contains the record

used to fill in interaction data. You get this record by calling the `OutboundChain.getActiveRecord()` method, as shown in the following code snippet:

```
public void handleInteractionEvent(InteractionEvent event)
{
        if(event.getStatus() == Interaction.Status.DIALING
        )
        {
                Interaction outboundIxn = event.getInteraction();
                OutboundChain outboundChain = outboundService.getOutboundChain(ixn);
                OutboundRecord outboundRecord = outboundChain.getActiveRecord();
                //...
        }
}
```

When the agent has processed the outbound record and released the outbound interaction, he or she must specify the processing result by calling the `OutboundRecord.setCallResult()` method. The `OutboundRecord.CallResult` enumeration lists the possible result your application can provide the agent with.

```
// Successfully processed the interaction
outboundRecord.setCallResult(OutboundRecord.CallResult.ANSWER);
```

If the interaction is processed, you mark the corresponding outbound chain as processed, else for instance, you can reschedule the record.
In any case, after you mark done the interaction, call the `OutboundChain.close()` method to terminate properly the outbound processing of the chain.

```
outboundChain.markProcessed();
outboundIxn.markDone();
outboundChain.close();
```

Handle Outbound Chains

This section covers additional information about outbound chains, regardless campaign modes.

Mark Processed

After the agent releases the outbound interaction, your application can set a call result to the active record and mark the chain processed, as shown in the following code snippet:

```
outboundRecord.setCallResult( OutboundRecord.CallResult.FromString(result));
outboundChain.markProcessed();
```

Reschedule the Record

After releasing the call, the agent can request a callback. In this case, your application needs to create a `Calendar` object to set up the callback at the required date, then reschedule the record.

```
java.util.Calendar myCalendar = java.util.Calendar.getInstance();
myCalendar.setTimeZone(java.util.TimeZone.getTimeZone("GMT-8:00"));

// Reschedule the call: 12/01/2008 at 9:00
myCalendar.set(2008, 01, 12, 9, 0);

// Anybody logged in the campaign is authorized to make the call
selectedRecord.reschedule(myCalendar, OutboundRecord.CallbackType.CAMPAIGN);
```

> **Important**
>
> The time zone of the `Calendar` instance must be set to GMT.

Next step is to close the outbound chain. See Close the Chain.
As a result of the `reschedule` command, a new chain is sent at the required date and time. To determine whether the active record of the outbound chain is a callback, call the `OutboundChain.isScheduled()` method (which returns `true` if the active record is a callback).

Close the Chain

In any case, after your application released and marked done the outbound interaction, call the `OutboundChain.close()` method to terminate properly the outbound processing of the chain, as shown in the following code snippet.

```
ixn.releaseCall(null);
outboundChain.markProcessed();
ixn.markDone();
outboundChain.close();
```

Routing Points

This chapter explains how to monitor routing points.

Routing Point Design

With the 7.*x* releases, the Agent Interaction (Java API) introduces visibility into route point and queue activity (based on the Genesys routing model).

The Universal Routing Server (URS) interprets strategies and routes interactions through routing points or routing queues before they are delivered to agents. Routing points can process interactions and provide results used by URS for further routing. For instance, a T-Server can host an Interactive Voice Response, which corresponds to a routing point and provides additional interaction data.

According to the type of routing point or queues through which the interaction goes, the treatment period varies. Agent Interaction (Java API) provides features to monitor the interaction activity on routing points:

- Monitoring a Routing Point/Queue that may be of the following types:
  - External Routing Point
  - Routing Point
  - Routing Queue
  - Virtual Routing Point
- Receiving events raised on these objects.
- Retrieving data of the `Interaction` that has been routed to the routing point.

Routing Point Information

To monitor routing points, Agent Interaction (Java API) provides access to DN routing point information and to the routing status of interactions processed on routing points.
The `DnRoutingPoint` interface registers to the T-Server and DN provides routing point information. With `DnRoutingPoint` methods, you can register `DnRoutingPointListener` listeners for getting `DnRoutingPointEvent` events, which describe status events like `in_service` or `out_of_service`.

The `RoutingInteraction` class describes an `Interaction` that arrived on a `DnRoutingPoint` instance. The`RoutingInteraction.Status` enumeration lists the possible interaction status on the routing point. For instance, if the routing interaction status is `IDLE`, it means the interaction is no longer on the Routing point, but it does not mean that the interaction is terminated (an agent may process it or another routing point may handle it.)

With `RoutingInteraction`methods, you can register `RoutingInteractionListener` listeners for getting `RoutingInteractionEvent` events, which notify changes on a routing interaction, that is, status or attached data changes.

Steps for Monitoring Routing Points

Now that you have been introduced to the routing point feature's design, it is time to outline the steps you will need to work with its events and objects.
In this section, monitoring routing points means monitoring status changes on the routing points and monitoring interactions on these routing points. the following code snippets shows how you can create
There are three basic things you will need to do:

- **Implement a DnRoutingPointListener listener** for getting status changes on routing points. This listener gets status changes for both DN routing points and routing point interactions. Here is how a `SimpleRoutingExample` class would do this:

```
public class SimpleRoutingExample implements DnRoutingPointListener {
        //...
        void handleDnRoutingPointEvent(DnRoutingPointEvent event)
        {
                DnRoutingPointEventThread p = new DnRoutingPointEventThread(event);
                p.start();
        }
        void handleInteractionEvent(RoutingInteractionEvent event)
        {
                RoutingInteractionEventThread p = new
RoutingInteractionEventThread(event);
                p.start();
        }
}
```

- **Implement the handleDnRoutingPointEvent() method** of your `DnRoutingPointListener` with a thread which updates your application's routing information. as shown here:

```
class DnRoutingPointEventThread extends Thread
{
        DnRoutingPointEvent event;

        public DnRoutingPointEventThread(DnRoutingPointEvent _event)
        {
                event=_event;
        }

        public void run()
        {
                DnRoutingPoint dnRoutingPoint = event.getDnRoutingPoint();
                System.out.println( “Event on DN RP: "+dnRoutingPoint.getId() + "
reason: "+ event.getReason().toString() + " new status: "+ event.getStatus().toString()
);
        }
}
```

- **Implement the `handleInteractionEvent()` method** (inherited from the `RoutingInteractionListener` interface) with a thread which updates your application with new interactions and interaction status changes for the monitored routing points.

```
class RoutingInteractionEventThread extends Thread
{
        RoutingInteractionEvent event;
```

```
        public RoutingInteractionEventThread( RoutingInteractionEvent _event)
        {
                event=_event;
        }

        public void run()
        {
                RoutingInteraction routingInteraction = event.getRoutingInteraction();
                //display new status is status change reported
                if(event.isStatusChanged())
                        System.out.println( "Ixn Event on ixn:
"+routingInteraction.getId() + " reason: "+ event.getReason().toString() + " new status:
"+ event.getStatus().toString() );
                //else the event reports extension changes
                else
                {
                        System.out.println( "Ixn Event on ixn:
"+routingInteraction.getId() + " reason: "+ event.getReason().toString() + " extension
changed: ");
                        // handle interaction extensions
                        //...
                }
        }
}
```

- **Register the listener for each Routing Point to be monitored** . For instance, the SimpleRoutingExample listener is added to a DN routing point in its constructor, as shown here:

```
public class SimpleRoutingExample implements DnRoutingPointListener
{
        //...
        public SimpleRoutingExample(
                        AilFactory ailFactory, String routingPointID)
        {
                try
                {
                        DnRoutingPoint dnroutingPoint = ailFactory.getRoutingPoint(
                                        routingPointID
                        );

                        dnRoutingPoint.addDnListener(this);

                }catch(RequestFailedException __e)
                {
                        System.out.println(__e.toString());
                }
        }
}
```

In your agent application, you can now create a SimpleRoutingExample instance for each routing point that you wish to monitor.

Service Status and Connection

This chapter explains how to deal with connection maintenance.

Service Status Design

Basically, a service represents the status of a connection to a server in the Genesys Framework or in Multi-Channel Routing. To handle connection changes, you implement a `ServiceListener` class, that monitors events on `ServiceStatus` objects. Then your application registers this listener for each service to be listened.
To determine which service your application will be able to listen, refer to the `ServiceStatus.Type` enumeration:

- `CONFIG` —Connection to the Configuration Layer.
- `TELEPHONY` —Connection to a T-Server.
- `IS` —Connection to an Interaction Server.
- `CHAT` —Connection to a Chat Server.
- `AIL` —for AIL itself.
- `DATABASE` —Connection to the Contact Server database.

The possible `ServiceStatus.Status` values are:

- `ON` —Server is running and connection is established.
- `OFF` —Server is down or connection is broken.
- `ABSENT` —Connection has never been established with this server.
- `LICENSE` —Connection could not be established for license reasons.
- `RESTORING` —Connection is being restored.
- `ALREADY_RUNNING` —Server is already running.
- `UNKNOWN` —Server status is unknown.
- `UNKNOWN_HOST` —Server host is unknown.
- `APPLICATION_NAME_INCORRECT` —Connection could not be established due to an incorrect application name.
- `APPLICATION_TYPE_INCORRECT` —Connection could not be established due to an incorrect application type.
- `LOGIN_INCORRECT` —Connection could not be established due to incorrect credentials.
- `CONNECTION_FAILED` —Connection could not be established.

