GENESYS
AN ALCATEL-LUCENT COMPANY

Queued Interaction SDK 7.6

Java

Developer’s Guide

About Genesys

Genesys Telecommunications Laboratories, Inc., a subsidiary of Alcatel-Lucent, is 100% focused on software for call centers. Genesys recognizes that better interactions drive better business and build company reputations. Customer service solutions from Genesys deliver on this promise for Global 2000 enterprises, government organizations, and telecommunications service providers across 80 countries, directing more than 100 million customer interactions every day. Sophisticated routing and reporting across voice, e-mail, and Web channels ensure that customers are quickly connected to the best available resource—the first time. Genesys offers solutions for customer service, help desks, order desks, collections, outbound telesales and service, and workforce management. Visit www.genesyslab.com for more information.

Each product has its own documentation for online viewing at the Genesys Technical Support website or on the Documentation Library DVD, which is available from Genesys upon request. For more information, contact your sales representative.

Notice

Your Responsibility for Your System's Security

You are responsible for the security of your system. Product administration to prevent unauthorized use is your responsibility. Your system administrator should read all documents provided with this product to fully understand the features available that reduce your risk of incurring charges for unlicensed use of Genesys products.

Trademarks

Technical Support from VARs

If you have purchased support from a value-added reseller (VAR), please contact the VAR for technical support.

Technical Support from Genesys

If you have purchased support directly from Genesys, please contact Genesys Technical Support at the following regional numbers:

Region	Telephone	E-Mail
North and Latin America	+888-369-5555 or +506-674-6767	support@genesyslab.com
Europe, Middle East, and Africa	+44-(0)-118-974-7002	support@genesyslab.co.uk
Asia Pacific	+61-7-3368-6868	support@genesyslab.com.au
Japan	+81-3-6361-8950	support@genesyslab.co.jp

Prior to contacting technical support, please refer to the *Genesys Technical Support Guide* for complete contact information and procedures.

Ordering and Licensing Information

Complete information on ordering and licensing Genesys products can be found in the ***Genesys 7 Licensing Guide*.**

Released by

Genesys Telecommunications Laboratories, Inc. www.genesyslab.com

Document Version: 76sdk_dev_ixn_java-queued_02-2008_v7.6.001.00

Table of Contents

Preface

Welcome to the *Queued Interaction SDK 7.6 Java Developer's Guide.* This guide will show you how to develop applications that can monitor Genesys queues and connections to the Genesys framework and Genesys servers.

This document provides a high-level overview of the features and functions of Genesys Queued Interaction (Java API) 7.6, together with information about its architecture and deployment-planning materials. This document is valid only for the 7.6 release of this product.

Note: For versions of this document created for other releases of this product, please visit the Genesys Technical Support website, or request the Documentation Library CD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

This preface provides an overview of this document, identifies the primary audience, introduces document conventions, and lists related reference information:

The Genesys Queued Interaction SDK (Java API) is built around the Queued Interaction Layer library, which presents a Java API for developing monitoring applications.

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 Multimedia 7.6 features.
- Genesys Routing 7.6 features.

Usage Guidelines

The Genesys developer materials outlined in this document are intended to be used for the following purposes:

- Creation of contact-center agent desktop applications associated with Genesys software implementations.
- Server-side integration between Genesys software and third-party software.
- Creation of a specialized client application specific to customer needs.

The Genesys software functions available for development are clearly documented. No undocumented functionality is to be utilized without Genesys's express written consent.

The following Use Conditions apply in all cases for developers employing the Genesys developer materials outlined in this document:

1. Possession of interface documentation does not imply a right to use by a third party. Genesys conditions for use, as outlined below or in the *Genesys Developer Program Guide,* must be met.
2. This interface shall not be used unless the developer is a member in good standing of the Genesys Interacts program or has a valid Master Software License and Services Agreement with Genesys.
3. A developer shall not be entitled to use any licenses granted hereunder unless the developer's organization has met or obtained all prerequisite licensing and software as set out by Genesys.
4. A developer shall not be entitled to use any licenses granted hereunder if the developer's organization is delinquent in any payments or amounts owed to Genesys.

5. A developer shall not use the Genesys developer materials outlined in this document for any general application development purposes that are not associated with the above-mentioned intended purposes for the use of the Genesys developer materials outlined in this document.
6. A developer shall disclose the developer materials outlined in this document only to those employees who have a direct need to create, debug, and/or test one or more participant-specific objects and/or software files that access, communicate, or interoperate with the Genesys API.
7. The developed works and Genesys software running in conjunction with one another (hereinafter referred to together as the “integrated solutions”) should not compromise data integrity. For example, if both the Genesys software and the integrated solutions can modify the same data, then modifications by either product must not circumvent the other product’s data integrity rules. In addition, the integration should not cause duplicate copies of data to exist in both participant and Genesys databases, unless it can be assured that data modifications propagate all copies within the time required by typical users.
8. The integrated solutions shall not compromise data or application security, access, or visibility restrictions that are enforced by either the Genesys software or the developed works.
9. The integrated solutions shall conform to design and implementation guidelines and restrictions described in the *Genesys Developer Program Guide* and Genesys software documentation. For example:
   a. The integration must use only published interfaces to access Genesys data.
   b. The integration shall not modify data in Genesys database tables directly using SQL.
   c. The integration shall not introduce database triggers or stored procedures that operate on Genesys database tables.

Any schema extension to Genesys database tables must be carried out using Genesys Developer software through documented methods and features.

The Genesys developer materials outlined in this document are not intended to be used for the creation of any product with functionality comparable to any Genesys products, including products similar or substantially similar to Genesys’s current general-availability, beta, and announced products.

Any attempt to use the Genesys developer materials outlined in this document or any Genesys Developer software contrary to this clause shall be deemed a material breach with immediate termination of this addendum, and Genesys shall be entitled to seek to protect its interests, including but not limited to, preliminary and permanent injunctive relief, as well as money damages.

Chapter Summaries

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

- Chapter 1, "About the Queued Interaction (Java API)," on page 15. Introduces the Queued Interaction (Java API), its components, features, and scope of use.
- Chapter 2, "About the Code Examples," on page 23. Introduces the code examples that accompany this developer's guide.
- Chapter 3, "Feature Examples," on page 35. Explains `SimpleService`, which shows how to display connection services and monitor their status; `SimpleMonitorQueue`, which shows how to start and stop monitoring queues; `SimpleMonitorInteraction`, which shows how to monitor interactions; and `SimpleSupervisor`, which shows how to handle Ad'Hoc features.
- Chapter 4, "Alarm Examples," on page 55

Document Conventions

This document uses certain stylistic and typographical conventions—introduced here—that serve as shorthands for particular kinds of information.

Document Version Number

A version number appears at the bottom of the inside front cover of this document. Version numbers change as new information is added to this document. Here is a sample version number:

72fr_ref_09-2005_v7.2.000.00

You will need this number when you are talking with Genesys Technical Support about this product.

Type Styles

Italic

In this document, italic is used for emphasis, for documents' titles, for definitions of (or first references to) unfamiliar terms, and for mathematical variables.

Examples:

- Please consult the *Genesys 7 Migration Guide* for more information.
- *A customary and usual practice* is one that is widely accepted and used within a particular industry or profession.
- Do *not* use this value for this option.

- The formula, $x + 1 = 7$ where x stands for . . .

Monospace Font

A monospace font, which looks like `teletype or typewriter text`, is used for all programming identifiers and GUI elements.

This convention includes the *names* of directories, files, folders, configuration objects, paths, scripts, dialog boxes, options, fields, text and list boxes, operational modes, all buttons (including radio buttons), check boxes, commands, tabs, CTI events, and error messages; the values of options; logical arguments and command syntax; and code samples.

Examples:

- Select the `Show variables on screen` check box.
- Click the `Summation` button.
- In the `Properties` dialog box, enter the value for the host server in your environment.
- In the `Operand` text box, enter your formula.
- Click `OK` to exit the `Properties` dialog box.
- The following table presents the complete set of error messages T-Server® distributes in `EventError` events.
- If you select `true` for the `inbound-bsns-calls` option, all established inbound calls on a local agent are considered business calls.

Monospace is also used for any text that users must manually enter during a configuration or installation procedure, or on a command line:

Example:

- Enter `exit` on the command line.

Screen Captures Used in This Document

Screen captures from the product GUI (graphical user interface), as used in this document, may sometimes contain a minor spelling, capitalization, or grammatical error. The text accompanying and explaining the screen captures corrects such errors *except* when such a correction would prevent you from installing, configuring, or successfully using the product. For example, if the name of an option contains a usage error, the name would be presented exactly as it appears in the product GUI; the error would not be corrected in any accompanying text.

Square Brackets

Square brackets indicate that a particular parameter or value is optional within a logical argument, a command, or some programming syntax. That is, the parameter's or value's presence is not required to resolve the argument, command, or block of code. The user decides whether to include this optional information. Here is a sample:

```
smcp_server -host [/flags]
```

Angle Brackets

Angle brackets indicate a placeholder for a value that the user must specify. This might be a DN or port number specific to your enterprise. Here is a sample:

```
smcp_server -host <confighost>
```

Related Resources

Consult these additional resources as necessary:

- *Interaction SDK 7.6 Java Deployment Guide,* which is delivered on the documentation CD and details important configuration data.
- *Queued Interaction SDK 7.6 Java API reference*, which is located in the `doc/` subdirectory within the product installation directory tree.
- *Queued Interaction SDK 7.6 Java Code Examples*, which are located in `.zip` and `.tar.gz` archive files on the documentation CD in the `InteractionSDK_examples.java/` directory under the developer directory tree.
- *Open Media Interaction SDK 7.6 Application Blocks Guide*, which presents the application blocks for the Queued Interaction SDK, available on the product CD.
- The *Genesys Technical Publications Glossary,* which ships on the Genesys Documentation Library CD and which provides a comprehensive list of the Genesys and CTI terminology and acronyms used in this document.
- The *Genesys 7 Migration Guide*, also on the Genesys Documentation Library CD, which provides a documented migration strategy from Genesys product releases 5.1 and later to all Genesys 7.x releases. Contact Genesys Technical Support for additional information.
- The Release Notes and Product Advisories for this product, which are available on the Genesys Technical Support website at `http://genesyslab.com/support`.

Information on supported hardware and third-party software is available on the Genesys Technical Support website in the following documents:

- *Genesys 7 Supported Operating Systems and Databases*
- *Genesys 7 Supported Media Interfaces*

Genesys product documentation is available on the:

- Genesys Technical Support website at `http://genesyslab.com/support`.
- Genesys Developer website at `http://devzone.genesyslab.com`.

- Genesys Documentation Library CD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

Making Comments on This Document

If you especially like or dislike anything about this document, please feel free to e-mail your comments to `Techpubs.webadmin@genesyslab.com`.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this document. Please limit your comments to the information in this document only and to the way in which the information is presented. Speak to Genesys Technical Support if you have suggestions about the product itself.

When you send us comments, you grant Genesys a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

Chapter

1

About the Queued Interaction (Java API)

This chapter introduces the Queued Interaction SDK, its components, features, and scope of use. In this chapter you will find the following topics:

Overview

The Queued Interaction (Java API) lets you build Java applications to monitor queues available in the Genesys framework.

The Queued Interaction (Java API) 7.6 presents a simple Java API that allows for access to configuration information and monitoring of queue activity.

Components

The Queued Interaction (Java API) comprises the following:

- The Open Media Commons library, written entirely in the Java programming language, delivered as a set of `.jar` files on the product CD.
- The Queued Interaction Layer (QIL) library, also written entirely in Java, delivered as a set of `.jar` files on the product CD.
- The *Queued Interaction SDK 7.6 Java API Reference*, which is an HTML tree in the `docs/` directory of the installed product directory tree.

- A set of code examples that exercise some important features of the API, delivered in `.zip` and `.tar.gz` format on the documentation CD. For details, see "About the Code Examples" on page 23.
- *Open Media Interaction Application Blocks for Java*, available on the product CD, include application blocks to develop a QIL application. For further details, refer to the *Open Media Interaction SDK 7.2 Application Blocks Guide*.

