Cincom

VisualWorks®

MQ-Interface Guide

P46-0148-00

Part Number: P46-0148-00

Software Release 7.4

This document is subject to change without notice.

RESTRICTED RIGHTS LEGEND:

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Trademark acknowledgments:

CINCOM, CINCOM SYSTEMS, the Cincom logo and Cincom Smalltalk logo are registered trademarks of Cincom Systems, Inc. ParcPlace and VisualWorks are trademarks of Cincom Systems, Inc., its subsidiaries, or successors and are registered in the United States and other countries. ObjectLens, ObjectSupport, Cincom Smalltalk, Database Connect, DLL & C Connect, COM Connect, and StORE are trademarks of Cincom Systems, Inc., its subsidiaries, or successors. WebSphere and MQSeries are registered trademarks of IBM. All other products or services mentioned herein are trademarks of their respective companies. Specifications subject to change without notice.

The following copyright notices apply to software that accompanies this documentation:

VisualWorks is furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No class names, hierarchies, or protocols may be copied for implementation in other systems.

This manual set and online system documentation © 2005 by Cincom Systems, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent from Cincom.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book

Overview

This document describes the usage of the interface for WebSphere® MQ using the bshared libraries provided by IBM. WebSphere MQ is a message base communication system where two or more application can exchange messages through a queue. It is like electronic mail between applications.

The MQ-Interface provides an easy-to-use interface for VisualWorks to connect to the WebSphere MQ shared libraries. It simplifies the tedious handling of the C-call interface and its parameter. A developer can use WebSphere MQ through a simple and efficient feature wrapper. This document describes the feature wrapper, its classes and methods, and how to use it.

Audience

This guide assumes that you already know VisualWorks and Smalltalk, as well as a familiarity with WebSphere MQ.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.

Example	Description
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File ⟶ New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help → About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help → About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincom.com/smalltalk/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select one of the help options.

News Groups

The Smalltalk community is actively present on the internet, and willing to offer helpful advice. A common meeting place is the comp.lang.smalltalk news group. Discussion of VisualWorks and solutions to programming issues are common.

VisualWorks Wiki

A wiki server for VisualWorks is running and can be accessed at:

http://brain.cs.uiuc.edu:8080/VisualWorks.1

This is becoming an active place for exchanges of information about VisualWorks. You can ask questions and, in most cases, get a reply in a couple of days.

Commercial Publications

Smalltalk in general, and VisualWorks in particular, is supported by a large library of documents published by major publishing houses. Check your favorite technical bookstore or online book seller.

1

Using MQ-Interface

This chapter discusses the public classes and methods provided in the MQ-Interface library and ther use in building an application.

Public Classes

The following are the classes in the VisualWorks MQ-Interface that are intended for use by an application programmer to access the functionality of WebSphere MQ.

QueueManager
An abstract class that represents a WebSphere MQ queue manager. It instantiate a concrete class based on parameter an application provides in the instance creation methods

LocalQueueManager
A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on the same machine as the image. It is using the WebSphere MQ none threaded server library

LocalThapiQueueManager
A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on the same machine as the image. It is using the WebSphere MQ threaded server library

RemoteQueueManager
A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on a different machine using the WebSphere MQ none threaded client library

RemoteThapiQueueManager
A concrete queue manager class that connects the VisualWorks image to a queue manager that is installed on a different machine using the WebSphere MQ threaded client library

MessageQueue
An abstract class that represents the link to a queue in a queue manager. It links to a queue by opening it. In WebSphere MQ a queue can only be opened for writing or for reading. It is not possible to open a queue for both.

ReceiverQueue
A concrete message queue class that is used for reading messages from a queue.

SenderQueue
A concrete message queue class that is used for writing messages to a queue.

MQMessage
An abstract class for the four different message types IBM defined for WebSphere MQ.

ActionMessage
An abstract class for messages that is indented to trigger an action in the receiver of the message.

Request
A concrete message class for a request. You should use this message type if the receiver has to create a reply.