Connection Loss

If AIL loses its connection to a server, your application gets an event related to the associate service, turning its status to OFF. See Steps for Listening to Service Status for further details about receiving these events.
However, the connection loss has additional repercussions for the objects that depend on the disconnected server.

T-Server Use Case

If AIL loses its connection to T-Server (for instance, the network link breaks), the associated TELEPHONY service turns its status to OFF. As a consequence, from your application's point of view, the DN and its voice interactions are no longer available:

- if your application implements the `handleDnEvent()`method of your `PlaceListener`, it catches a `DnEvent`which notifies the `OUT_OF_SERVICE` status (`DnEvent.EventReason.STATUS_CHANGED`).
- if your application implements the `handleInteractionEvent()` method of your `PlaceListener` , it catches an `InteractionEvent` which notifies the `IDLE` status (`InteractionEvent.EventReason.ABANDONED`).

Actually, if only the network link breaks, voice interactions still exist (the agent and the customer are still talking); the strategy determines whether interactions are re-routed or not.

> **Important**
> Do not assume that if your interaction status turns to `IDLE` (reason `ABANDONED`), or if your DN status is `OUT_OF_SERVICE`, AIL has lost its connection to T-Server. Rely on the service status to diagnose the lost connection.

Multimedia Use Case

In case your application loses its connection to the Interaction Server (for instance, the network link breaks), the associated `IS` service turns its status to OFF. As a consequence, from your application's point of view, the media are no longer available.

- if your application implements the `handlePlaceEvent()`method of your `Place` , or `Agent`instance, it catches a `PlaceEventMediaStatusChanged`which notifies the `Media.Status.OUT_OF_SERVICE` status (`Media.Reason.STATUS_REASON_CHANGED`) .
- if your application implements the `handleInteractionEvent()` method of your `PlaceListener` , it catches an `InteractionEvent` which notifies the `IDLE` status (`InteractionEvent.EventReason.ABANDONED`) for each multimedia interaction.

As with voice interactions, media interactions are `IDLE` only from the API point of view. On the Interaction Server side, the place's media are logged out. The interactions are no longer associated with the disconnected place and are pushed back in queues (according the deployed strategy).

Reconnection

When AIL loses its connection to a server, it tries to reconnect till the connection attempt succeeds. Then, your application gets an event for the associated service, turning its status to `ON` . See Steps for Listening to Service Status for further details about receiving these events. As for the disconnection, the reconnection generates several events to update object statuses.

T-Server Use Case

For example, in the case that the link to a T-Server is restored, the switch is able to provide AIL with information. So, the `Agent`, `Place`, and `Dn` instances update, and your application gets status notifications according to the implemented event handlers. Additionally, if the agent is still talking to the contact, your application will get a new `InteractionVoice` instance in `TALKING` status,`InteractionEvent.EventReason.ESTABLISHED` .

Multimedia Use Case

At the reconnection, AIL logs into the media of the place. If your application implements the `handlePlaceEvent()`method of your `PlaceListener` , it catches a `PlaceEventMediaStatusChanged`which notifies the new `Media.Status.READY` status (`Media.Reason.BACK_IN_SERVICE`) . Then, getting back interactions is a job for the strategy or the agent application.

Restart all the Connections

At runtime, your application deals with a unique instance of the `AilFactory` . If you need to restart the AIL library and all its connections to Genesys servers, first kill your instance of `AilFactory` by calling the `AilLoader.killFactory()` method, as shown in the `release()` method of the `Connector` application block.

```
mAilLoader.killFactory();
```

If this method call succeeds, you can get a new reference on the `AilFactory` singleton (see Five Rules to Build an AIL Server Application).

> **Important**
>
> Genesys recommends the use of `ailLoader.getFactory()` in your AIL client application (instead of having a reference to the singleton throughout the code). This decreases the risk of reference issues associated with `killFactory()` usage.

Steps for Listening to Service Status

Now that you have been introduced to connection maintenance, it is time to outline the steps you will need to work with its events and objects.
As specified in the previous section, you need to register a listener per `ServiceStatus` object that your application should listen to. In the following code snippets, a single listener class is implemented to listen to the whole set of `ServiceStatus` objects.
There are five basic things you will need to do in your AIL applications to monitor service connections:

- **Implement a ServiceListener class**. Here is how a SimpleService class would do this:

```
public class SimpleService implements ServiceListener {
```

- **Implement the serviceStatusChanged() method**, that notifies service status changes. In the following code snippet, the code implemented displays those statuses in real time.

```
// This method must be implemented because this class
// implements ServiceListener

public void serviceStatusChanged(
                ServiceStatus.Type service_type,
                String service_name,
                ServiceStatus.Status service_status) {

        System.out.println("Connection maintenance - " + service_type.toString()+ ": " +
service_name+ " in status " + service_status.toString());
}
```

- **Register the listener for each service to be monitored** using this listener. In the following code snippet, the constructor of `SimpleService` registers for all services.

```
//This constructor registers for all service available in AIL
public SimpleService(AilFactory ailFactory)
{
        // For each service...
        Iterator it = services.entrySet().iterator();
        while (it.hasNext()
        {
                Map.Entry entry = (Map.Entry) it.next();
                // Get the service name
                String name = (String) entry.getKey();
                ServiceStatus service = (ServiceStatus)        entry.getValue();
                ailFactory.addServiceListener(service.m_type, this) ;
        }
}
}
```

Voice Callback

Implementing voice callback is a matter of handling additional callback record information that the Agent Interaction (Java API) provides to interactions.

Callback Design

Scenario

If a customer requests a callback, the Voice Callback server records the request. Interaction SDK (Java) supports Web Callback, that is, the customer can request a callback from a web application.

At the time that the customer requested (As Soon As Possible, or some specific calendar day/time), the Voice Callback server inserts a record of the request into an appropriate queue. From the queue, the Voice Callback request is sent to the place of an available, appropriate agent. When the AIL library receives the request, it creates a `CallbackRecord` for the request, creates a new accompanying `Interaction` (which may be cast as `InteractionVoice`) on the place, and sends appropriate event objects to registered listeners.

As the phone call progresses, the library updates the `Interaction`'s status.
The completion of the phone-call attempt may have various outcomes, including successful interaction with the customer, or busy, or connection to an answering machine or fax machine, and so on.
Upon completion of the attempted call, the application lets the agent signal the AIL library that the `CallbackRecord` is processed, and the `CallbackRecord` status is updated. (When the `Interaction` is closed and marked done, the library also closes the associated `CallbackRecord`.)

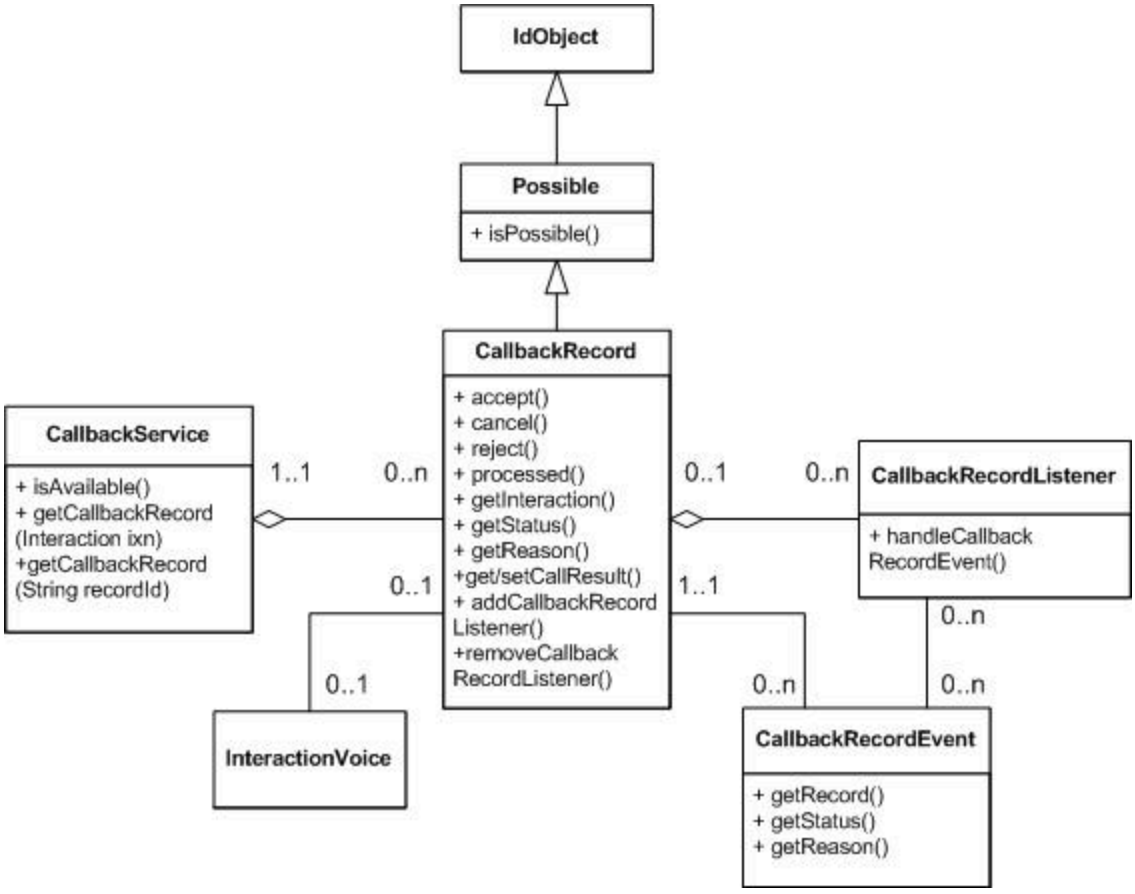
The `CallbackRecord`'s outcome status determines future actions for this Voice Callback request. For example, if the `CallbackRecord` is not successful, the Voice Callback server may re-insert the request in the queue.