Scope Of Use

The typical usage scenarios for the Queued Interaction (Java API) include:

- Enabling connections to Genesys servers, using the Open Media Commons library. Connection services involve the following components:
  - Interaction Server
  - Configuration Layer
- Getting overall configuration information:
  - Queues that have a strategy
  - Business attributes
- Monitoring:
  - Start monitoring a queue.
  - Stop monitoring a queue.
  - Listen for changes to queues—status and interaction activity.
  - Listen for changes to interactions—status and properties.
- Ad'Hoc management features:
  - Manage interactions (pull interactions, place an interaction in a queue, stop processing an interaction).
  - Modify interaction properties.
  - Get interactions through SQL queries.

Architecture

The Queued Interaction (Java API) is part of the 7.6 Open Media Interaction SDKs. It works with the Open Media Commons library to manage connections to the Genesys Framework and Genesys servers, as shown in Figure 1.

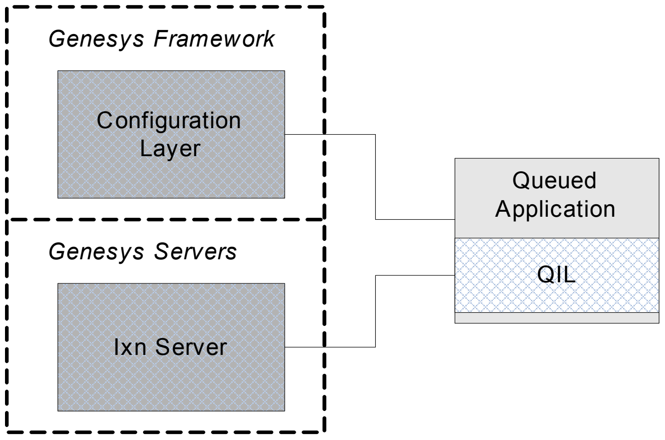


Figure 1: Architectural Overview

The QIL (Queued Interaction Layer) (Java API) exposes objects—such as `BusinessAttribute`, `ServiceInfo`, and `QILQueue`—as interfaces that provide access to information available through the Configuration Layer and Interaction Server.

The QIL library core is responsible for maintaining TCP/IP connections to servers, for maintaining the context, and for consolidating the data. The core also manages the state of the objects you use in your applications. You can listen to QIL events, which will notify your application of changes in the state of these objects.

Interfaces to Core Objects

You do not access the core objects of the Queued Interaction (Java API) directly. Rather, you get interfaces on them using the `QILFactory` or another QIL interface.

Your application uses the `QILFactory` interface to initialize, establish connections, and access the internal core factory object. The `QILFactory` object is a singleton.

Because of its singleton design, only one instance of the core factory object exists at runtime. All of the `QILFactory` interfaces in your application refer to this single object.

Application Development Design

The QIL library is designed to work across any network that provides TCP/IP access to Genesys servers.

If you use the Queued Interaction (Java API) to develop a client application—for example, a stand-alone desktop application—the QIL library runs in the same JVM as the client application code, GUI, and other related processes.

The Java client instantiates the QIL library, which establishes connections to the Genesys servers, as shown in Figure 2.

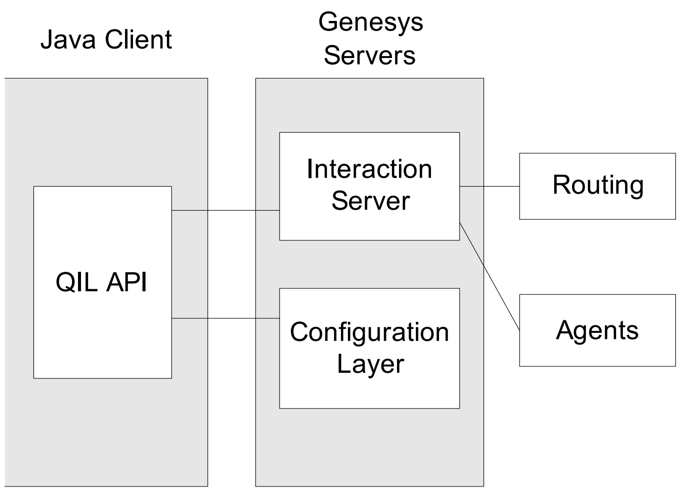


Figure 2: Client Architecture Design

If you employ the QIL library to develop a server application, you can use one of two options:

- 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 QIL library is embedded in a web container such as Tomcat.

If the library is loaded in a web container, a presentation servlet instantiates the QIL library, which establishes the connections to the Genesys servers, as shown in Figure 3 on page 18.

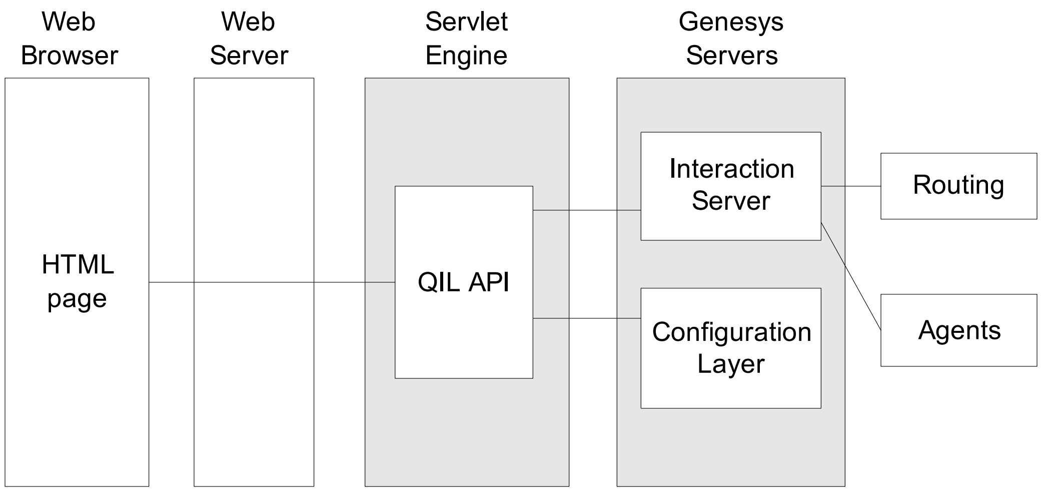


Figure 3: Web Server Architecture Design

In this model, clients access the web server, which requests pages from the web container. A presentation servlet requests data from the QIL library to build a page according to the current states and events.

Connectivity to Other Genesys Components

Connections to Genesys servers are maintained by the library core. There is a mechanism through which the Queued Interaction (Java API) user can be notified of changes in the status of the Genesys servers—namely, the loss of a connection.

Interaction Server

Interaction Server manages interactions and queues. The QIL library core monitors Interaction Server to obtain information about queues' activity, that is, added and deleted interactions. Additionally, the QIL library core provides Ad'Hoc Management to manage interactions through Interaction Server.

Configuration Layer

The Genesys Configuration Layer stores configuration information, such as application parameters; or object descriptions, such as queues, media types, interaction types, and interaction subtypes.

To run a Queued Interaction (Java API) application, you must define its application parameters in the Configuration Layer.

The QIL and Open Media Commons libraries provide full integration with Genesys Configuration Layer objects such as `QILQueue`, `BusinessAttribute`, and `BusinessAttributeValue`.

API Overview

The Queued Interaction (Java API) presents an interface for working with queue activity by abstracting Configuration Layer objects and queue event flow. Your client application design is largely a matter of monitoring the event flow of various queues and presenting this information to the user. You will do this by implementing event listeners on objects.

As you receive events that reflect changes in the state of an object, your application should make method calls accordingly. To ensure good performance and avoid deadlocks, listeners should only contain the bare minimum amount of code. Any lengthy processing should be delegated to a separate thread (see the section "Event and Listeners" on page 20).

Packages

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

- `com.genesyslab.omsdk.commons`—Exposes the Open Media Commons classes for connecting to servers, accessing connection service,s and getting application information.
- `com.genesyslab.omsdk.commons.event`—Exposes classes and interfaces for notification of connection-related events.
- `com.genesyslab.omsdk.commons.exception`—Exposes exceptions thrown by Open Media Commons API methods.
- `com.genesyslab.omsdk.qil`—Exposes the main QIL API classes.
- `com.genesyslab.omsdk.qil.events`—Exposes classes and interfaces for notification of QIL-related events.
- `com.genesyslab.omsdk.qil.exception`—Exposes exceptions thrown by QIL API methods.

Event and Listeners

The QIL library core provides a push model through the Observer design pattern. For instance, objects such as `QILFactory` and `QILQueue` implement this pattern.

The following sections give you further details about the Observer pattern and the event-thread implementation that Genesys recommends for QIL.

Event Push Model

This model involves sending an event on an object to a listener, which permits each object to implement its own set of listeners and methods.

Generally, a listener declares only one method, a `handle*Event()` 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.

Threads and Listeners

QIL events are time-ordered and should be published in listeners as soon as they occur, to ensure workflow and information consistency.

In the Open Media Commons library, events occurring for a service cannot block events of another service.

If you want to perform a long treatment, or a treatment making calls to QIL methods, be sure 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.

What's Next

The next chapter goes into greater detail about the examples provided with this Java API. It provides installation instructions and gives a basic explanation of the supported features.

Chapter

2

About the Code Examples

This chapter introduces the code examples that accompany this developer's guide. It presents essential design considerations and also some of the initial tasks an application will have to carry out to use the QIL library. This chapter contains the following sections:

Overview of the Code Examples

All examples are Java client applications that integrate a specific set of the features provided with QIL.

- `OpenMediaSdkData`—reads the `MediaSDK.properties` file, which contains connection data.
- `OpenMediaSdkGUI`—provides a unified graphical user interface for the examples.
- `OpenMediaSdkTableModel`—provides a table model for `OpenMediaSdkGui`.
- `SimpleConnector`—establishes connections to servers, based on information contained in `OpenMediaSDKData`. This example shows you how to connect to servers, using the Open Media Commons library.
- `SimpleService`—displays and monitors connections using the Open Media Commons library.
- `SimpleMonitorQueue`—extends `SimpleService` and manages queue monitoring.
- `SimpleMonitorInteraction`—extends `SimpleService` and manages interaction monitoring for monitored queues.

- `SimpleSupervisor`—extends `SimpleService` and provides additional ad' hoc features.
- `SimpleQueueAlarm`—monitors a queue and sends an alarm depending on the number of interactions available in the queue.
- `MultipleAlarm`—monitors all queues, displays the queues' activity and fires alarms according to the number of interactions available in queues.

Installing the Code Examples

In order to develop applications with the Queued Interaction (Java API), you will need a compiler, such as the one delivered in the Java 2 Standard Edition (J2SE) SDK. It must conform to release 1.4.2 or 1.5.

In this guide, JDK 1.4.2 from Sun Microsystems was used to compile and run the code examples.

Before you install and use the examples, install the Queued Interaction SDK Library. Refer to *Interaction SDK 7.6 Java Deployment Guide* for further details.

Then, set the following environment variables:

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

Source-Code Examples

The source code for the examples is contained on the Genesys Documentation CD. When you expand the `sdk_exmpl_ixn_java-queued` archive file containing the code examples, you will find the following directory structure:

- 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 (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 (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 `OpenMediaSDK.properties` file (used by the `OpenMediaSdkData` class in `OpenMediaSdkData.java`).
- Java class files are stored in the `classes/` directory as you compile them.

- Source files are stored in the `queued/sdk/java/examples/` directory.
- There is also a `doc/` directory containing Javadoc comments for each of the examples.

Using the Code Examples

The examples are designed to run with the Genesys Configuration Layer and Interaction Server.

For the examples provided with this document to work, they need valid configuration data, including connections to servers and configuration objects such as queues and business attributes (media type, interaction type, subtype, and so on.)

For configuration details, see the *Interaction SDK Java 7.6 Deployment Guide*.

Introducing the Queued Interaction Code Examples

These code examples were designed to be interactive and to isolate API-related code from presentation-related code as much as possible. This design should make it easier for you to learn the functionality of the Queued 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 in Figure 4 on page 26. As you are learning the API functionality, you can ignore the `OpenMediaSdkData`, `OpenMediaSDKGUI`, and `AlarmGUI` classes.

In 7.6, code examples include some of the Open Media Interaction Application Blocks for Java, which present best practices for the Queued Interaction (Java API) and the Media Interaction (Java API). All the examples are built on top of the `SimpleConnector` class that includes the `ConnectorQIL` application block and will be explained in the next section.

All the code examples that inherit the `SimpleService` class are explained in the next chapter and demonstrate Queued Interaction (Java API) features, such as monitoring service status, monitoring queues, monitoring interactions, and so on.