AsynchronousMessage
A concrete message class for a message that simply carry some information to another application. It does not generate a reply.

Reply
A concrete message class for a reply an application has to generate when receiving a request.

Report
A concrete message class for reports generated by WebSphere MQ itself or by an application about the state of a message or an error.

ActionDecorator
An abstract class to parameterize the storage of the action selector.

DefaultActionDecorator
A concrete action decorator class that combines the action selector with the application data.

Program Flow

An application should connect to the queue manager when it initializes its external resources. This is usually when the application is starting. Now the application is able to use the services provided by WebSphere MQ. It should disconnect from the queue manager when the application discards its external resources. This is usually when the application is terminated.

An application has to open a queue located in a queue manager before it can send or receive messages. It should close the queue if it does not need it anymore. An application may send several messages to an open queue or receive several messages from an open queue. A typical live cycle in the MQ-Interface can follow the sequence below:

```
Connect to a queue manager
    Open a receiver queue
    Open a sender queue
        Send messages to the sender queue
        Receive messages from the receiver queue
    Close the sender queue
    Close the receiver queue
Disconnect from the queue manager
```

You may skip the closing of queues since the queue manager will handle the live cycle of its queues.

Queue Mangers

A queue manager is a kind of provider for WebSphere MQ services. An application connects to a queue manager in order to use its services. An application must be connected to a queue manager before it can utilize its service. Therefore an application should connect to the queue manager when it initializes its external resources.

A queue manager can be installed on the same machine as the image (local queue manager) or on another machine (remote queue manager). For both setups IBM provide a different shared library (DLL on Windows). Additionally VisualWorks needs a different syntax for threaded and none threaded API calls.

Therefore the MQ-Interface contains a four concrete queue manager classes:

LocalQueueManager	local queue manager	none threaded
LocalThapiQueueManager	local queue manager	threaded
RemoteQueueManager	remote queue manager	none threaded
RemoteThapiQueueManager	remote queue manager	threaded

An application connects to a queue manager by using the instance creation method new: *aSymbol* threaded: *aBoolean* of the class QueueManager. The method is using the parameter to determine the concrete sub class. The first parameter defines the location of the queue manager: #local for a local queue manager and #remote for a remote queue manager. The second parameter defines if we use threaded (true) or none threaded (false) API.

Additionally a queue manager serves as container for its queues. The QueueManager instance keeps a list of all queues that are created in its focus. This allows the queue manager to maintain the live cycle of its queues. Each queue belongs to one queue manager but an application may send messages to queues in different queue managers.

Local Queue Manager

An instance of LocalQueueManager or LocalThapiQueueManager represents a queue manager that is installed on the same machine as the VisualWorks image. For this setup we only need to know the name of the queue manager.

```
| manager |
manager := QueueManager new: #local threaded: false.
manager name: 'venus.queue.manager'.
manager connect.
```

The above code fragment connects to a local queue manager using none threaded API. This queue manager is represented by the class LocalQueueManager. This will be a bit faster then using the treaded API but it will bock the whole image when calling a procedure in the library. You should only use a none threaded manager for a client that requests services from a server and that will actively waiting for replies.

```
| manager |
manager := QueueManager new: #local threaded: true.
manager name: 'venus.queue.manager'.
manager connect.
```

By simply changing the second parameter of the instance creation method new:threaded: you can create a local queue manager using threaded API. This queue manager is represented by the class LocalThapiQueueManager. Every API call will only block the actual process and not the whole image. But there is a small overhead for the threaded calls. They are slower.

If you don't need to exchange information anymore (i.e. you terminate the application) you send the message disconnect to the queue manager instance. It will take care of any open queue. You don't have to close any queue individually.

A queue manager has a registry where it keeps track of all its message queues. When an application requests a message queue the first time it will create a new message queue instance and register it. When the application requests an already registered queue it just returns the registered queue.