If the agent rejects the task, the Voice Callback request remains at the top of the queue to be sent to some other agent's place.

Callback Information

To access callback information, you deal with the `CallbackService` instance of the current `Place` in use. With `CallbackService` methods, you can access to `CallbackRecord` objects for this `Place`.

Each `CallbackRecord` instance enables you to register listeners for getting callback events and contains record data that you use to callback the customer.
To determine which callback record is associated with an interaction, you call the `CallbackService.getCallbackRecord(Interaction)` method.



Interfaces for Callback Features

Callback Campaign Modes

To access callback features, your application gets the `CallbackService` interface associated with the place. Your application receives `InteractionEvent` events for voice interactions in different statuses, depending on the mode of the callback server.

For each voice interaction, use the `CallbackService` interface to get the associated callback record (if any). Then, use the `CallbackRecord` interface to manage the callback activity and to display information.

In preview mode, your application gets the callback record in `PREVIEW` status: the application can accept or reject the callback record by calling the corresponding `CallbackRecord` methods. Then, if the agent accepts the callback, your application makes the call using the `InteractionVoice` object associated with the `CallbackRecord` object.

In predictive mode, the application gets the `CallbackRecord` in `OPEN` status and the `InteractionVoice` object is already in a `DIALING` status. For further details about callback servers' modes, refer to the Voice Callback 7 documentation.

Your application can process the `InteractionVoice` interaction as usual. When the call is released,

assign a call result to the CallbackRecord object, mark it as processed by calling the CallbackRecord.processed() method, then mark the interaction as done.

Steps for Writing a Callback Application

Now that you have been introduced to the callback feature's design, it is time to outline the steps you will need to work with its events and objects.

As specified in the previous section, callback record data does not interfere with interaction management. You should implement a PlaceListener class that manages voice interactions, as explained in previous chapters. Then, modifications in your agent application to handle callback record data consist of a few add-ins.

There are five basic things you will need to do in your AIL applications:

- **Implement a CallbackRecordListener listener** to get notified of changes in active outbound campaigns. Here is how a SimpleExample class would do this:

```
public class SimpleCbRecordListener implements CallbackRecordListener {
        //...
        public void handleCallbackRecordEvent(CallbackRecordEvent event)
        {
                CallbackRecord record = event.getCallbackRecord();
                // update your application with callback information
                // for instance, buttons for callback actions
                //...
        }
}
```

- **Get a callback service** to test whether your PlaceListener should handle callback record on InteractionEvent events, and keep a reference to be able to retrieve callback records. For instance, you could modify one stand-alone code example by declaring a private callbackService variable, then by adding the following code snippet in the constructor method:

```
Class SimpleCallbackExample implements PlaceListener
{
        CallbackService callbackService;

        //...
        public SimpleCallbackExample(Place samplePlace)
        {
                callbackService = samplePlace.getCallbackService();
                if(callbackService.isAvailable())
                {
                        //...
                }
        }
        //...
}
```

- **Set up button actions** (or actions on other GUI components) tied to callback features, according to the CallbackRecord objects' methods, such as accept(), reject(), reschedule(), processed(), and so on.
- **Check if interactions own callback information** in the implemented handleInteractionEvent() methods. To determine whether

```
public handleInteractionEvent(InteractionEvent event)
{
        Interaction sampleInteraction = event.getInteraction();

        if(sampleInteraction.getType() == Interaction.Type CALLBACKREQUEST)
        {
                CallbackRecord callbackRecord =
callbackService.getCallbackRecord(sampleInteraction);
                if(callbackRecord != null)
                {
                        //update your application with callbackRecord info
                        //...
                        //add your listener to get callback record changes
                        callbackRecord.addCallbackRecordListener(new
SimpleCbRecordListener()) ;

                }
        }
}
```

Note that `Interaction` status changes are not automatically coordinated with `CallbackRecord` status. Coordination depends on correct agent behavior.

Expert Contact

Handling expert contact information does not require handling a particular interaction type. Your application manages voice interactions. Implementing expert contact is a matter of handling additional expert contact information that the Agent Interaction (Java API) provides to interactions. This chapter shows you how to deal with the expert contact service.

Expert Contact Design

The Agent Interaction Java API supports features for an application that enables expert users, who are not part of an enterprise's Contact Center, to provide their expertise to Contact Center agents or customers.

Usage Scenario

Contact center agents, in the course of responding to customers, sometimes need information beyond their training. In such cases, they can benefit from contacting people with special expertise. But typically, such experts (knowledge workers) are not part of the Contact center CTI infrastructure: their telephones connect to a switch that does not have a T-Server or Framework support. In such cases, their interactions with Contact center agents (or their customers) are not tracked, and neither the agents nor the experts can benefit from the information provided by Genesys Solutions.
In a site without a CTI link, the expert receives phone calls directly from a public network without the involvement of any Genesys platform components. Therefore such calls are not automatically monitored or controlled. For example, there is no way to detect the state of the expert's telephone (`Ready` , `OnCall` , and so on).
Genesys Expert Contact addresses this problem.

Expert Contact Components

There are two major components involved in making expert contact work:

- A Genesys CTI-less T-Server, which works without monitoring a switch. This component provides a connection to the expert's desktop application and to a T-Server in the contact center.
- An Expert Contact desktop application that can use AIL library features to connect to a CTI-less T-Server and monitor its events for expert contact interactions.

CTI-Less T-Server

A CTI-less T-Server provides a virtual CTI environment to track the expert's telephone states, to send messages to other Genesys server components, to handle data for current interactions, and to coordinate voice and data delivery to the expert's desktop application.
The CTI-less T-Server must communicate with a T-Server within the contact center infrastructure. It receives events from this T-Server and sends its own events to applications.

Expert Contact Application

An expert contact application provides a means for an expert to reflect his or her call state. It can also present the expert with information from the Genesys Framework.
When an expert receives a call transferred from a Genesys supported contact center, the Genesys platform components communicate with the CTI-less T-Server, which notifies the expert's desktop application of an incoming phone call. The application presents an indication to the expert, who can choose to accept or reject the phone call.
If the expert accepts the phone call, the desktop application can present available information, including the contact history if there is a connection to a Genesys Contact Server.
As the phone call progresses, the expert must use the application to view this progress. The application passes the expert's activity to the CTI-less T-Server, which in turn passes the data to the Contact Center Framework components.
The application uses AIL library features to process events from the CTI-less T-Server.

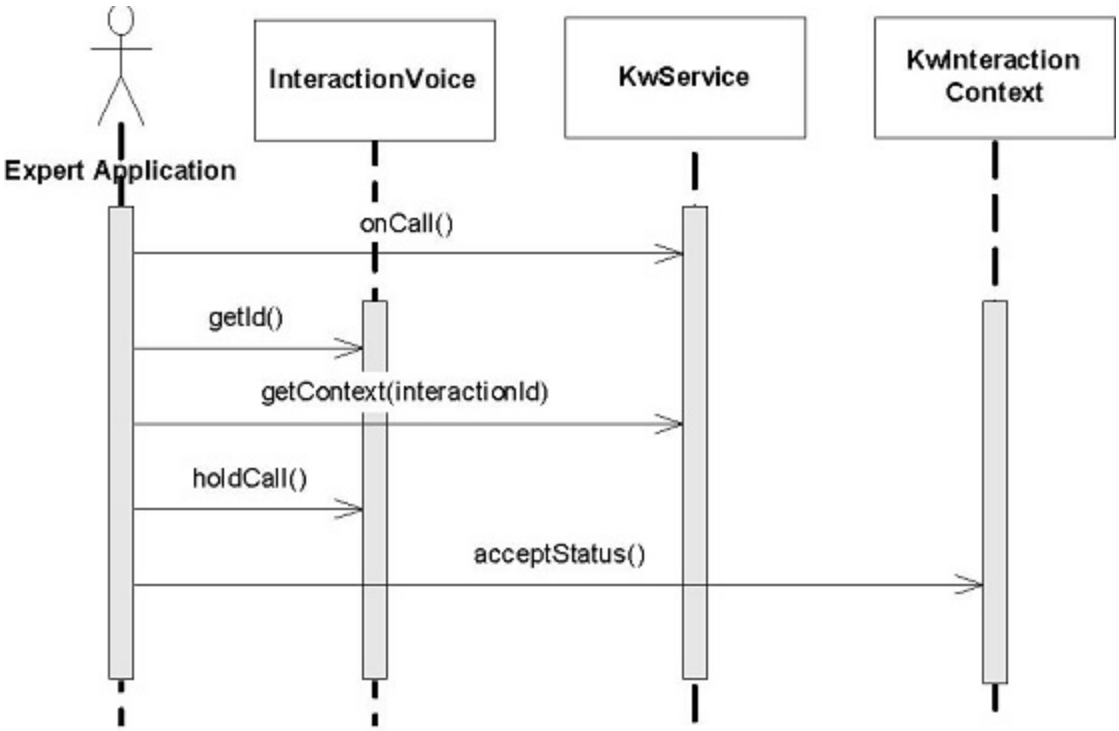
Configuration

An expert contact desktop application built on the AIL library must connect to the CTI-less T-Server. It must also connect to a Configuration Layer that has information about the CTI-less T-Server, person information for the expert, and so on.
In addition to these configuration objects, the contact center site switch and T-Server must be configured for external routing.