The `SimpleQueueAlarm` and the `MultipleAlarm` code examples are additional examples that show queue alarm scenarii.

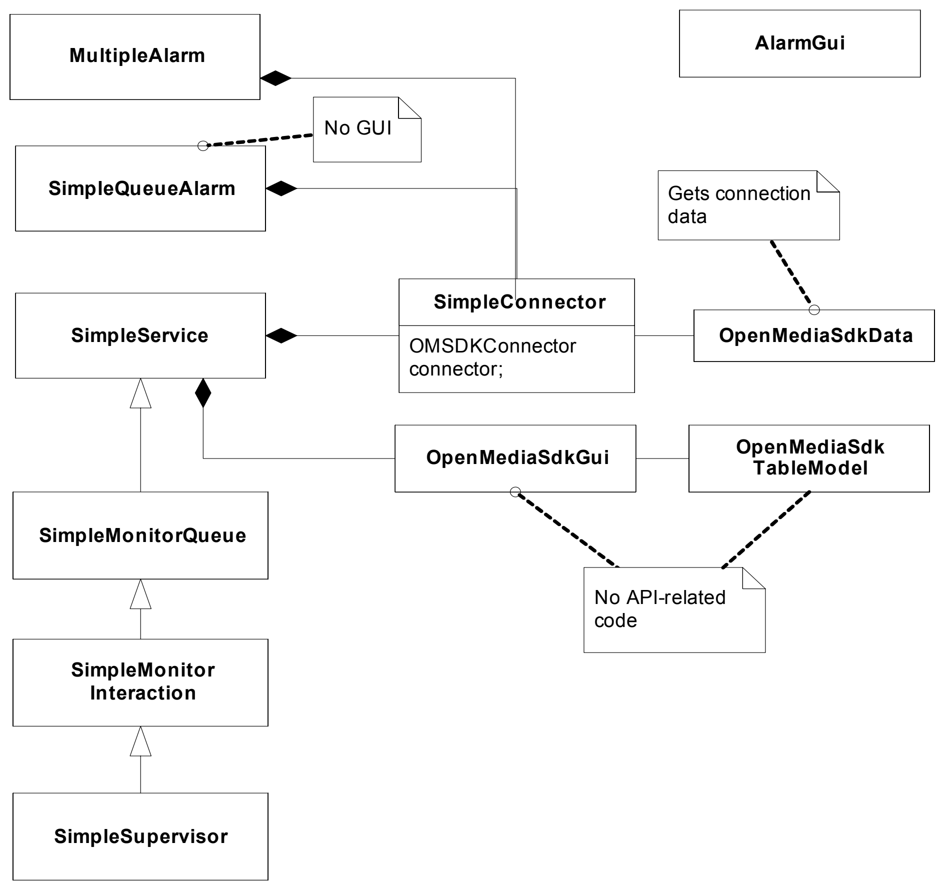
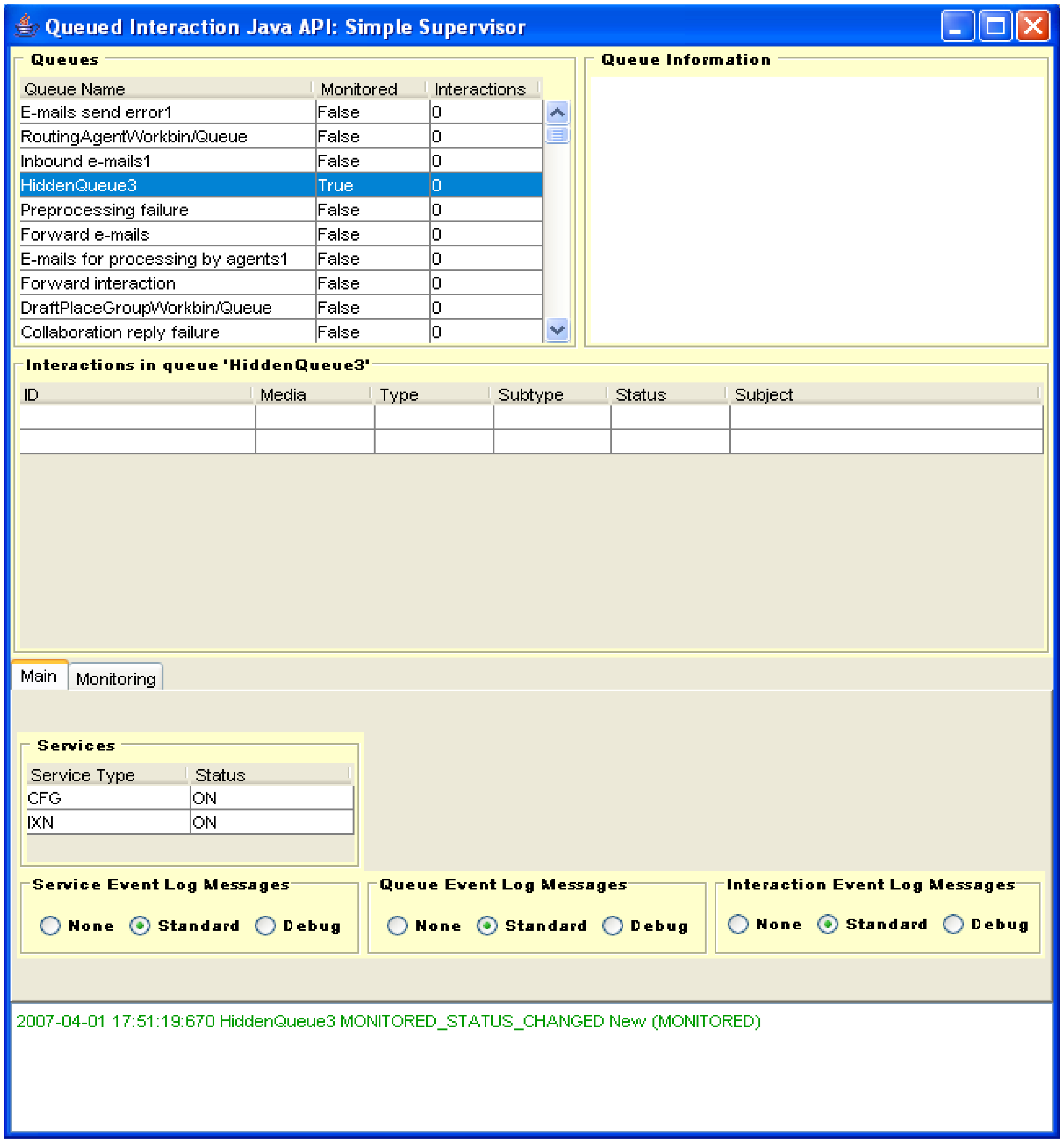


Figure 4: Architectural Overview of the Queued Interaction Examples

Figure 5 shows the user interface for the code examples that inherit `SimpleService`. The window title indicates the name of the runned example, `SimpleSupervisor`. As you can see, some components of the window have a light yellow background. This shows you which section of the GUI is active for the example you are working on.


Queued Interaction Java API: Simple Supervisor

Queues

Queue Name	Monitored	Interactions
E-mails send error1	False	0
RoutingAgentWorkbin/Queue	False	0
Inbound e-mails1	False	0
HiddenQueue3	True	0
Preprocessing failure	False	0
Forward e-mails	False	0
E-mails for processing by agents1	False	0
Forward interaction	False	0
DraftPlaceGroupWorkbin/Queue	False	0
Collaboration reply failure	False	0

Queue Information

Interactions in queue 'HiddenQueue3'

ID	Media	Type	Subtype	Status	Subject

Main | Monitoring

Services

Service Type	Status
CFG	ON
IXN	ON

Service Event Log Messages: None [✓ Standard] Debug

Queue Event Log Messages: None [✓ Standard] Debug

Interaction Event Log Messages: None [✓ Standard] Debug

```
2007-04-01 17:51:19:670 HiddenQueue3 MONITORED_STATUS_CHANGED New (MONITORED)
```

Figure 5: User Interface for Runnable Examples

The `Queues` and `Queues Information` panels at the top of the user interface display queue information available through the `SimpleMonitorQueue` code example. The center panel presents the list of interactions if the queue selected in the Queues list is monitored.

Then, there are two tabbed pane. The main tab shows `Services` status and includes controls for the bottom log panel. Radio buttons allow you to control the display of the event messages that appear in the bottom pane of the user interface. The examples generate service and queue 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 type of log event.

In order to make the event messages easier to tell apart, they have been assigned their own colors. For instance, queue status events appear in green, as shown in Figure 5.

The `Monitoring` tab provides the user with monitoring and supervisor features for Ad'Hoc Management. Depending on the code example that you run (`SimpleMonitorQueue`, `SimpleMonitorInteraction`, or `SimpleSupervisor`) different buttons are activated.

If you have comments on these examples, please contact Genesys. Information on how to contact Genesys is provided at the end of the preface, in section "Making Comments on This Document" on page 13.

Open Media Commons

The Open Media Commons library includes all of the components required to connect to the Genesys servers. This section discusses the contents of the Open Media Commons library, including essential API connection features needed for every application.

The discussion refers to the `SimpleConnector.java` example in the `sdk_exmpl_ixn_java-queued/queued/sdk/java/examples` directory. This example is not a stand-alone application: the example classes use it to handle all of its connection-related tasks.

Before running the examples, be sure to edit the `OpenMediaSDK.properties` file to specify the correct data in your Configuration Layer (host, port, application name, and so on.) You will also need to compile the Open Media Interaction Applications Blocks for Java and the other `.java` files in addition to compiling `SimpleService.java`.

Every application, whether client or server, must use the `OMSDKConnector` class, passing correct configuration data arguments. The `ConnectorQIL` application block is in charge of this task and returns a reference to the `QILFactory`.

The remainder of this section focuses mostly on classes and interfaces of the `com.genesyslab.omsdk.commons.*` packages.

Connection

Every QIL application must use the `OMSDKConnector` class to initialize the Open Media Commons library. To do this, the application must pass in the valid configuration data.

To make connections possible and start the Open Media Commons library, your application will need to do two basic things:

- Set initialization parameters.
- Initialize the library.

To set initialization parameters, you need to create and fill in an instance of `InitializationParameters`. Before doing that, you must set or obtain the following minimum configuration data:

- Configuration Layer host name.
- Configuration Layer port.
- Reconnection period.
- Maximum number of reconnection attempts.

In the `ConnectorQIL` application block, this information is passed on to the `connect()` method.

```
InitializationParameters initParams = new InitializationParameters(
                primaryHost,
                primaryPort,
                backupHost,
                backupPort,
                applicationName,
                reconnectionPeriod,
                reconnectionAttempts);
```

Then, to properly fill in this object, you need to specify the correct list of services that your application will use. Therefore, you must get an `InitializationServices` instance by calling the static `QILFactory.getInitializationServices()` method. The returned object contains all the services you need and you can add it to your `InitializationParameters` object, as shown here:

```
initParams.addInitializationServices(
                QILFactory.getInitializationServices());
OMSDKConnector.initialize(initParams);
```

Once you have initialized the Open Media Commons library through the `OMSDKConnector` interface, you can initialize the `QILFactory` singleton.

```
QILInitializationParameters config = new QILInitializationParameters();
QILFactory.initialize(config);
```

Now you are ready to use the QIL library.

Services

Connections are represented as services available through the `OMSDKConnector` interface. After you have connected, your application uses the features of these services to monitor the state of these connections and take into account possible disconnections.

Types of Services

Service features are available for several types of services—see `ServiceType` for further details:

- `CONFIGURATION`—Connection to the Configuration Layer. This connection lets your application access configuration information.
- `INTERACTION_SERVER`—Connection to Interaction Server. This connection lets your application monitor queues.

Service Interfaces

The `OMSDKConnector` interface is the entry point to the service features:

- It accesses each `ServiceInfo` instance that associates a `ServiceStatus` with a `ServiceType`.
- It lets your application associate a `ServiceListener` with a service type, in order to track status changes, which are propagated in `ServiceEvent` events.

For further details about interfaces, see the *Queued Interaction SDK 7.6 Java API Reference*.

The `SimpleService` example demonstrates the use of these interfaces. See “SimpleService” on page 37.

Queued Interaction (Java API)

Queued Interaction (Java API) includes all of the components you will use to monitor queues. This section discusses the main features available through the API, including the QIL initialization needed for every application.

This section will focus on the classes and interfaces of the `com.genesyslab.omsdk.qil.*` packages.