Remote queue manager

An instance of RemoteQueueManager or RemoteThapiQueueManager represents a queue manager that is installed another machine as the VisualWorks image. The image connects to the queue manager through a channel. Additionally to the name of the manager we have to know:

- the machine where the queue manager is installed,
- the port of the remote queue manager listener. An application has to provide this information if there is more then one queue manager installed on the host. If there is only one queue manager on the host then this setting can be omitted.
- the name of the channel that connect the machine where the VisualWorks image is installed to the queue manager

The instance creation is similar to the instance creation of a local queue manager but you have to provide the connection parameter.

```
| manager |
manager := QueueManager new: #remote threaded: false.
manager name: 'venus.queue.manager'.
manager host: 'Fuji1'.
manager port: 1423.
manager channel: 'cei.snk99.srvconn'.
manager connect.
```

The above code fragment connects to a remote queue manager using none threaded API. This queue manager is represented by the class RemoteQueueManager. Remember in none threaded libraries every API call will block the whole image.

```
| manager |
manager := QueueManager new: # remote threaded: true.
manager name: 'venus.queue.manager'.
manager host: 'Fuji1'.
manager port: 1423.
manager channel: 'cei.snk99.srvconn'.
manager connect.
```

By simply changing the second parameter of the instance creation method new:threaded: you can create a remote queue manager using threaded API. This queue manager is represented by the class RemoteThapiQueueManager. Every API call will only block the actual process and not the whole image. But there is a small overhead for the threaded calls. They are slower.

Message queues

A MessageQueue instance represents a queue that is or will be opened for either reading information from it or writing information to it. A message queue is identified by its name. When creating a new MessageQueue object it will not automatically opens the real queue in the queue manager. The queue will be opened when the first get or put message is sent to the MessageQueue instance.

This enables an application developer to initialize all needed queues at start time without caring of external resources. The developer just has to create instances for all necessary queues. The instances itself open the queue if needed.

An application may define a time frame for keeping a queue open. If no get: or put: message is sent to the queue within this time frame, then the queue instance will close the queue. The instance is now in the state it was just after it was created. By default this time frame is infinite.

A queue cannot be opened for both reading and writing information. You have to define what you want to do with the queue. The functionality for reading information and for writing information is separated into two subclasses.

A queue can be closed by sending the message close to it. It will also be closed if the queue manager that contains the queue is terminated.

Receiver queue

A receiver queue is used to read messages from a queue. You create a ReceiverQueue instance by sending the message getReceiverQueue: to a QueueManager instance.

```
| receiver |
receiver := queueManager getReceiverQueue: 'armor.1.reply'.
receiver waitInterval: 60000.
```

Now you can receive messages from the queue. In the above code fragment we create a receiver queue with the name "armor.1.reply". We set the default wait interval to 60000 milliseconds or 60 seconds. This default setting is later used for receiving messages. The queue will wait for 60 seconds after you sent a get message to the receiver queue. If no message arrives in time the queue raise an exception. An application may override this setting every time it wants to receive a message.

Normally when retrieving a message the queue will return the first message in the queue. Using match parameter a developer may select specific messages from the queue. These are the message parameter message id, the correlation id, group id, sequence number, offset and token. The MQ-Interface allows several setups to define match parameter. Using the class method defaultMatchingParameter: a developer can define match parameter for all queues. This setting can be overridden in a queue by defining individual parameter for a queue using the instance method matchingParameter:.

When issuing a GET a developer can define special match parameter for a single GET.

Sender queue

A sender queue is used to send messages to a queue and with this to another application. You create a SenderQueue instance by sending the message getSenderQueue: to a QueueManager instance.

```
| sender |
sender:= queueManager getSenderQueue: 'armor.1.ent.p2p'.
sender setReceiverQueue: receiver.
```

The above code fragment creates a new SenderQueue instance. This sender queue can now be used to send messages to another application.

The second statement links a receiver queue to the sender queue. When sending a request or an asynchronous message, the name of the receiver queue and the name of its queue manager will be copied to the message. The receiving application needs this information to send a reply or report back to the sender application.

An application may use this convenience feature to simplify the message handling. We don't force a developer to use so. If you don't want to use this feature, you have to provide the queue and queue manager name for requests and asynchronous messages. If you use this feature you can override the queue and queue manager name for any message you send through sender queue.