> **Important**
>
> See the *Genesys Expert Contact 7.6 Deployment Guide* for instructions on how to configure a Genesys expert contact application in the Configuration Layer.

Expert Contact Information

To get expert contact information, you deal with the `KwService` instance that you get by calling the `AilFactory.getKwService()` method. With `KwService` methods, you can get `KwInteractionContext` objects. Each `KwInteractionContext` object has an associated `InteractionVoice` on the CTI-less DN, and passes the expert's activity to the CTI-less T-Server, as shown below.



Using Expert Features

Your application uses `KwInteractionContext` methods to access the CTI-less T-Server, and uses `InteractionVoice` methods to emulate the expert's actions on the voice interaction.
For further information, refer to the *Genesys Expert Contact 7.x* documentation.

On Call

The expert (or knowledge worker) can receive direct calls. The expert must notify the CTI-less T-Server of such calls. To perform that notification, your application must use the `KwService.onCall()` method.
The `onCall()` method sends a message to the CTI-less T-Server to send a new interaction to the specified DN. It also creates a `KwInteractionContext` instance for the voice interaction. Your application receives and manages this voice interaction as usual.
Then, the expert uses the voice interaction to report his or her actions on the call. For instance, when the expert manually answers the call, he or she uses your application to perform an `ANSWER` action on the voice interaction.

Preview

The CTI-less T-Server can send a call to the expert. In this case, your application receives an `InteractionVoice` object associated with a `KwInteractionContext` object that has a `KwInteractionContext.Status.PREVIEW` status.
The expert can either accept or reject the call. Use the `accept()` or `reject()` method of the `KwInteractionContext` interface for this purpose.
A call to the `accept()` method sends the call to the expert's phone. Then, the expert uses the voice

interaction to report his or her actions on the call.

Status Request

If your application has set listeners, your application can receive a `KwInteractionContextEvent` event propagating the `KwInteractionContext.Status.STATUS_REQUEST` of a `KwInteractionContext` object. This periodically happens when the CTI-less T-Server requests that the expert set up his or her voice interaction's status.
The expert can choose between the `KwInteractionContext.Action.CONFIRM_STATUS` or `KwInteractionContext.Action.REJECT_STATUS` actions.
A call to the `KwInteractionContext.confirmStatus()` method indicates that the expert is still on call. A call to the `KwInteractionContext.rejectStatus()` method indicates that the expert has terminated the call and the voice interaction is automatically released.

Easy New Call and Auto Mark Done

When the expert makes a phone call, he or she must also create, and then dial, a voice interaction on his or her CTI-less DN using your application. This creates a `KwInteractionContext` instance for the voice interaction.

Your application processes the creation of this voice interaction as it would process the creation of a standard voice interaction. For further information, see Six Steps to an AIL Client Application.
Depending on your AIL configuration settings, your application can benefit from the `Easy New Call` feature. This feature changes the voice interaction's status to `TALKING` as of the interaction's creation on CTI-less DNs. The interaction's creation is therefore less time-consuming for the expert.

> **Important**
>
> To activate the `Easy New Call` feature, set the `easy-newcall` option to `true`. See the *Interaction SDK Java Deployment Guide* for further details.

When the expert hangs up a call, he or she should release the voice interaction and mark it as done.
Depending on your AIL configuration settings, your application can benefit from the `Auto Mark Done` feature. This feature automatically marks released interactions as done for CTI-less DNs.

> **Important**
>
> To activate the `Auto Mark Done` feature, set the `auto-markdone` option to `true`. See the *Interaction SDK Java Deployment Guide* for further details.

Re-Route

Depending on your AIL configuration settings, the expert is able to re-route calls. See the *Interaction SDK 7.2 Java Deployment Guide* for further details.
If you properly set `kworker` routing options, your application can use the `KwInteractionContext.reroute()` method to notify the CTI-less T-Server of the voice interaction's

routing.

Steps for Writing an Expert Contact Application

Now that you have been introduced to the expert contact feature's design, it is time to outline the steps you will need to work with its events and objects.
As specified in the previous section, expert contact data do not interfere with voice interaction management. Modifications in your voice application consist in a few adds-in to handle expert contact data and provide the user with expert contact features (for instance, by implementing a GUI panel dedicated to expert contact.)
There are five basic things you will need to do in your AIL applications:

- **Implement a KwInteractionContextListener listener** to get notified of changes in `KwInteractionContext` objects. Here is how a SimpleContextListener class would do this:

```
public class SimpleContextListener implements KwInteractionContextListener {
        //...
        public void handleKwInteractionContextEvent( KwInteractionContextEvent event)
        {
                KwInteractionContextEvent context = event.getInteraction();
                // update your application with expert context information
                // for instance, buttons for actions on the expert context
                //...
        }
```

- **Get a KwService interface** to access expert contact data. For instance, you could modify one standalone code example by declaring a private `kwService` variable, then by adding the following code snippet in the constructor method:

```
Class SimpleExpertContactExample implements PlaceListener
{
        KwService kwService;
        //...
        public SimpleExpertContactExample(AilFactory ailFactory)
        {
                kwService = ailFactory.getKwService();
                if(callbackService.isAvailable())
                {
                        //...
                }
        }
        //...
}
```

- **Set up button actions** (or actions on other GUI components) tied to expert contact features, according to the `KwInteractionContext` interface's methods.
- **Check if interactions own expert contact information** in the implemented `handleInteractionEvent()` methods. Create a thread that manages the interaction event and update your application with expert contact information.

```
Interaction interaction = event.getInteraction();

//Getting the associated expert context (if any)
KwInteractionContext kwInteractionContext = myKwService.getContext(interaction);
```

```
//Add a listener for changes in the expert context
kwInteractionContext. addKwInteractionContextListener(new SimpleContextListener()) ;

// update your application with the kwInteractionContext
//...
```

Additional Details

This chapter describes how to manage several categories of data that the AIL library provides.

Attached Data

User data, or attached data, can be any data attached to an interaction. For example, an IVR transaction may generate attached data associated with a phone call.
Attached data has the following characteristics:

- It is one or more key-value pairs.
- It is available for the whole life of an interaction—it exists in the interaction from its creation till its end.
- The API has features for managing attached data.
- Attached data can be saved in the history as part of the call, once the call is released and marked as done.

Because an attached data is a writable key-value map, it can be any data useful to your application's design. However, it can also include the following specific attached data:

- Interaction attribute values.
- Custom attached data's values.

The Configuration Layer defines keys and information for this attached data, available through the `InteractionManager` interface, as detailed in the following subsections.

InteractionManager

The `InteractionManager` interface gives access to metadata information describing interactions' attached data. The Configuration Layer defines this information in the `Business Attributes` section.

Custom Properties

The `Interaction Custom Properties` in the Configuration Layer correspond to the `CustomAttachedData` objects that your application can retrieve using the `InteractionManager.getAllCustomAttachedData()` method.
The `CustomAttachedData` class describes a single custom property. This class includes methods to get the corresponding name, display name, and description of a custom property. It also provides the predefined values for the custom attached data (if any).
Call the `CustomAttachedData.getName()` method to get the name of a custom property and use it as a key to access or modify the corresponding value in an interaction's attached data map.

Interaction Attributes

TheInteraction Values in the Configuration Layer correspond to the InteractionAttributeMetaData objects that your application can retrieve using the InteractionManager.getAllInteractionAttributeMetaData() method.

The InteractionAttributeMetaData class describes an interaction attribute. This class includes methods to get, for example, the corresponding name, display name, and description of a custom property. It also provides the predefined values for the attribute (if any).

Call the InteractionAttributeMetaData.getName() method to get the name of an interaction attribute and use it as a key to access or modify the corresponding value in an interaction's attached data map.

> **Important**
>
> These attributes can be used to retrieve interactions from a contact history. See Contact History.

Handling

The API provides you with a set of methods dedicated to attached data in the AbstractInteraction superinterface. All Interaction interfaces extend the AbstractInteraction superinterface. The following code snippet shows an example of how to create or set new values for the user data attached to the InteractionVoice object:

```
// creation of an Interaction
InteractionVoice voice = (InteractionVoice)
mAgent1.createInteraction(MediaType.VOICE, null,Queue);
voice.makeCall( DN2,null,InteractionVoice.MakeCallType.REGULAR,null,null,null);
//...
// Setting or adding new values
voice.setAttachedData("1", "one");
voice.setAttachedData("two", new Integer(2));
//Saving changes
voice.saveAttachedData();
```

If your application calls a setAttachedData(String or Object) method to modify some attached data, save the attached data by immediately calling the AbstractInteraction.saveAttachedData() method to commit all modifications on key-value pairs in the database and the T-Server.