QILFactory

`QILFactory` is the entry point to Queued Interaction (Java API). You need this interface to initialize the library so you can access the other QIL interfaces.

Before initializing the QIL library, you must enable connections to servers with the `OMSDKConnector` class. See "Connection" on page 29 for further details.

After you have connected, the initialization is straightforward, as shown in this code snippet.

```
QILFactory.initialize(new QILInitializationParameters());
```

Then, you must call the static `QILFactory.getQILFactory()` method to get a reference on the `QILFactory` to access the main QIL interfaces—for instance, `QILQueue` and `BusinessAttribute`.

Configuration Data

After your application has connected, the QIL configuration features give you access to the Configuration Layer data for this application's tenant, such as:

- Application information.
- Business attributes:
  - Media types defined in the `Media Type` section.
  - Interaction types defined in the `Interaction Type` section.
  - Interaction subtypes defined in the `Interaction Subtype` section.

Business Attributes

The interfaces related to business attributes defined for the application's tenant include:

- `BusinessAttribute`—Metadata for a business attribute's type.
- `BusinessAttributeValue`—A value for a business attribute's type.

The `BusinessAttribute` interface defines metadata for information concerning interactions. The `QILFactory` interface can access three types of business attributes defined in the `BusinessAttributeType` class:

- `MEDIA_TYPE`—Business attribute for media types.
- `INTERACTION_TYPE`—Business attribute for interaction types.
- `INTERACTION_SUBTYPE`—Business attribute for interaction subtypes.

To get a business attribute, use a method of your `QILFactory`—for instance, the `QILFactory.getBusinessAttribute()`.

Each business attribute contains a set of `BusinessAttributeValue` interfaces. Each `BusinessAttributeValue` describes a value characterized by the parent `BusinessAttribute`.

If your application handles interactions, use business attributes to display more information about these interactions. Each `QILInteraction` instance contains a value name for each business attribute type.

You can access the corresponding `BusinessAttributeValue` (if any) as shown in the following code snippet:

```
//Getting the media type name of a QILInteraction interface
ip.addInitializationServices(initSrvForQIL);
String mediaTypeName = myQILInteraction.getMediaType();
BusinessAttribute mediaTypes = QILFactory.getQILFactory().getMediaTypes();
BusinessAttributeValue mediaTypeValue = mediaTypes.getValue(mediaTypeName);
```

Queues

The main interface for dealing with queues is `QILQueue`. You can access it from one of the `QILFactory` methods.

If the queue monitoring status is `QILQueueMonitorStatus.MONITORED`, it means you can get events for interaction activity depending on which listeners you add to the `QILQueue` interface:

- `QILQueueContentChangedEvent` events, provided through the `QILQueueListener`, concern interactions added to or deleted from the `QILQueue`.
- `QILInteractionEvent` events, provided through the `QILInteractionListener`, concern status and property changes in interactions.

These events let your application access the `QILInteraction` objects that contain detailed information about an interaction. For further details, see the *Queued Interaction SDK 7.6 Java API Reference*.

The `SimpleMonitorQueue`, `SimpleMonitorInteraction`, `SimpleQueueAlarm`, and `MultipleAlarm` examples demonstrate the concurrent use of `QILQueue`, `QILQueueListener`, and `QILInteractionListener` interfaces. See Chapter 3 on page 35 and Chapter 4 on page 55.

Ad'Hoc Management

First, to access Ad'Hoc Management features, your application must be in supervisor mode. Switch to supervisor mode by calling the `QILFactory.changeOperationalMode()` method, as shown in the following code snippet:

```
try {
   mQILFactory.changeOperationalMode(
      QILOperationalMode.SUPERVISOR_MODE );
} catch (QILUnsuccessfulModeChangeException e) {
   e.printStackTrace();
   System.out.println("Cannot change operation mode " + e);
}
```

By default, QIL is started in reporting mode. To set supervisor mode at application startup, change the operational mode before you initialize the factory, as shown here:

```
QILInitializationParameters config = new
QILInitializationParameters();
config.setOperationalMode(QILOperationalMode.SUPERVISOR_MODE);
QILFactory.initialize(config);
```

Manage Interactions

To manage interactions, you need an instance of the `QILInteractionManager` interface, that you retrieve with a single call to the `QILFactory.getInteractionManager()` method, as shown here:

```
QILInteractionManager interactionManager = mQILFactory.getInteractionManager();
```

Then, to manage an interaction or modify its properties, your application has to become its owner. You can either:

- Pull the interaction. In this case, place it into a queue, or leave it if you no longer need to move it.
- Lock the interaction, process the changes, and unlock the interaction to make it available for agents.

Refer to the *Queued Interaction SDK 7.6 Java API Reference* for further details about the `QILInteractionManager` interface.

Get Interactions Through SQL Queries

The `QILQueue` interface provides the synchronous `getInteractionsByQuery()` method and the asynchronous `asyncGetInteractionsByQuery()` method to retrieve interactions by SQL queries.

The following code snippet shows a synchronous SQL query.

```
QILInteractionQueryBySQL query = new QILInteractionQueryBySQL();
query.setSqlCriteria(new StringBuffer().append
     ("id = \"").append(myIxnID).append("\"").toString());
QILInteractionList list = queue.getInteractionsByQuery(query,
false);
```

The returned `InteractionList` instance contains IDs for the `QILInteraction` objects to be retrieved.

Note: The options defined in the Interaction Server limits the maximum number of returned IDs. Refer to the Multimedia 7.6 Deployment Guide for further details.

What’s Next

The next chapter will go in further details about the examples provided with this SDK. It gives an explanation of how to use QIL to listen to connections and monitor queues.

Chapter

3 Feature Examples

This chapter explains four examples, `SimpleService.java`, `SimpleMonitorQueue.java`, `SimpleMonitorInteraction.java`, and `SimpleSupervisor.java`.

This chapter comprises the following sections:

Introduction

To follow the discussion in this chapter, you will need the *Queued Interaction SDK 7.6 Java API Reference*, which is located in the `doc/` subdirectory under the Queued Interaction product installation directory, and the source code for the `SimpleService.java`, `SimpleMonitorQueue.java`, `SimpleMonitorInteraction.java`, and `SimpleSupervisor` examples. Refer to the discussion in Chapter 2,“About the Code Examples” for more information on how to use the examples.

This set of code examples uses Open Media Interaction Java Application Blocks, according to the following principles:

- Instantiating the application block to use it as a regular class.
- Using parts of the application block by copying and pasting its source code into your application.
- Extending the application block to fulfill your needs.

For further details about the application blocks used in these code examples, refer to the javadoc delivered in the `doc/` subdirectory of their source code.

More Application Essentials

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

- **Connect to servers.** The examples use the `SimpleConnector` class to do this, because, as explained earlier, it includes the `ConnectorQIL` application block:

```
SimpleConnector connector = new SimpleConnector();
```

See “Open Media Commons” on page 28 for further details.

- **Implement a listener** from among those provided by QIL. The examples use a `ServiceListener` for listening to service changes and a `QILQueueListener` for listening to queues, and a `QILInteractionListener` for listening to interactions. Here is how `SimpleService` implements a `ServiceListener`:

```
public class SimpleService implements ServiceListener {
```

- **Set up the GUI components** tied to QIL functions. The examples have a `linkWidgetsToGui()` method that does this.
- **Add event-handling code** to the appropriate QIL event handler. The examples demonstrate the use of the `ServiceListener.handleServiceEvent()` and `QILQueueListener.handleQueueEvent()` methods.
- **Register your application** for events on the object that your listener refers to. For instance, the `SimpleMonitorQueue` example uses a `QILQueueListener`, so it uses this method call to register with the `QILQueue` object:

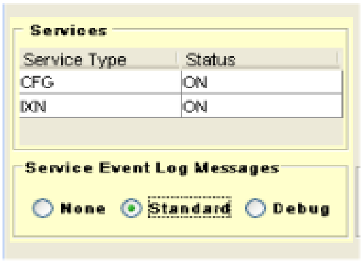
```
queue.addQueueListener(this);
```

- **Synchronize the user interface** with the state of the QIL objects referred to by your application. The examples have several methods for this, including `setQueueWidgets()` and `handleQueueSelection()`.

The 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’s time to see how they are implemented in the `SimpleService` example.

SimpleService

The `SimpleService` program provides a GUI-based desktop application that displays service status and service events in real-time. These tasks use `ServiceEvent` and correspond to the `Main` tab of the user interface presented on page 27 and shown again in Figure 6.



Service Type	Status
CFG	ON
IXN	ON

Service Event Log Messages: None [✓ Standard] Debug

Figure 6: Select a Trace Level with SimpleService

This section will focus on the API features for working with `ServiceEvent` events. Here is how `SimpleService` carries out the six steps to writing a QIL application.

Implement a Listener

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

```
public class SimpleService implements ServiceListener {
```

`SimpleService` uses `ServiceListener` because it can handle the `ServiceEvent` events that this example uses to update the `Services` table.

Connect to Servers

As explained earlier, the example uses the `SimpleConnector` class to establish the all-important connection with the Genesys servers and initialize the `OMSDKConnector`. For more information on how this is done, you can refer to "Open Media Commons" on page 28. For the purposes of this example, here is all you need to do in the constructor:

```
SimpleConnector connector = new SimpleConnector();
```

Set Up the GUI Components

The `SimpleService` constructor calls the `linkWidgetsToGui()` method which uses the `connection.qilFactory` instance to fill in the `Services` table.

The method gets a `ServiceInfo` object for each `ServiceType` enumerated value used by this example, that is, `ServiceType.CONFIGURATION` and

`ServiceType.INTERACTION_SERVER`. Then it calls the `OpenMediaSdkGui.setService()` method to add the service to the table, as shown in the following code snippet:

```
// Getting a ServiceInfo interface
// for the Configuration service
ServiceInfo srv = connection.connector.getServiceInfo(ServiceType.CONFIGURATION);

if(srv != null)
     sdkGui.setService(srv.getType().toString(), srv.getStatus().toString());
```

Register Your Application

The next step is to register your application so it can receive the events you will need to work with the services. You register the listener on each connection service you need to listen to.

In this case, the example registers for both the interaction and the configuration services so that the `Services` table will be updated.

This is done by `linkWidgetsToGui()` after getting each available service:

```
try {
   //Getting service information
   //...
   //Registering for this service
   //THIS IS AN IMPORTANT STEP:
   OMSDKConnector.addServiceListener(this,ServiceType.INTERACTION_SERVER);
} catch (Exception exception) {
    exception.printStackTrace();
}
```

After registering, if the services change, the QIL library gets a `ServiceEvent` object and calls the `handleServiceEvent()` method implemented in this example.

Add Event-Handling Code

Classes implementing the `ServiceListener` interface must include the `handleServiceEvent()` method. As Queued Interaction (Java API) is used for publishing event, you do not implement extended processing directly in this event handler: You do it in a thread. For more information, see “Threads and Listeners” on page 20.

`SimpleService` uses the `ServiceEventThread` class to process `ServiceEvent`, as shown here.

```
public void handleServiceEvent(ServiceEvent event) {
   ServiceEventThread p = new ServiceEventThread(event);
   p.start();
}
```

Processing `ServiceEvents` is performed in the `ServiceEventThread.run()` method. Most of the code in this method is for writing messages to the log panel at the bottom of the `SimpleService` user interface. Since `SimpleService` only needs to update a line in the `Services` table when a service event occurs, all the GUI synchronization is done there:

```
ServiceInfo service = event.getServiceInfo();
sdkGui.setService(service.getType().toString(),
          service.getStatus().toString());
```

The next example will have more complicated event-handling code, but for this example, this is all you have to do for your event handlers.

Wrapping Up

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

Set Example Type

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

Connect to QIL and Make Configuration Data Available

Next, a new instance of `SimpleConnector` is created. This class uses an `OpenMediaSDKData` instance to read the configuration data from `OpenMediaSDK.properties` and connects to the servers, as described earlier.

```
connection = new SimpleConnector();
```

Create and Link to the GUI

At this point, the constructor calls `OpenMediaSdkGui`, which creates the graphical user interface. With the GUI components created, it is possible to link them to actions that affect QIL 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`:`

Finally, `SimpleService` adds its `Connection` attribute as a `WindowListener` to the `OpenMediaSdkGui` attribute. If the user closes the `OpenMediaSdkGui` window, this object calls the `windowClosing()` method of the `connection` filed that releases the QIL library and disconnects by releasing the `OMSDKConnector`. This is done when setting properties for the GUI main frame.

