# Agent Interaction SDK Java Developer Guide

## API Overview

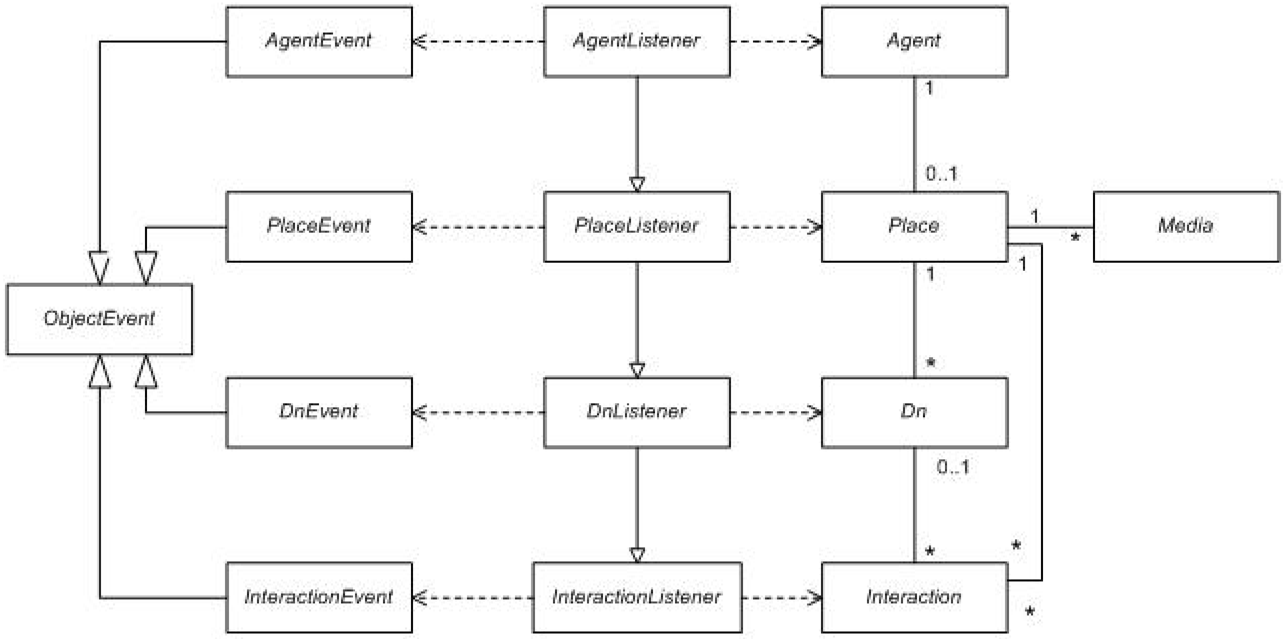
12/12/2025

## Contents

- 1 API Overview
    - 1.1 Objects and Events in AIL (Not applicable to all scenarios)
    - 1.2 Some Lightweight Interfaces for Agents and Interactions
    - 1.3 Packages
    - 1.4 Generics
    - 1.5 Agents
    - 1.6 Place, DNs, and Media
    - 1.7 Interactions
    - 1.8 Interaction Characteristics
    - 1.9 Events
    - 1.10 Event Listeners
    - 1.11 State and Possible Actions

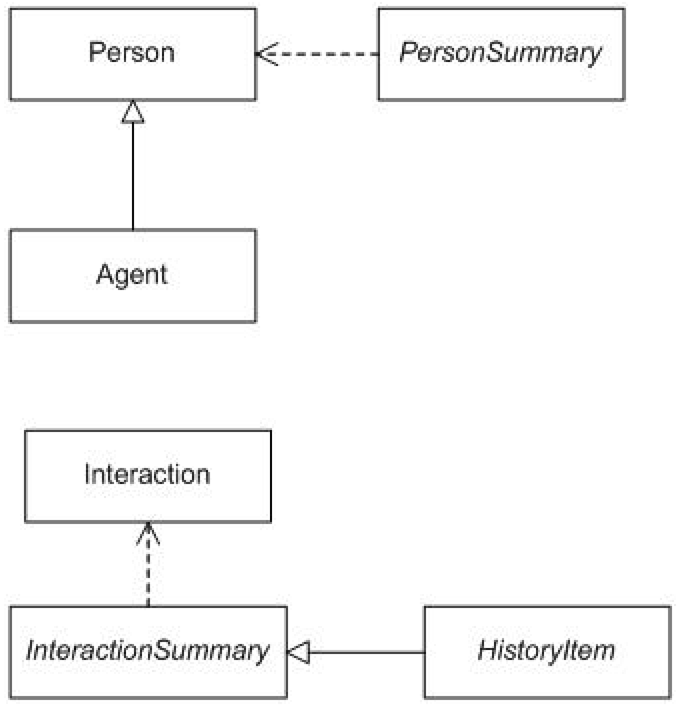
# API Overview

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.