> **Important**
>
> If your application uses the setAttachedData(Map) method passing in all the key-value pairs in the Map argument, there is no need to save attached data. The changes are committed when calling the method.

You can also create and fill a Map, then pass its reference in as a parameter of a call method. This is illustrated in the following code snippet in a makeCall():

```
HashMap userData = new HashMap();
userData.put("3", "Three");
voice.makeCall( DN2,
                null,
                InteractionVoice.MakeCallType.REGULAR,
                userData,
                null,
                null);
```

> **Important**
>
> Your program can be notified of an attached data change when an Event occurs. Use the Extension Map and the ATTACHED_DATA_CHANGED key to retrieve the data of interest. For details, see the Event-AIL Data section immediately below.

Event-AIL Data

Within InteractionEvent events, the library propagates additional AIL information called Extensions. They are different from TEvent Extensions. AIL Extensions can be retrieved through dedicated methods.

The InteractionEvent.getExtensions() method returns extended information about the event in a Map. Any keys present in this Map are defined in an InteractionEvent.Extension enumeration.

The following code snippet shows how to access an Extension in a transfer context. It implements an Agent handler, which takes into account the possibility of a transferred call ringing and manages the corresponding extension.

```
//Implementation of the Agent.HandleInteractionEvent() method
public void handleInteractionEvent(InteractionEvent _ie) {
        //Retrieval of the map containing the AIL Extensions
        Map extensions = ie.getExtensions());
        //Current status
        Interaction.Status eventStatus=interaction.getStatus();
        switch(eventStatus.toInt()) {
                //...
                // The interaction is ringing case
                Interaction.Status.RINGING_: {
                        // Retrieval of the possible transfer
                        String transferReason = (String) extensions.get(
InteractionEvent.Extension.RINGING_TRANSFER_REASON);
                        // Test if there is a transfer reason
                        if(transferReason!= null){
                                // Display of the corresponding reason
                                System.out.println("Transfer reason" +transferReason);
                        }
                }
                break;
                //...
```

```
        }
}
```

See the Javadoc API Reference for details on `InteractionEvent.Extension` keys.

Log Management

The Interaction SDK's log management is based on the `org.apache.log4j` package. The following sections first describe the default log level provided, and then describe the log system in the library.

Default AIL logs

This section discusses the default log level provided. It introduces the `log4j` package and the default log features in the AIL library.

log4j

log4j is an open-source tool designed to help write log statements to a variety of output targets. The AIL library uses the `org.apache.log4j` package to write traces to log files and to the console.

Log4J instantiation and bootstrapping are done internally by the library. You do not have to write code to perform these tasks.

The AIL library uses the main components of this package and follows Apache recommendations. The log4j version number is available in the `log4j.jar` file delivered with the Interaction SDK.

> **Warning**
> Genesys does not provide any technical support for the `org.apache.log4j` package.

AilLoader

By default, the Interaction SDK provides you with console and file traces. You can access these default logs with the `AilLoader` interface.
The `AilLoader` class enables you to:

- Disable the logs with the:
  - `AilLoader.noTrace()` method for the console.
  - `AilLoader.noLogFile()` for the log file.
- Set a debug level for the traces.
- Set your log file location.

Please refer to the AIL Javadoc API Reference for more details.

> **Warning**
> If the debug level for the traces is defined in the Configuration Layer, the library core will take this level into account upon connection to the Configuration Layer.

Adding Logs

You can add logging to your application with or without using the `log4j` package. AIL does not require you to use `log4j` for your own system trace. If you choose to use `log4j`, you can follow the recommendations in this section. For further information, refer to Jakarta documentation at:`http://jakarta.apache.org/log4j/docs/documentation.html`.
The following subsections discuss how you can use the `log4j` package to:

- Mix your own traces with the library traces.
- Generate your own traces separated from AIL logs.

Mixed Traces

You can choose to use `log4j` to add your own traces to the log, in order to mix them with AIL-generated traces. For example, you can use the Root Logger object of the `org.apache.log4j` package. The following code snippet uses the Root Logger that has already been internally instantiated by the library:

```
// Retrieving the root Logger
LoggerRepository mLoggerRepository = LogManager.getLoggerRepository();
Logger mRoot = mLoggerRepository.getRootLogger();
// Defining a layout
PatternLayout layout = new PatternLayout("%d{dd MM HH :mm:ss:SSS} [%20.20t] %-5.5p %20.20c
%m%n");
// Creating a FileAppender object to append the logs
// events occurring.
FileAppender mFile = new FileAppender(layout, "./myFile.log");
mFile.setThreshold(Level.DEBUG);
// Adding your FileAppender to the Root
mRoot.addAppender(mFile);
// Adding a message of level debug:
mRoot.debug("**** My debug message! ****");
```

Separated Traces

You can also use `log4j` to create separated logs. You just have to create your own `Logger` object, as shown in the following code snippet:

```
// Creating the Logger
Logger mLogger = Logger.getLogger("myFile.Log");
PatternLayout layout = new PatternLayout("%d{dd MM HH :mm:ss:SSS} [%20.20t] %-5.5p %20.20c
%m%n");
// Creating a FileAppender object to append the logs
// events occurring.
FileAppender mFile = new FileAppender(layout, "./myFile.log");
```

```
mFile.setThreshold(Level.DEBUG);
// Adding the FileAppender to the Logger
mLogger.addAppender(mFile);
// Adding a message of level debug:
mLogger.debug("**** My debug message! ****");
```

> **Warning**
>
> All the previous code snippets are for illustration purposes only. Code examples are not tested and not supported by Genesys.

Voice Sequence Diagrams

This appendix presents sequence diagrams for voice interactions.

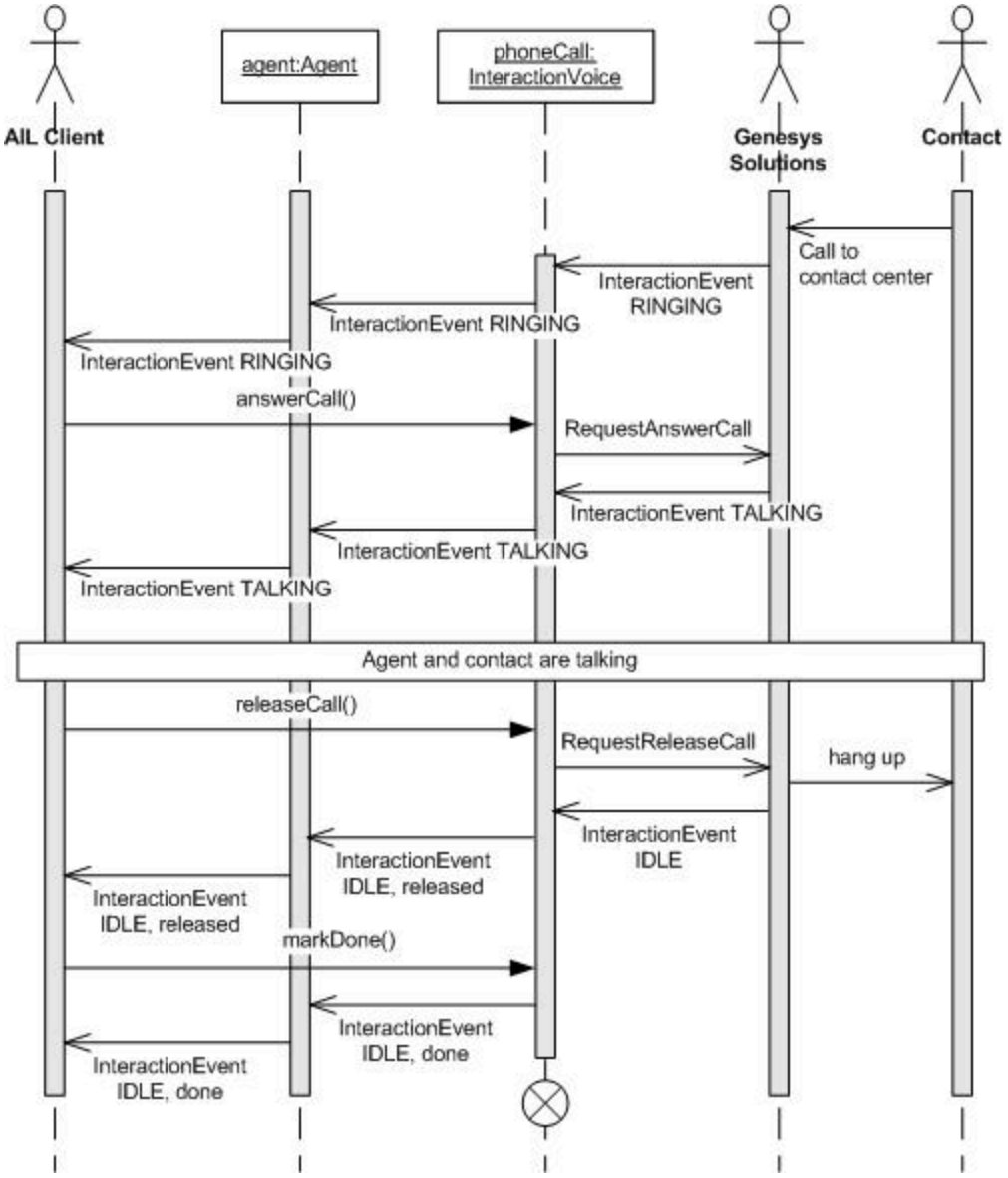
Make a Phone Call

The first step in making a phone call is to create an interaction of type `VOICE`. You do this by calling the `createInteraction()` method on your `Agent` or`Place` interface with the type of interaction. As a result, your application gets an `InteractionEvent` for a voice interaction in `NEW` status.