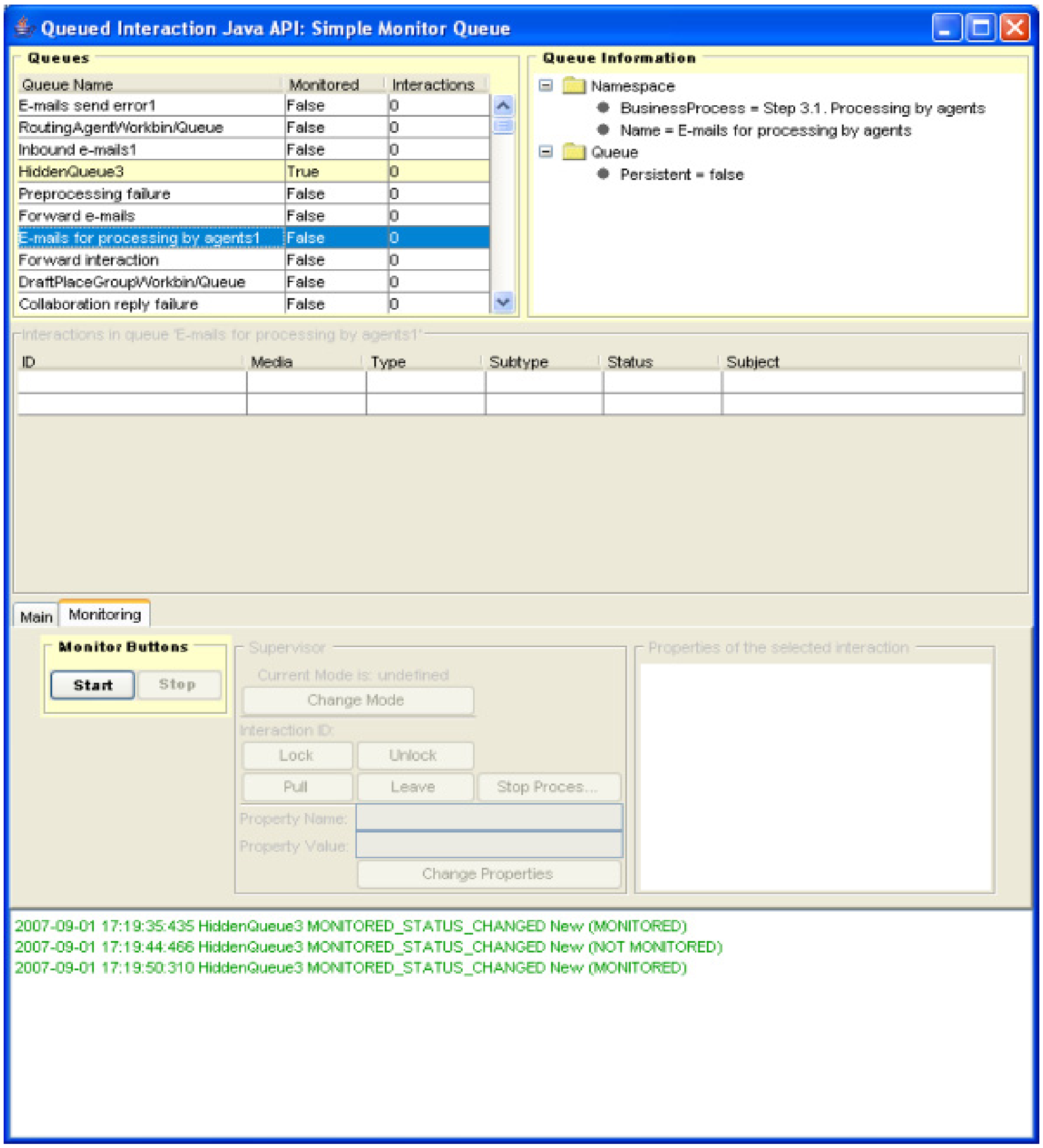
```
sdkGui.mainGuiWindow.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_
CLOSE);
sdkGui.mainGuiWindow.addWindowListener(connection);
sdkGui.mainGuiWindow.pack();
sdkGui.mainGuiWindow.setVisible(true);
```

About the User Interface

Now that you understand the basics of the `SimpleService` 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, only when a disconnection occurs. The user interface is designed to make it easy for you to track these messages by giving each type its own color. The `ServiceEvent` messages are blue and errors have red messages.

SimpleMonitorQueue

The `SimpleMonitorQueue` example extends the `SimpleService` example. It activates a GUI panel that displays a list of queues in real-time and lets you start and stop monitoring these queues. These tasks use `QILQueueMonitorStatusEvent` and `QILQueueContentChangedEvent` events for which you can select a trace level, as described in Chapter 2, “About the Code Examples,” on page 21 and shown again in Figure 7.



Queue Name	Monitored	Interactions
E-mails send error1	False	0
RoutingAgentWorkbinQueue	False	0
Inbound e-mails1	False	0
HiddenQueue3	True	0
Preprocessing failure	False	0
Forward e-mails	False	0
E-mails for processing by agents1	False	0
Forward interaction	False	0
DraftPlaceGroupWorkbinQueue	False	0
Collaboration reply failure	False	0

- Namespace
  - BusinessProcess = Step 3.1. Processing by agents
  - Name = E-mails for processing by agents
- Queue
  - Persistent = false

ID	Media	Type	Subtype	Status	Subject

```
2007-09-01 17:19:35:435 HiddenQueue3 MONITORED_STATUS_CHANGED New (MONITORED)
2007-09-01 17:19:44:466 HiddenQueue3 MONITORED_STATUS_CHANGED New (NOT MONITORED)
2007-09-01 17:19:50:310 HiddenQueue3 MONITORED_STATUS_CHANGED New (MONITORED)
```

Figure 7: Simple Monitor Queue

As shown in the above figure, the example lets the user select a queue in the `Queues` table and displays the queue's annex in the `Queue Information` tree. `Start` and `Stop` buttons update the user interface accordingly with the `Monitor` status displayed in the table. Monitor queues are highlighted in light yellow.

As you might expect, the application updates when the application receives a queue event. Now that you have an idea of what this example does, here is a description of how it carries out the steps in writing a QIL application.

Implement a Listener

`SimpleMonitorQueue` is a subclass of `SimpleService`. Because of this, it already implements the `ServiceListener` interface. In order to handle queue events, it implements the `QILQueueListener` interface. Here is the class declaration for `SimpleMonitorQueue`:

```
public class SimpleMonitorQueue extends SimpleService implements
QILQueueListener{
```

Set up Button Actions

Since `SimpleMonitorQueue` needs to use the `SimpleService` widgets, the first thing done by the `linkWidgetsToGui()` method is call the superclass's method:

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

Now `SimpleMonitorQueue` can get queue information to fill in the `Queues` table. To do that, it calls the `QILFactory.getAllQueues()` and adds a line to the table for each available queue:

```
Iterator itQueues = QILFactory.getQILFactory().getAllQueues().iterator();
while(itQueues.hasNext())
{
   QILQueue queue = (QILQueue) itQueues.next();
   sdkGui.setQueue(queue.getID(), queue.getStatus().toString(), queue.isMonitored());
   //...
}
```

Now `SimpleMonitorQueue` can link to the GUI buttons and add button actions to them. The code to carry out these actions is handled by methods which are copied/pasted from application blocks, as shown in these examples for the `Start` button:

```
startButton = sdkGui.startButton;
startButton.setAction(new AbstractAction("Start") {
   public void actionPerformed(ActionEvent actionEvent) {
      if(selectedQueueName != null)
      {
         startMonitoring(selectedQueueName);
      }
   }
});

/// Source from the StartMonitoringQueue Application Block
public void startMonitoring(String queueName) {
   try {
    QILFactory.getQILFactory().getQueue(queueName).startMonitoring();
  } catch (QILUninitializedException __e) {
    sdkGui.writeLogMessage("QILFactory is not initialized.+__e.toString(),
      OpenMediaSdkGui.errorStyle);
  } catch (QILRequestFailedException __e) {
    sdkGui.writeLogMessage("Connection to Configuration Server
    may be lost. "+__e.toString(), OpenMediaSdkGui.errorStyle);
  }
}
```

Register Your Application

The next step is to register your application so it can receive the events you will need to work with queues. You register the listener on each queue.

This is done by `linkWidgetsToGui()` when filling in the `Queues` table at startup. This method includes a copy of the `GetQueue` application block :

```
/* Source from the GetQueue Application Block, refer to the getAllQueues() method */
try{
  Collection queues = QILFactory.getQILFactory().getAllQueues();
  queues.iterator();
  for (Iterator iterator = queues.iterator(); iterator.hasNext();) {
    QILQueue queue = (QILQueue) iterator.next();
    sdkGui.setQueue(queue.getID(), queue.isMonitored(),new Integer(0).toString());
    ixnPerQueue.put(queue.getID(), new Integer(0));
    queue.addQueueListener(this);
  }
}
catch (QILRequestFailedException __e)
{
  sdkGui.writeLogMessage("Request failed. Connection to Configuration Server may be
   lost. "+__e.toString(), OpenMediaSdkGui.errorStyle);
}
```

After registering, if a queue changes, the QIL library gets a `QILQueueEvent` object and calls the `handleQueueEvent()` method implemented in this example.

Add Event-Handling Code

As Queued Interaction (Java API) is used for publishing events, you do not implement extended processing directly in this event handler: You do it in a thread. For more information, see “Threads and Listeners” on page 20.

`SimpleMonitorQueue` uses the `QueueEventThread` class to process `QILQueueEvent` in the `handleQueueEvent()` method, as shown here.

```
public void handleQueueEvent(QILQueueEvent event)
{
        QueueEventThread p = new QueueEventThread(event);
        p.start();
    }
```

`QILQueueEvent` events can be cast to `QILQueueMonitorStatusEvent` or `QILQueueContentChangedEvent`, as shown in the `MyQILQueueListener` application block. The example synchronizes the GUI by calling the `setQueueWidget()` method whenever it receives any `QILQueueMonitorStatusEvent` events. It also indicates changes in status by writing to the log panel, as shown in this code snippet:

```
////////////// Source from the MyQILQueueListener Application Block ////////////
public void run()
{
  if(event instanceof QILQueueMonitorStatusEvent)
  {
    //Getting the event
    QILQueueMonitorStatusEvent statusEvent = (QILQueueMonitorStatusEvent)event;
    QILQueue queue = statusEvent.getQueue();
    String queueName = queue.getID();
    //Updating GUI
    setQueueWidgets(queueName);

    //Creating a log message
    //...
  }
```

For further details about synchronizing with the user interface, see “Synchronize the User Interface” on page 45.

Most of this method’s code for handling `QILQueueContentChangedEvent` events is for writing messages to the log panel at the bottom of the `SimpleMonitorQueue` user interface.

The message built by this method lists all interaction activity in the queue, that is, all of the interactions that have been added and removed, as shown here:

```
  else if(event instanceof QILQueueContentChangedEvent)
  {
    QILQueueContentChangedEvent changeEvent = (QILQueueContentChangedEvent)event;
    QILQueue queue = changeEvent.getQueue();
    String detailedLogMessage = queue.getID().toString();
    try{
      //Getting Added Interactions
      Iterator addedIxns = changeEvent.getAddedInteractions().iterator();
      while (addedIxns.hasNext())
      {
        QILInteraction ixn = (QILInteraction) addedIxns.next();
        detailedLogMessage+="\nAdded "+ixn.getID();
      }
```

```
        //Getting Removed Interactions
        Iterator deletedIxns = changeEvent.getRemovedInteractions().iterator();
        while (deletedIxns.hasNext())
        {
          QILInteraction ixn = (QILInteraction) deletedIxns.next();
          detailedLogMessage+="\nRemoved "+ixn.getID();
        }
      sdkGui.writeLogMessage(detailedLogMessage,OpenMediaSdkGui.interactionEventStyle);
      }catch(Exception __e)
      {
        sdkGui.writeLogMessage("ContentChange: "+__e.getMessage(),
            OpenMediaSdkGui.errorStyle);
      }
    }
}
```

Synchronize the User Interface

The example uses the `setQueueWidgets()` methods to synchronize the user interface widgets with the queue events received.