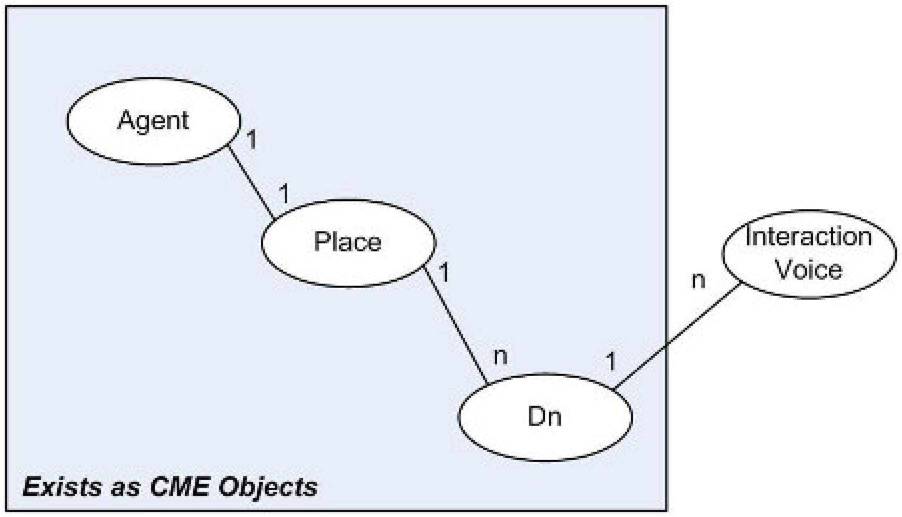
## Generics

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**

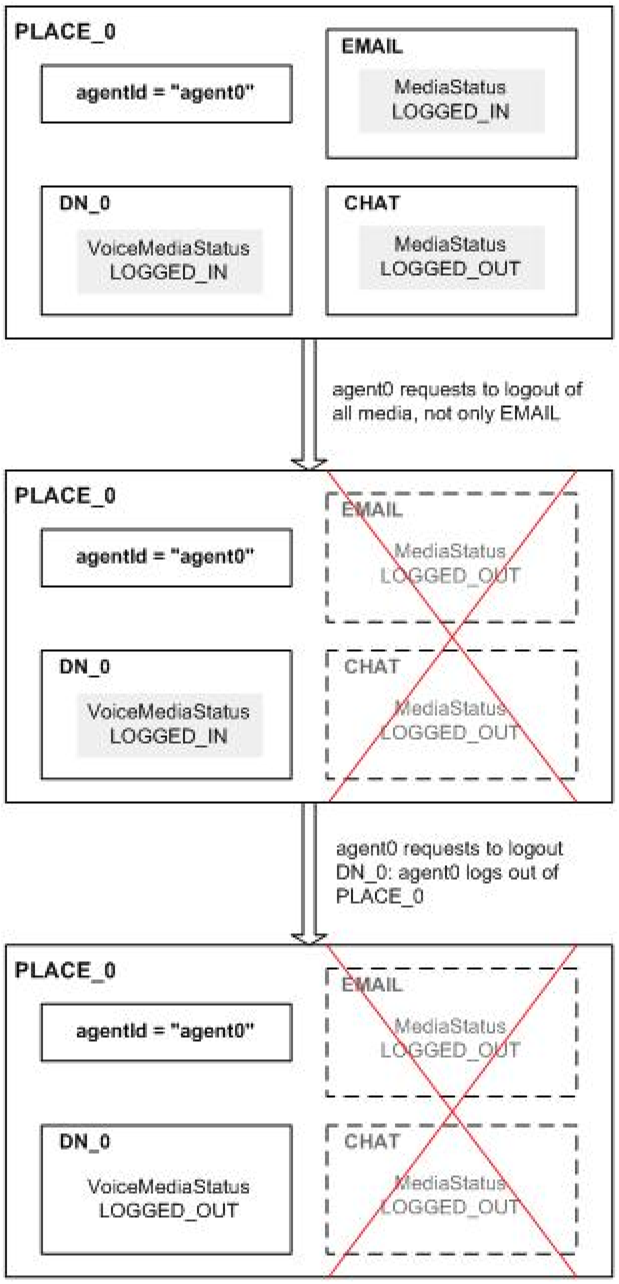
The `Agent` interface includes the following features:

- 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 a`logoutMultimedia(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**

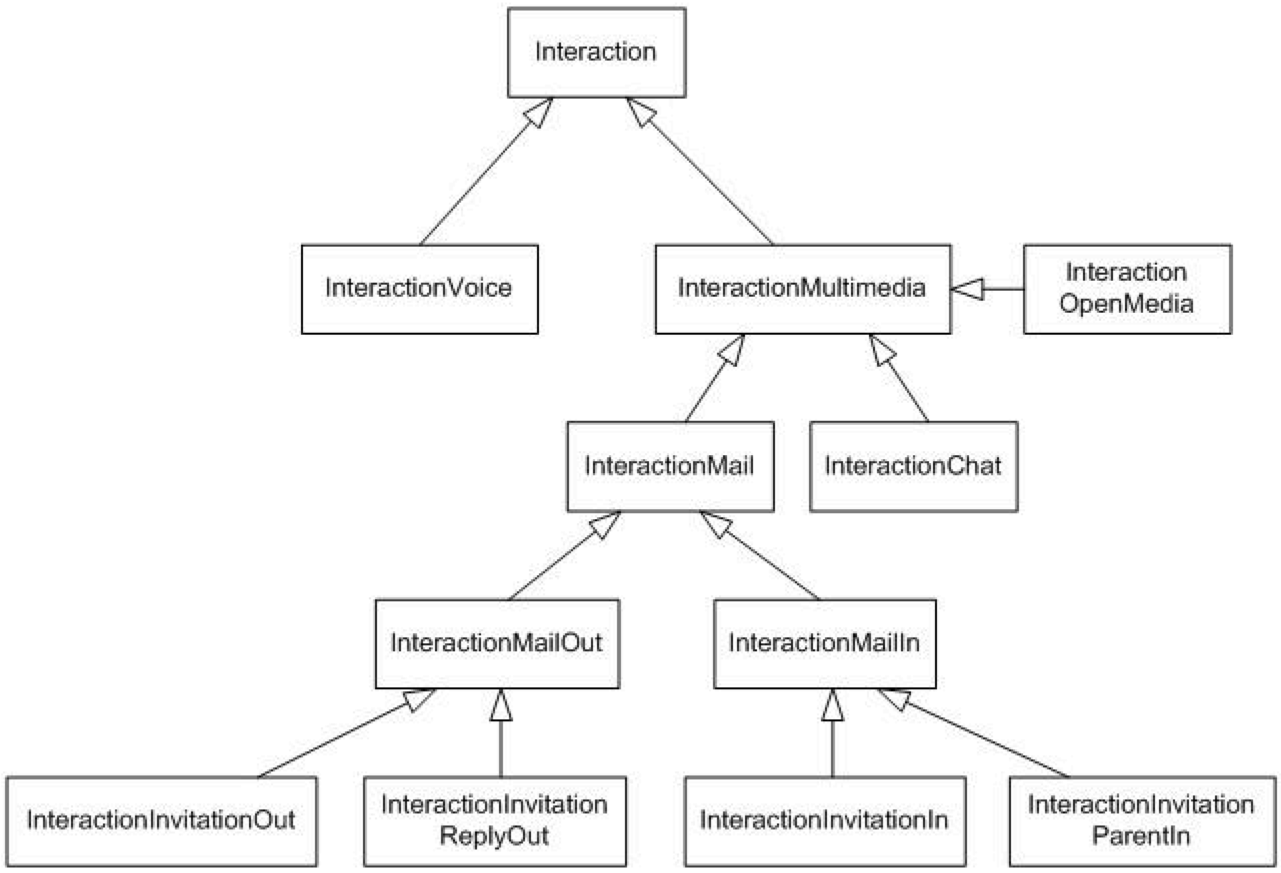
As shown above, if the agent logs out from all the media, they are removed from the place. Once the agent is logged in to the place, that agent has to log out of that place before another agent can use the place.

## 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.