Make the phone call by invoking the `makeCall()` method on the `InteractionVoice` interface. At this point, the interaction status becomes `DIALING`, as specified in the corresponding `InteractionEvent`. When the connection is established, during the call, the interaction is in `TALKING` status.

Either you initiate the hang up by calling the `releaseCall()` method, or the peer has hung up. Either way, you receive an `InteractionEvent` event of type `IDLE`.

You finish and clean up the interaction by calling the `markDone()` method, which releases any reference to the interaction in the library. It also saves the interaction in the history, if you have established a connection between the library and the Contact Server database.
Event flow for making a voice call is shown in Making a Call.



Making a Call

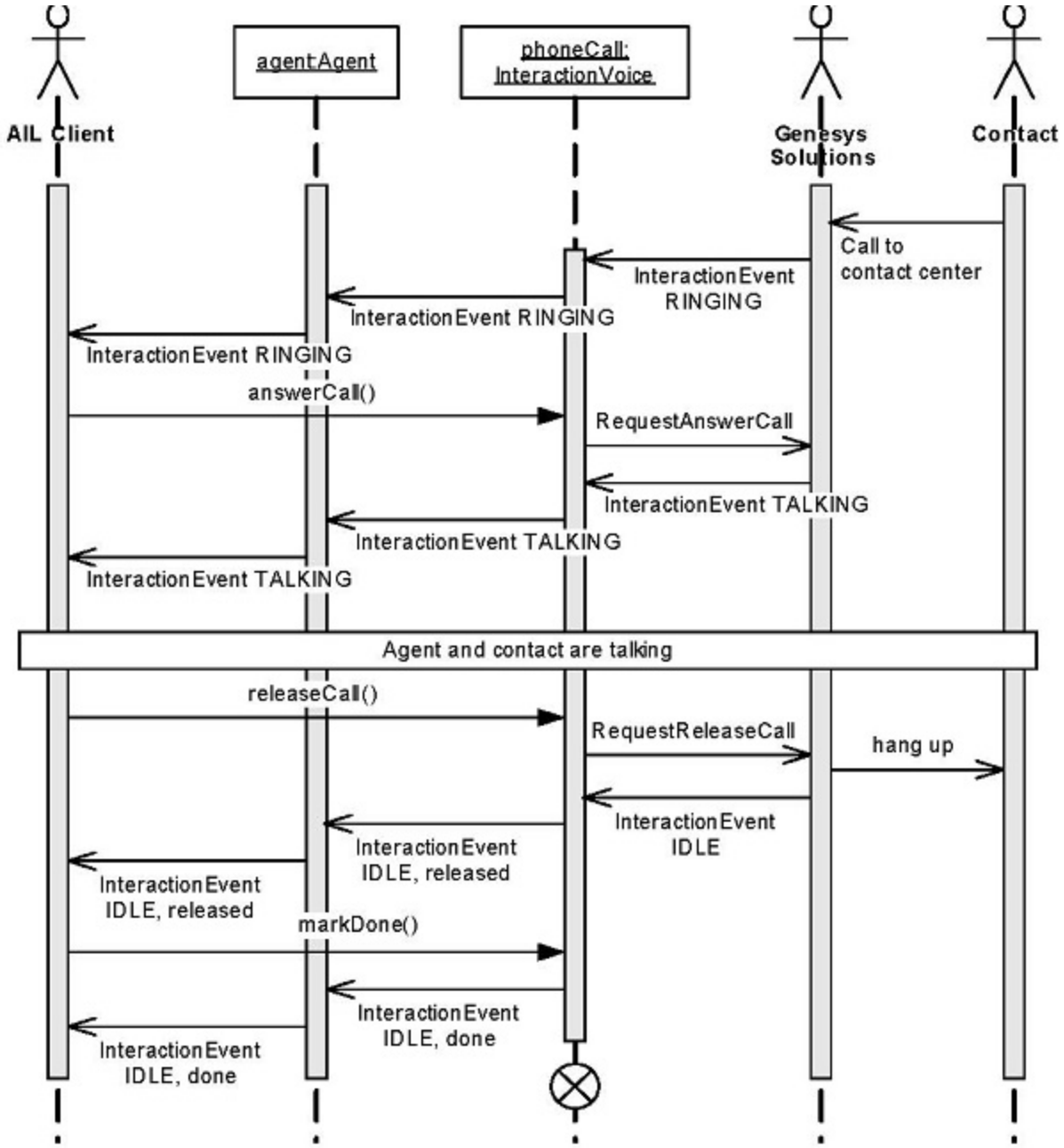
Answer a Phone Call

You have to answer a phone call when you get an `InteractionEvent` for a voice interaction in `RINGING` status. To answer the phone call, use the `InteractionVoice.answerCall()` method. As a result of the `ANSWER` action, you receive the event indicating that the interaction is in status `TALKING`.

The end of the interaction is the same as making a phone call: you hang up with `InteractionVoice.releaseCall()` method and you finish with the `InteractionVoice.markDone()`

method.

The event flow for answering a call is shown below.



Answering a Call

Conferencing

A conferencing scenario can be divided into three steps:

1. A Contact calls a first agent, named agent1.
2. agent1 initiates a conference with a second agent, named agent2.
3. agent1 creates the conference.

In the first step, agent1 receives a phone call. It is exactly the same scenario as Answer a Phone Call:

- agent1 receives an InteractionEvent event carrying an InteractionVoice interface, named phoneCall1, with the status RINGING.
- agent1 takes the call by invoking the answerCall() method.

In the second step, agent1, already in communication with the contact, invokes the initiateConference() method on the InteractionVoice interface phoneCall1 to prepare the conference with agent2.

After this call, agent1 receives an InteractionEvent event setting the InteractionVoice phoneCall1 to the status HELD. The communication with the contact is paused.

The Agent Interaction Layer creates then two InteractionVoice core objects:

- One for agent1, setting an interaction with agent2. This interaction follows the scenario in Make a Phone Call, and is named phoneCall2.
- The other for agent2, representing his communication first with agent1 and then with the conference. This interaction follows the scenario in Answer a Phone Call and is named phoneCall3.

Next, agent1 receives an InteractionEvent event carrying the newly created InteractionVoice phoneCall2 with the status DIALING. Simultaneously, agent2 receives an InteractionEvent event carrying its InteractionVoice phoneCall3 with the status RINGING. When agent2 answers the call by invoking the answerCall() method, both agents receive an InteractionEvent event showing that their InteractionVoice core objects have the status TALKING. They are now in communication with each other.

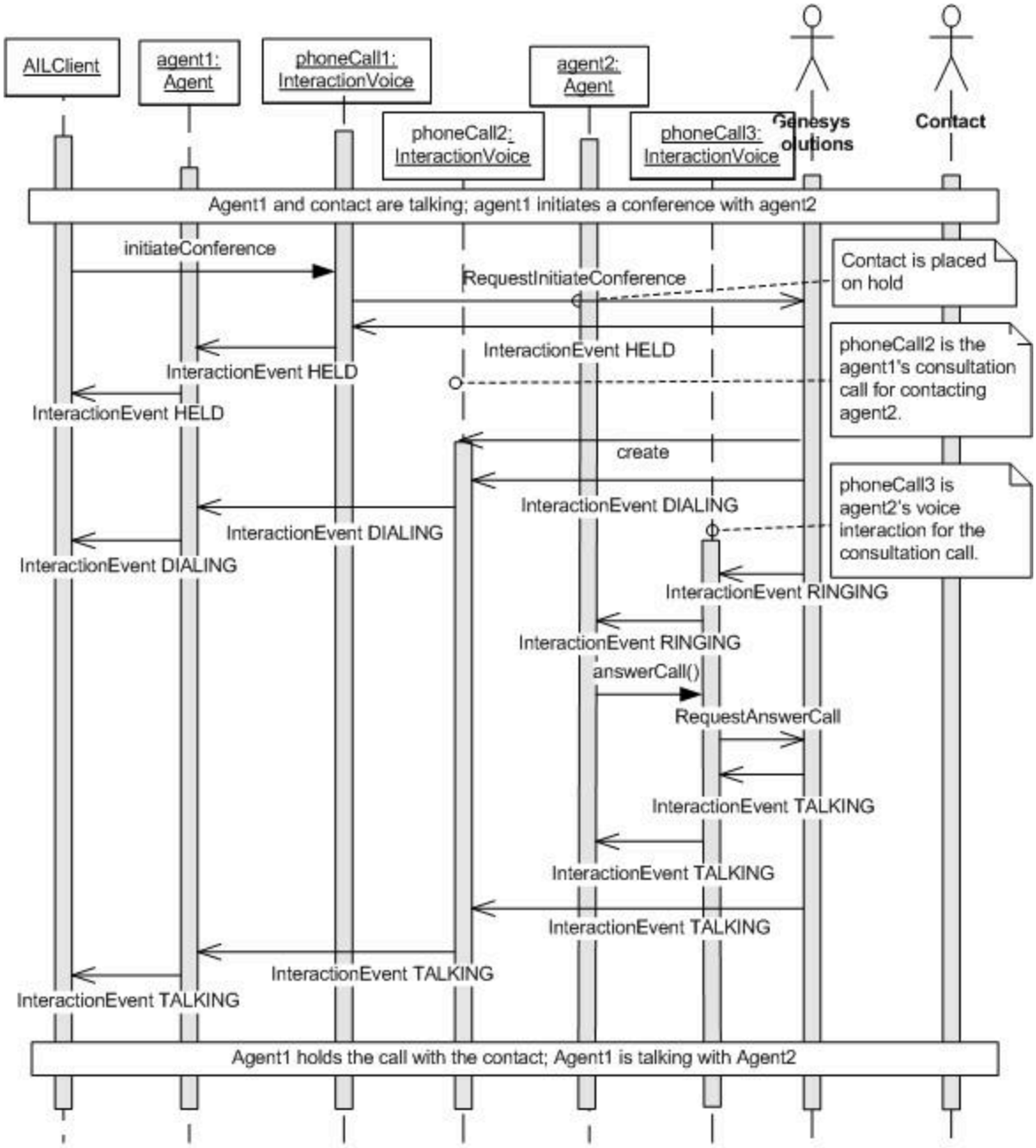
In the third step, when the two agents are ready to proceed, agent1 invokes the completeConference() method.