This method uses the `QILQueue` interface methods to update the information displayed, that is:

- The queue's status in the `Queues` table.
- And, if the queue is selected in the `Queues` table:
  - The `Queue information` tree.
  - The `Start` and `Stop` buttons to be enabled or disabled.

Here is the code to enable or disable `Start` and `Stop` buttons:

```
//Updating the Queues table
sdkGui.setQueue(queue.getID(), queue.getStatus().toString(),
    queue.isMonitored());
//Updating the Annex tree and the buttons if needed
if(selectedQueue.getID().equals(queue.getID())){
  sdkGui.setAnnex(selectedQueue.getAnnex());
  if(selectedQueue.getStatus() == QILQueueStatus.ACTIVE)
  {
    if(selectedQueue.isMonitored()){
      startButton.setEnabled(false);
      stopButton.setEnabled(true);
    } else{
      startButton.setEnabled(true);
      stopButton.setEnabled(false);
    }
  }else {
    startButton.setEnabled(false);
    stopButton.setEnabled(false);
  }
}
```

Wrapping up

The `SimpleMonitorQueue` example includes a `SelectedQueueListener` class that implements `ListSelectionListener`. This listener enables you to handle line selection in a table by calling the `valueChanged()` method.

`SelectQueueListener` implements this method and makes a call to the `SimpleMonitorQueue.handleQueueSelection()` method. This method synchronizes the user interface with the current state of the selected queue, as shown here:

```
String queueName = sdkGui.getQueueNameAt(selectedRow);
try{
   selectedQueue = QILFactory.getQILFactory().getQueue(queueName);
   setQueueWidgets(selectedQueue);
}catch(Exception __e){
   sdkGui.writeLogMessage(__e.toString(),
      OpenMediaSdkGui.errorStyle);
}
```

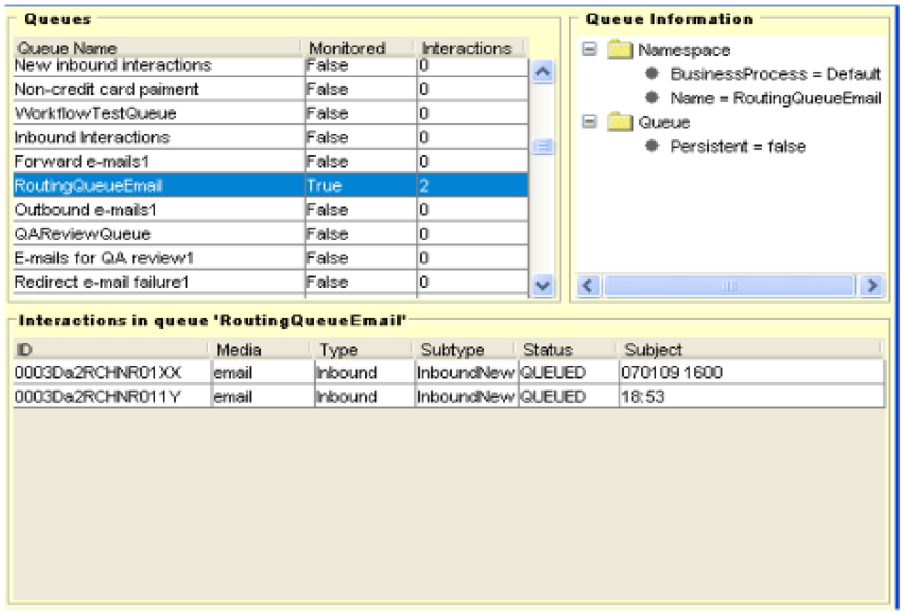
At startup, the `SimpleMonitorQueue()` constructor registers this listener on the `Queues` table by calling the `OpenMediaSdkGui.activateQueueRowSelection()` method.

```
public SimpleMonitorQueue(String windowTitle) {
   super(windowTitle);
   //Register a ListSelectionListener on the Queues table
   sdkGui.activateQueueRowSelection(
      new SelectedQueueListener(this));
}
```

SimpleMonitorInteraction

The `SimpleMonitorInteraction` example extends the `SimpleMonitorQueue` example and implements the `QILInteractionListener` interface. It activates a GUI panel that displays the interactions of a monitored queue in real-time.

These tasks use `QILQueueMonitorStatusEvent, QILQueueContentChangedEvent, and QILInteractionEvent` events to update the interactions' list for a selected queue, as shown in Figure 8.



Queues

Queue Name	Monitored	Interactions
New inbound interactions	False	0
Non-credit card paiment	False	0
WorkflowTestQueue	False	0
Inbound Interactions	False	0
Forward e-mails1	False	0
RoutingQueueEmail	True	2
Outbound e-mails1	False	0
QAReviewQueue	False	0
E-mails for QA review1	False	0
Redirect e-mail failure1	False	0

Queue Information

- Namespace
  - BusinessProcess = Default
  - Name = RoutingQueueEmail
- Queue
  - Persistent = false

Interactions in queue 'RoutingQueueEmail'

ID	Media	Type	Subtype	Status	Subject
0003Da2RCHNR01XX	email	Inbound	InboundNew	QUEUED	070109 1600
0003Da2RCHNR011Y	email	Inbound	InboundNew	QUEUED	18:53

Figure 8: SimpleMonitorInteraction Panels

As shown in the above figure, the example lets the user select a queue in the `Queues` table. If the queue's `Monitored` status is `True`, the `Interactions` table updates and displays interaction information for this queue.

As you might expect, the `Interactions` table updates when the application receives queue and interaction events. Now that you have an idea of what this example does, here is a description of how it carries out the steps in writing this QIL application.

Implement a Listener

`SimpleMonitorInteraction` is a subclass of `SimpleMonitorQueue`. Because of this, it already implements the `ServiceListener and the QILQueueListener` interface. In order to handle interaction events, it implements the `QILInteractionListener` interface. Here is the class declaration for `SimpleMonitorInteraction`:

```
public class SimpleMonitorInteraction extends SimpleMonitorQueue
implements QILInteractionListener {
```

Register Your Application

The next step is to register your application so it can receive the events you will need to work with. The super class `SimpleMonitorQueue` already registers the listener on each queue for managing queue events.

To register the `QILInteractionListener` handler for the interactions of a queue, you need to check whether your application is monitoring the queue. `SimpleMonitorInteraction` overloads the `QILQueueListener.handleQueueEvent()` method, to register and unregister for interactions on `QILQueueMonitorStatusEvent` events, as explained in the following section.

Add Event-Handling Code

Because `SimpleMonitorInteraction` inherits `SimpleMonitorQueue`, it already implements the `QILQueueListener.handleQueueEvent()` method. However, to handle interactions' listener registration and interaction widgets' updates, it overloads this method and manages the `QILQueueEvent` events in the `QueueEventForIxnThread` thread.

```
public void handleQueueEvent(QILQueueEvent event)
{
   super.handleQueueEvent(event);
   QueueEventForIxnThread p = new QueueEventForIxnThread(event,this);
   p.start();

}
```

`SimpleMonitorInteraction` takes into account `QILQueueMonitorStatusEvent` events to register for interactions, and `QILQueueContentChangedEvent` events to update the `Interactions` table, because they indicate changes in the queue content—that is, added and removed interactions. The `QueueEventForIxnThread` thread uses the `MyQILQueueListener` application block to process these queue events, as shown here:

```
class QueueEventForIxnThread extends Thread
{
   QILQueueEvent event;
   SimpleMonitorInteraction app;

   //..
   ////// Source from the MyQILQueueListener Application Block /////////////
   public void run()
   {
      if(event instanceof QILQueueMonitorStatusEvent)
      {
         //Getting the event
         QILQueueMonitorStatusEvent statusEvent = (QILQueueMonitorStatusEvent)event;
         QILQueue queue = statusEvent.getQueue();
         //Updating GUI
         if(statusEvent.getNewMonitorStatus() ==  QILQueueMonitorStatus.MONITORED)
         {
            queue.addInteractionListener(app);
         }
         else if(statusEvent.getNewMonitorStatus() ==
                     QILQueueMonitorStatus.NOT_MONITORED)
         {
            queue.removeInteractionListener(app);
            sdkGui.clearInteractions(queue.getID());
         }
      } else if(event instanceof QILQueueContentChangedEvent)
      {
         QILQueueContentChangedEvent changeEvent = (QILQueueContentChangedEvent)event;
         QILQueue queue = changeEvent.getQueue();
         setIxnWidgets(queue.getID(), changeEvent.getAddedInteractions(),
            changeEvent.getRemovedInteractions());
      }
   }
}
```

`SimpleMonitorInteraction` implements the `QILInteractionListener.handleInteractionEvent()` method to update the `Interactions` table on `QILInteractionStatusChangedEvent` and `QILInteractionPropertiesChangedEvent` events.

It uses the `InteractionEventThread` thread to process interaction events, with respect to the `MyQILInteractionListener` application block's guidelines.

```
class InteractionEventThread extends Thread
{
   QILInteractionEvent event;

   public InteractionEventThread(QILInteractionEvent _event)
   {
      event=_event;
   }
   ////Source from the MyQILInteractionListener Application Block /////////////
   public void run()
   {

      QILInteraction interaction = event.getInteraction();
      if (event instanceof QILInteractionStatusChangedEvent)
      {
         QILInteractionStatusChangedEvent eventStatusChanged =
            (QILInteractionStatusChangedEvent) event;
         QILInteraction ixn = eventStatusChanged.getInteraction();
         // Creating a log message
         //...
         /// updating the table
         sdkGui.setIxn(ixn.getQueue().getID(), ixn.getID(),
            ixn.getMediaType().toString(),ixn.getType().toString(),
            ixn.getSubtype().toString(),eventStatusChanged.getStatus().toString(),
            ixn.getProperties());
      } else if (event instanceof QILInteractionPropertiesChangedEvent) {
         QILInteractionPropertiesChangedEvent eventPropertiesChanged =
            (QILInteractionPropertiesChangedEvent) event;
         QILInteraction ixn = eventPropertiesChanged.getInteraction();
         ///.... write log message
         // updating the table
         sdkGui.setIxn(ixn.getQueue().getID(),ixn.getID(),
            ixn.getMediaType().toString(),ixn.getType().toString(),
            ixn.getSubtype().toString(),ixn.getStatus().toString(),
            ixn.getProperties());
      }
   }
}
```

Synchronize the User Interface

`SimpleMonitorInteraction` overloads the `setQueueWidgets()` method to synchronize the `Interactions in queue` table with the queue events received.

```
public void setQueueWidgets(String queueName)
{
   super.setQueueWidgets(queueName);
   //Updating the Interaction table
   if(queueName.equals(selectedQueueName)){
      this.sdkGui.switchIxnTable(queueName);
   }
}
```

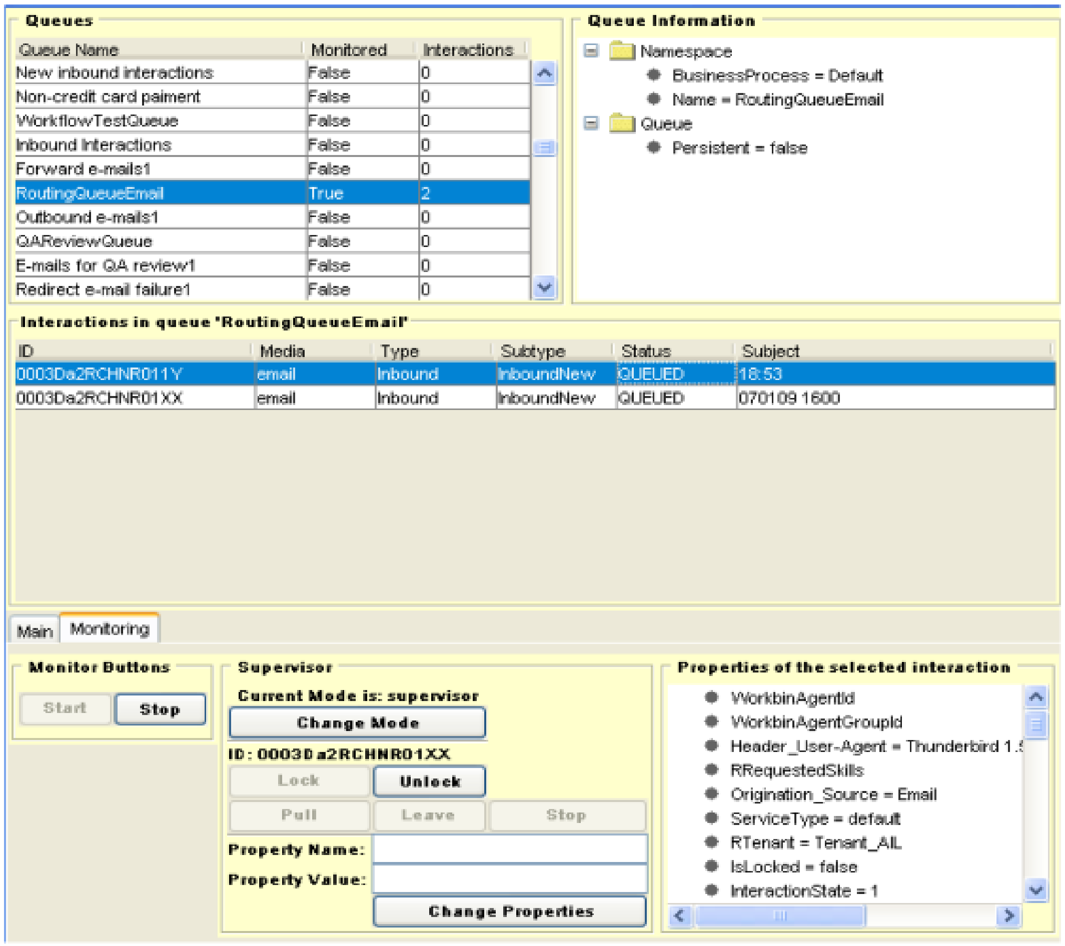
`SimpleMonitorInteraction` also includes the `setIxnWidgets()` method which adds interactions to or remove interactions from the `Interactions in queue` table.

```
public void setIxnWidgets(String queueName, Collection added, Collection removed){

   Iterator addedIxns = added.iterator();
   //Getting Added Interactions
   while (addedIxns.hasNext())
   {
      QILInteraction ixn = (QILInteraction) addedIxns.next();
      sdkGui.setIxn(queueName,ixn.getID(), ixn.getMediaType().toString(),
         ixn.getType().toString(), ixn.getSubtype().toString(),
         ixn.getStatus().toString(), ixn.getProperties());
   }

   //Getting Removed Interactions
   Iterator deletedIxns = removed.iterator();
   while (deletedIxns.hasNext())
   {
      QILInteraction ixn = (QILInteraction) deletedIxns.next();
      sdkGui.removeIxn(queueName, ixn.getID());
```

```
    }
}
```

SimpleSupervisor

The `SimpleSupervisor` example extends the `SimpleMonitorInteraction` example to provide Ad'Hoc features. When the user selects an interaction, `SimpleSupervisor` displays its properties in the tree `Properties of the selected interaction`, and enables Ad'Hoc buttons to perform actions such as `Lock`, `Pull`, and `Change properties` on the selected interaction, as shown in Figure 9.



Queue Name	Monitored	Interactions
New inbound interactions	False	0
Non-credit card paiment	False	0
WorkflowTestQueue	False	0
Inbound Interactions	False	0
Forward e-mails1	False	0
RoutingQueueEmail	True	2
Outbound e-mails1	False	0
QAReviewQueue	False	0
E-mails for QA review1	False	0
Redirect e-mail failure1	False	0

Namespace
- BusinessProcess = Default
- Name = RoutingQueueEmail

Queue
- Persistent = false

Interactions in queue 'RoutingQueueEmail'

ID	Media	Type	Subtype	Status	Subject
0003Da2RCHNR011Y	email	Inbound	InboundNew	QUEUED	18:53
0003Da2RCHNR01XX	email	Inbound	InboundNew	QUEUED	070109 1600

Current Mode is: supervisor

ID: 0003Da2RCHNR01XX

Property Name:

Property Value:

- WorkbinAgentId
- WorkbinAgentGroupId
- Header_User-Agent = Thunderbird 1.
- RRequestedSkills
- Origination_Source = Email
- ServiceType = default
- RTenant = Tenant_AIL
- IsLocked = false
- InteractionState = 1

Figure 9: SimpleSupervisor Panels

To get more information about the features associated with supervisor buttons, refer to "Ad'Hoc Management" on page 33.

Implement a Listener

SimpleSupervisor extends SimpleMonitorInteraction and implements no additional listener. By inheritance, SimpleSupervisor implements:

- QILServiceListener
- QILQueueListener
- QILInteractionListener

Set up Button Actions

SimpleSupervisor uses the AdHocManagement application block as a regular class to implement the buttons of the Supervisor panel.

The AdHocManagement application block is declared as a private member of SimpleSupervisor and it is instantiated in the constructor at the application's startup:

```
AdHocManagement adHocManager;
public SimpleSupervisor(String windowTitle) {
   super(windowTitle);
   //...
   adHocManager = new AdHocManagement() ;
   //...
}
```

Then, the linkWidgetsToGui() method makes call to the AdHocManagement instance to provide button actions, as shown for the button in the following code snippet:

```
lockButton.setAction(new AbstractAction("Lock") {
public void actionPerformed(ActionEvent actionEvent) {
   if(selectedInteractionId!=null)
   {
      try {
         adHocManager.lock(selectedQueueName,
               selectedInteractionId,"","");
         lockButton.setEnabled(false);
         unlockButton.setEnabled(true);
         pullButton.setEnabled(false);
         stopProcessingButton.setEnabled(false);
         managing = true;
      } catch (QILOperationalModeRestrictionException e) {
         sdkGui.writeLogMessage("Lock requested, not authorized: "
               +e.getMessage(), OpenMediaSdkGui.errorStyle);
      } catch (QILRequestFailedException e) {
         sdkGui.writeLogMessage("Lock requested, failed: "
               +e.getMessage(), OpenMediaSdkGui.errorStyle);
      }
   }
}});
```

As you can see, you have to implement buttons' logic to enable and disable them accordingly.

To get more information about the features associated with supervisor buttons, refer to "Ad'Hoc Management" on page 33.

Register Your Application

The super class `SimpleMonitorQueue` and `SimpleMonitorInteraction` already register the listener on each queue for managing queue and interaction events. `SimpleSupervisor` makes no additional registration.

Add Event-Handling Code

Because `SimpleSupervisor` inherits `SimpleMonitorInteraction`, it already implements the `QILQueueListener.handleQueueEvent()` and `QILInteractionListener.handleInteractionEvent()` methods.

However, to update supervisor widgets on events, it overloads these methods and implements two threads:

- The `RemovedInteractionThread` thread, run in the `handleQueueEvent()` method, removes interaction properties stored to manage the tree `Properties of the selected interaction` on queue content changes' events.
- The `InteractionPropertiesChangedThread` thread, run in the `handleInteractionEvent()` method, updates the tree `Properties of the selected interaction`, on interaction property changes' events.

Wrapping up

`SimpleSupervisor` includes a `SelectedInteractionListener` class that implements `ListSelectionListener`. This listener enables you to handle line selection in a table by calling the `valueChanged()` method.

`SelectedInteractionListener` implements this method and makes a call to the `SimpleSupervisor.handleInteractionSelectionAt()` method. This method synchronizes the `Supervisor` panel and the tree `Properties of the selected interaction`, as shown here:

```
try{
   this.selectedInteractionId =
      this.sdkGui.getInteractionNameAt(this.selectedQueueName,
         selectedRow);
   if(this.selectedInteractionId != null)
   {
      interactionIDLabel.setText("ID: "+selectedInteractionId);
      sdkGui.switchPropertyTree(selectedQueueName,
        selectedInteractionId);
    }
}catch(Exception __e)
{
    sdkGui.writeLogMessage("Interaction selection failed. "
      +__e.getMessage(), OpenMediaSdkGui.errorStyle);
}
```

At startup, the `SimpleSupervisor()` constructor registers this listener on the `Interactions in queue` table by calling an `OpenMediaSdkGui` method.

```
public SimpleSupervisor(String windowTitle) {
   super(windowTitle);
   adHocManager = new AdHocManagement() ;
   sdkGui.activateIxnRowSelection(
         new SelectedInteractionListener(this));
}
```

Chapter

4 Alarm Examples

This chapter explains two examples, `SimpleQueueAlarm.java` and `MultipleAlarm.java`. These code examples use the Queued Interaction (Java API) to notify users with queue activity, and to display alarms according to the queues' threshold.

This chapter comprises the following sections:

Introduction

To follow the discussion in this chapter, you will need the *Queued Interaction SDK 7.6 Java API Reference*, which is located in the `doc/` subdirectory under the Queued Interaction (Java API) product installation directory, and the source code for the `SimpleQueueAlarm.java` and `MultipleAlarm.java` examples. Refer to the discussion in Chapter 2, "About the Code Examples," on page 23 for more information on how to use the examples.

SimpleQueueAlarm

The `SimpleQueueAlarm` example provides a stand-alone application that connects to Genesys servers and monitors a queue. When a maximum number of interactions occurs on the monitored queue, this example sends an alarm.

This application has no GUI. So fewer steps are required to make this application work. This example has been designed to make its functional 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 `SimpleQueueAlarm` example.

Implement a Listener

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

```
public class SimpleQueueAlarm implements QILQueueListener {
```

`SimpleQueueAlarm` uses `QILQueueListener` because it can handle the `QILQueueContentChangedEvent` events that this example uses to monitor the interactions' activity in the queue.

Connect to Servers

As explained in the previous chapter, the example uses the `SimpleConnector` class to establish the all-important connection with the Genesys servers and initialize the `OMSDKConnector` with the `ConnectorQIL` application block. For more information on how this is done, you can refer to "Open Media Commons" on page 28. For the purposes of this example, here is all you need to do in the constructor:

```
SimpleConnector connector = new SimpleConnector();
```

Register Your Application

The next step is to register your application so it can receive the events you will need to work with this queue. This is done at the application's startup, in the `SimpleQueueAlarm()` constructor. It retrieves the `QILQueue` instance associated with the name of the queue to be monitored. Then, it registers as the listener on this queue. Finally, it starts monitoring the queue, as shown in the following code snippet:

```
QILQueue queue = QILFactory.getQILFactory().getQueue(queueName);
queue.addQueueListener(this);
queue.startMonitoring();
```

After monitoring begins, if the queue changes, the QIL library gets a `QILQueueEvent` object and calls the `handleQueueEvent()` method implemented in this example.

Add Event-Handling Code

Classes implementing the `QILQueueListener` interface must include the `handleQueueEvent()` method.

Most of the code in this method handles the alarm that must be sent if the number of interactions is greater than the alarm value. `SimpleQueueAlarm` takes into account `QILQueueContentChangedEvent` events only, because they indicate changes in the queue content—that is, added and removed interactions.

```
QILQueueContentChangedEvent changeEvent = (QILQueueContentChangedEvent)event;
nbInteractions += changeEvent.getAddedInteractions().size()
             - changeEvent.getRemovedInteractions().size();
```