Next, agent1 receives an InteractionEvent event setting the status of the interaction referred by phoneCall2 to IDLE, then this interaction is destroyed.
Finally, agent1 receives two successive InteractionEvent events:

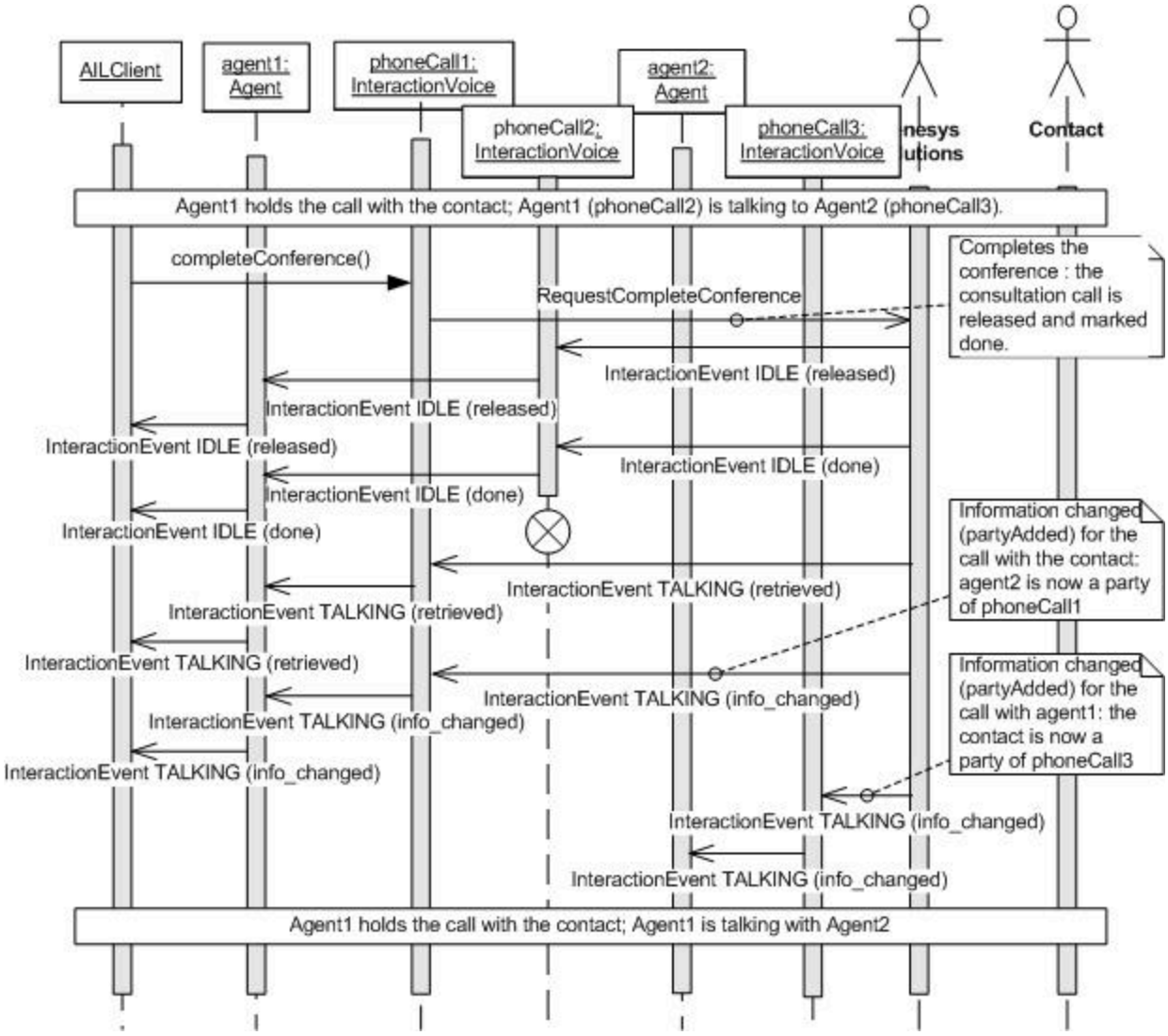
- One resuming the interaction phoneCall1 with the contact, setting it to the status TALKING.
- Another to notify the arrival of a third peer, agent2, on phoneCall1.

Agent2 also receives an InteractionEvent event notifying it of the arrival of a third peer, the contact, on its InteractionVoice phoneCall3.

The conference can now take place, and for each of the agents, each interaction is viewed as a standard phone call and must be ended accordingly.
The event flow is presented in the following two figures.