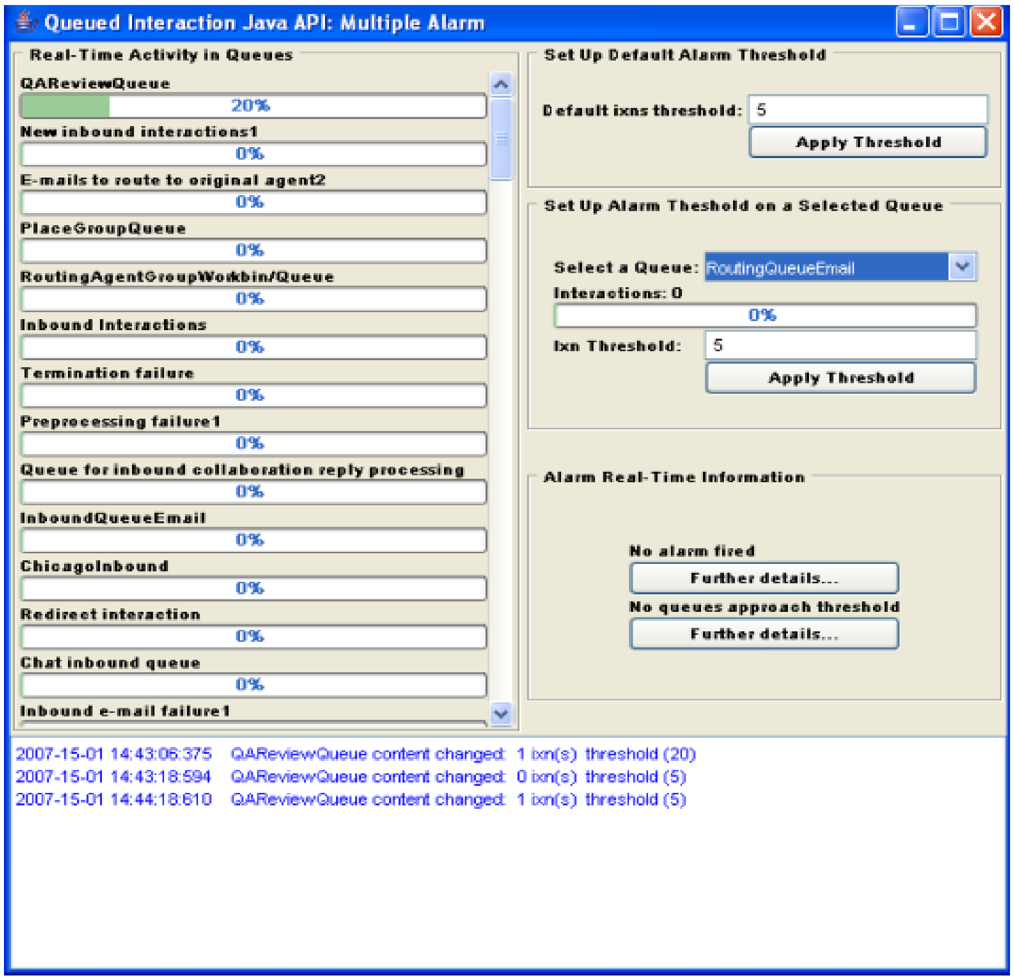
Then it tests whether or not an alarm should be sent, as shown here:

```
if(nbInteractions > alarm)
  System.out.println(createTimeStamp()+ " !!!!!!!! Alarm !!!!!!!!!!!
             More than "+alarm+" ixns in queue!");
```

Note: If you need to perform extended processing when your applications receive events, do not implement this processing directly in your event handler—do it in a thread. For more information, see "Threads and Listeners" on page 20.

MultipleAlarm

The `MultipleAlarm` code example provides a GUI-based desktop application that displays queues activity in real-time. For each queue, a process bar monitors interactions and indicates the queue's activity according to an interaction threshold that the user can modify. As shown in Figure 10, the process bar moves up in the `Real-time Activity in Queues` list according to its activity.


Queued Interaction Java API: Multiple Alarm
Real-Time Activity in Queues
Set Up Default Alarm Threshold
Apply Threshold
Set Up Alarm Theshold on a Selected Queue
Apply Threshold
Alarm Real-Time Information
Further details...
Further details...

QAReviewQueue 20%
New inbound interactions1 0%
E-mails to route to original agent2 0%
PlaceGroupQueue 0%
RoutingAgentGroupWorkbin/Queue 0%
Inbound Interactions 0%
Termination failure 0%
Preprocessing failure1 0%
Queue for inbound collaboration reply processing 0%
InboundQueueEmail 0%
ChicagoInbound 0%
Redirect interaction 0%
Chat inbound queue 0%
Inbound e-mail failure1

Default ixns threshold: 5

Select a Queue: RoutingQueueEmail
Interactions: 0
0%
Ixn Threshold: 5

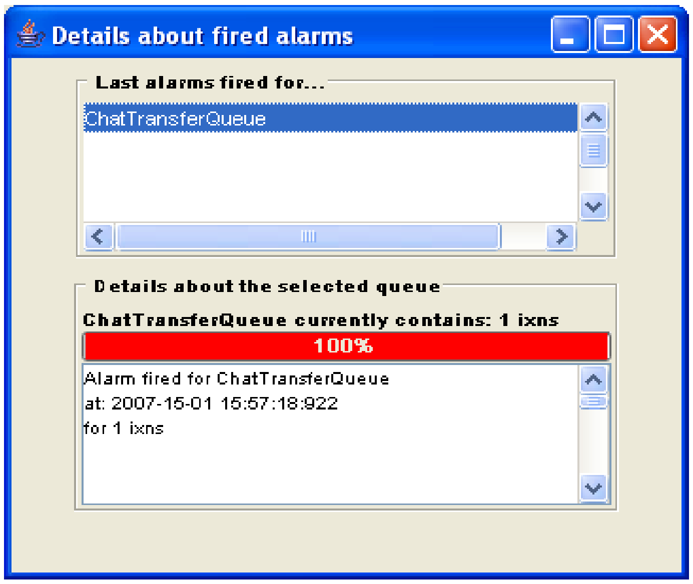
No alarm fired
No queues approach threshold

```
2007-15-01 14:43:06:375   QAReviewQueue content changed:  1 ixn(s)  threshold (20)
2007-15-01 14:43:18:594   QAReviewQueue content changed:  0 ixn(s)  threshold (5)
2007-15-01 14:44:18:610   QAReviewQueue content changed:  1 ixn(s)  threshold (5)
```

Figure 10: MultipleAlarm at Application's Startup

Two panels on the right upper corner enable the user to set a new interaction threshold for all queues or for a particular queue. If a queue reaches the interaction threshold, an alarm is fired in the log panel.

The `Alarm Real-Time Information` panel includes two buttons that open dialog boxes used as history components. The first dialog box lists all the queues for which alarms were fired, as shown in Figure 11, and the second dialog box shows queues approaching their interaction threshold.


Details about fired alarms
Last alarms fired for...
ChatTransferQueue
Details about the selected queue
ChatTransferQueue currently contains: 1 ixns
100%

Alarm fired for ChatTransferQueue
at: 2007-15-01 15:57:18:922
for 1 ixns

Figure 11: Details about Fired Alarms

If the user selects a queue in this list, the panel below displays the current process bar for this queue and the list of fired alarms.

`MultipleAlarm` registers for `QILQueueEvent` events to count interactions in queues. Then, most of the code example is a matter of updating counters and displaying this information for the user. To display information, this code example uses the `AlarmGui` class which separately manages GUI layouts.

Implement a Listener

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

```
public class SimpleQueueAlarm implements QILQueueListener {
```

As `SimpleQueueAlarm`, `MultipleAlarm` uses `QILQueueListener` because it can handle the `QILQueueContentChangedEvent` events that this example uses to monitor the interactions' activity in monitored queues.

Connect to Servers

As explained in the previous chapter, the example uses the `SimpleConnector` class to establish the all-important connection with the Genesys servers and initialize the `OMSDKConnector` with the `ConnectorQIL` application block. For more information on how this is done, you can refer to "Open Media Commons" on page 28. For the purposes of this example, here is all you need to do in the constructor:

```
SimpleConnector connector = new SimpleConnector();
```

Register Your Application

The next step is to register your application so it can receive the events you will need to work with queues. This is done at the application's startup by calling the `linkWidgetsToGui()` method in the `MultipleAlarm()` constructor.

This method uses and extends the source code of the `GetQueue` application block to retrieve all the available `QILQueue` instances, register for their `QILQueueEvent` events, then start monitoring them, as shown in the code snippet below:

```
public void linkWidgetsToGui()
{
   queueInfo= new HashMap();
   /* Source from the GetQueue Application Block, refer to the getAllQueues() method */
   try{
      Collection queues = QILFactory.getQILFactory().getAllQueues();
      queues.iterator();
      for (Iterator iterator = queues.iterator(); iterator.hasNext();) {
         QILQueue queue = (QILQueue) iterator.next();
         queueInfo.put (queue.getID(), new Integer[]{new Integer(0), new Integer(20)} );
         gui.setQueueInfo(queue.getID(), 0,0 );
         queue.addQueueListener(this);
         queue.startMonitoring();
      }
   }
   catch (QILRequestFailedException __e)
   {
      gui.writeLogMessage("Request failed. Connection to Configuration Server may be
    lost. "+__e.toString(), AlarmGui.errorStyle);
   }
   catch (QILUninitializedException e) {
      //  Exception thrown if QIL Factory was not initialized
      gui.writeLogMessage("Request failed. QILFactory is not initialized.
   "+e.toString(), AlarmGui.errorStyle);
   }
}
```

For each queue retrieved, `MultipleAlarm` stores the queue's information in the `queueInfo` map and displays this information in the GUI by calling the `AlarmGui.setQueueInfo()` method.

After monitoring begins, if one of the queues changes, the QIL library creates a `QILQueueEvent` object and calls the `handleQueueEvent()` method implemented in this example.

Add Event-Handling Code

As Queued Interaction (Java API) is used for publishing events, you do not implement extended processing directly in this event handler: You do it in a thread. For more information, see “Threads and Listeners” on page 18.

`MultipleAlarm` uses the `QueueEventThread` class to process `ServiceEvent`, as shown here.

```
public void handleQueueEvent(QILQueueEvent event) {
   QueueEventThread p = new QueueEventThread(event);
   p.start();
}
```

Processing `QueueEvents` is performed in the `QueueEventThread.run()` method. This method handles `QILQueueContentChangedEvent` events to update the number of interactions in the queue associated with the event and calculates the queue’s activity according to the queue’s interaction threshold (available in the `queueInfo` map), as shown here.

```
if(event instanceof QILQueueContentChangedEvent)
{
   QILQueueContentChangedEvent changeEvent = (QILQueueContentChangedEvent)event;
   String queueName = event.getQueue().getID();
   Integer[] infoIxns = (Integer[]) queueInfo.get(queueName);
   int nbInteractions = infoIxns[0].intValue() +
           changeEvent.getAddedInteractions().size()
           - changeEvent.getRemovedInteractions().size();
   int threshold = infoIxns[1].intValue();
   infoIxns[0]= new Integer(nbInteractions);
   queueInfo.put(queueName,infoIxns );
   int activity = getActivity(queueName);
   gui.writeLogMessage("   "+ queueName+ " content changed:  "
              + nbInteractions +" ixn(s) "+ " threshold (" + threshold + ")",
   AlarmGui.queueEventStyle);
   gui.setQueueInfo(queueName, nbInteractions, activity);
   if(activity >= 100)
      gui.writeLogMessage(" !!! Alarm on "+ queueName+ ": "+ nbInteractions +" ixns for
   a threshold of "+threshold +" ixns !!!", AlarmGui.errorStyle);
}
```

All the GUI synchronization is done there by calling the `AlarmGui.setQueueInfo()` method which refreshes all the components with the tuple (queue name, number of interactions, activity in percents).

Wrapping up

`MultipleAlarm` implements the `ItemListener` interface to listen user selection in the `Select a queue` combo box. If the user selects a queue, `MultipleAlarm` gets an `ItemEvent` event through the `itemStateChanged()` method. This method retrieves information from the `queueInfo` map to update the `Set up threshold` panel. Then, the user is able to modify the interaction threshold for this queue.

```
public void itemStateChanged(ItemEvent evt) {
   if (evt.getStateChange() == ItemEvent.SELECTED) {
      String queueName = (String) evt.getItem();
      Integer[] infoIxns = (Integer[]) queueInfo.get(queueName);
      int nbInteractions = infoIxns[0].intValue() ;
      gui.setSelectedQueueDetails( queueName, nbInteractions,
         getActivity(queueName));
      this.thresholdField.setText(infoIxns[1].toString());
   }
}
```

`MultipleAlarm also` includes two `ListSelectionListener` classes used for listening to user selection in the dialog boxes providing details about fired alarms and queues approaching their threshold.

`AlarmListSelectionListener` and `WarningListSelectionListener` behave identically. These listeners enable you to handle line selection in a table by calling the `valueChanged()` method. This method retrieves the name of the selected queue, gets the corresponding information in the `queueInfo` map, and finally updates the dialog box by displaying details in the accurate panel.

```
public void valueChanged(ListSelectionEvent evt) {
   // When the user release the mouse button and completes the
selection,
   // getValueIsAdjusting() becomes false
   if (!evt.getValueIsAdjusting()) {
      JList list = (JList)evt.getSource();
      // Get all selected items
      Object[] selected = list.getSelectedValues();
      // Iterate all selected items
      for (int i=0; i<selected.length; i++) {
         String sel = (String) selected[i];
         Integer[] info = (Integer[]) queueInfo.get(sel);
         gui.setDetailsAboutSelectedAlarm(sel, info[0].intValue(),
getActivity(sel));
      }
   }
}
```

At startup, the `MultipleAlarm()` constructor creates a listener for each dialog box, as shown here:

```
gui.alarmListener = new AlarmListSelectionListener();
gui.warningListener = new WarningListSelectionListener();
```

Index