Call Conferencing, Initiating

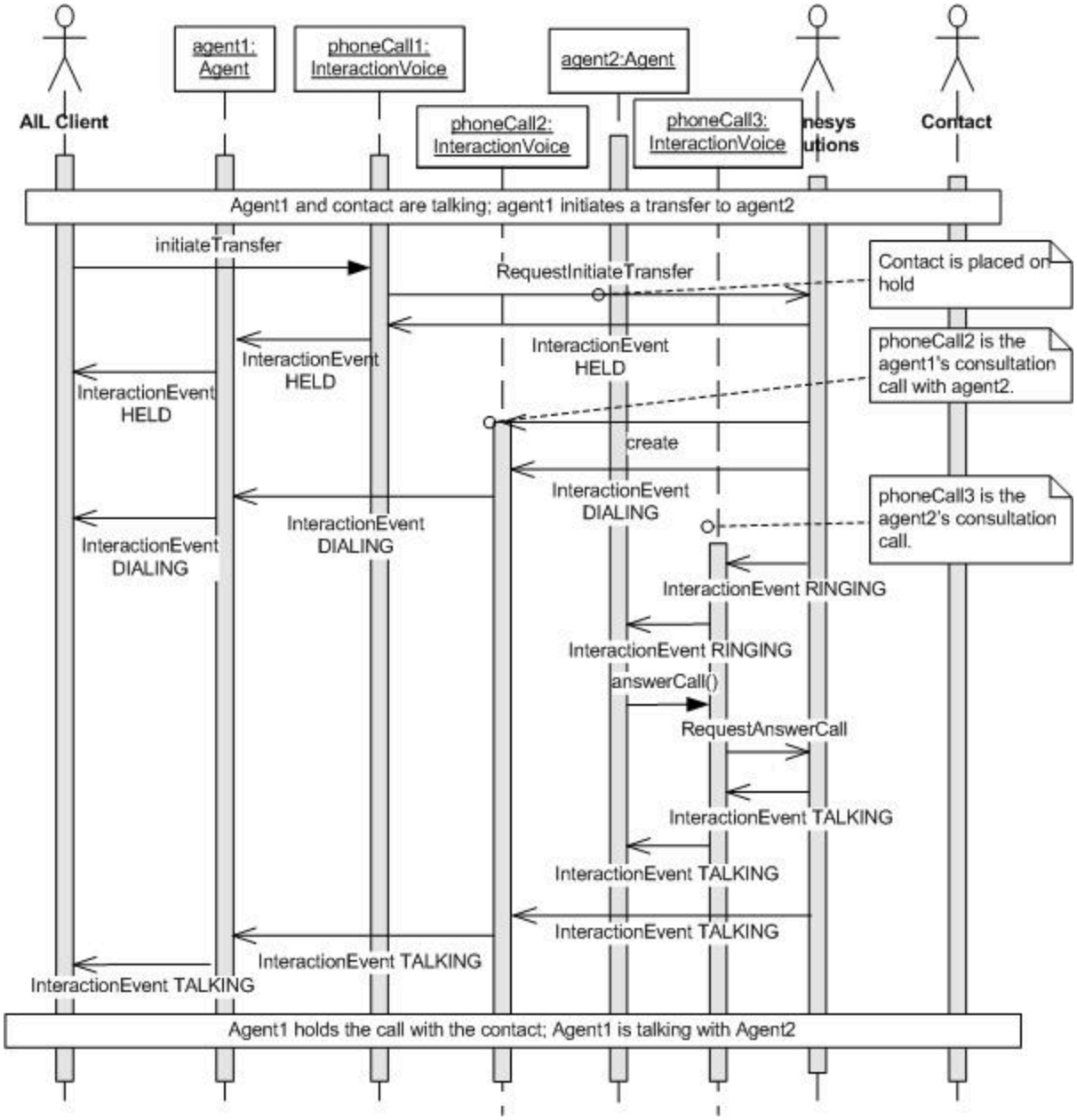


Call Conferencing, Completing

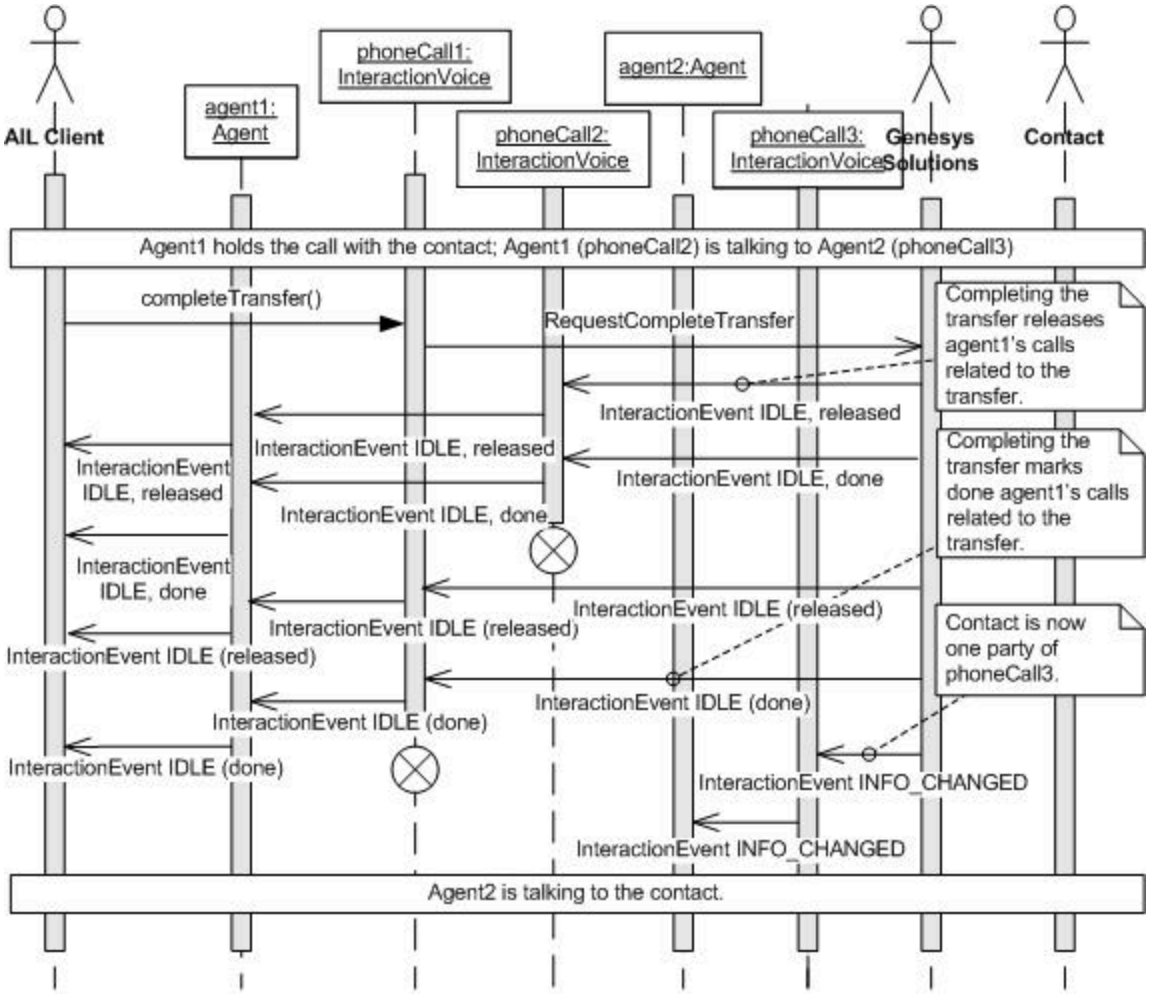
Transferring a Phone Call

Transferring a phone call follows the same steps as conferencing. Of course, all conference method calls must be replaced here by their transfer counterparts.
The difference lies in the fact that `agent1` does not resume its `InteractionVoice` `phoneCall1` at the end of the transfer. That is, instead of setting its first `InteractionVoice` to the status `TALKING` after the complete call, it sets it to `IDLE` and destroys it.

The event flow is presented in following two figures.



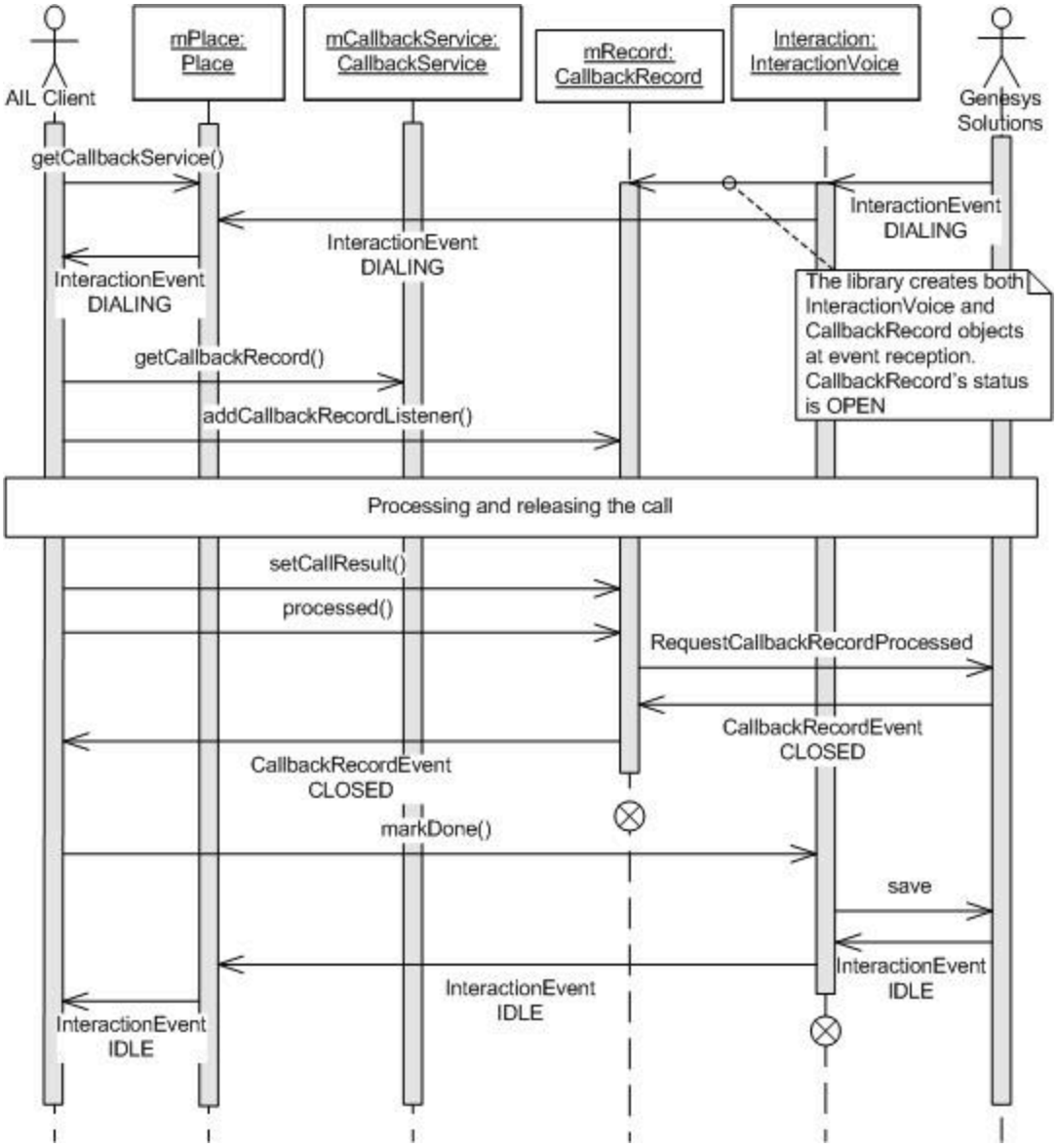
Transferring a Call, Initiating



Transferring a Call, Completing

Handling a Callback Phone Call

The following diagram shows the event flow for a callback record from a Callback Server in predictive mode.



Managing a Callback Record in a Predictive Callback Campaign