Messages

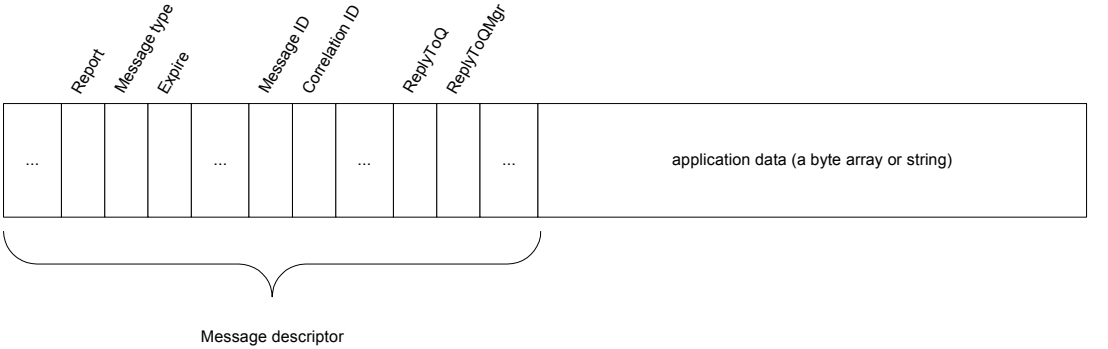
Messages represent the information, applications exchange via the message queues. Messages only carry raw data. An application itself is responsible for marshalling objects to a byte array and to demarshalling a byte array to an object. This is not part of the MQ-Interface.

IBM defines four types of messages:

- Datagram - A message where the application does not expect a reply. In the MQ-interface this message type is name asynchronous message since datagram normally has a different meaning.
- Request - A message where the application expects a reply.
- Reply - A message that is a reply to a request.
- Report - A status or error message.

The MQ-Interface provides a class for each message type.

A message contains a description that defines how a message has to be handled by WebSphere MQ and the receiving application.



The MQ-Interface preset the parameter according to the use of the message and the settings of the queue. A developer may set some of the parameters to change the default settings and settings copied from the queue:.

Report

By default this parameter is not set. A developer can request a status or an error report by defining the type of the report. The MQ-Interface provides a set of methods to define the reports to be sent back to the sender queue. This parameter is only valid for requests and for asynchronous messages.

- reportError - Request an error report if a message cannot be delivered or if the action triggered by a message failed.
- reportCOA - Request a report when the message is placed into the destination queue.
- reportCOD - Request a report when the message is read from the destination queue.
- reportNAN - Request a report when the receiving application accepts the message.
- reportPAN - Request a report when the receiving application rejects the message.
- flushReports - Reset the report parameter.

MessageType

Defines the type of a message. Currently this are the four types defined by Websphere MQ.

Expire

This parameter defines how long a message will be kept in a queue. If nobody read the message from the queue within this time the queue manager will remove it from the queue. By default this is set to unlimited. Somebody else (i.e. Tivoli) has to take care.

Message ID

The message identifier. This is a string that distinguishes one message from another. By default WebSphere MQ will generate this parameter.

Correlation ID

This is a string that relates one message to another (i.e. a reply to a request). By default WebSphere MQ will generate this parameter.

ReplyToQ
This parameter defines the reply queue to which a reply or report message has to be sent. It is only set for requests and asynchronous messages. An application retrieving a request has to use this parameter to identify the queue where to send a reply.

ReplyToQmgr
This parameter defines the queue manager that contains the reply queue to which a reply or report message has to be sent. It is only set for requests and asynchronous messages. An application retrieving a request has to use this parameter to identify the queue manager where to send a reply.

We may promote additional parameter later. Currently this set of parameter is sufficient to support the scenarios defined in the design document.

Messages that trigger an action in the receiver must store a selector for the action somewhere inside the message itself. The selector can be coded into the application data itself or it can be stored into an unused parameter in the message descriptors. The MQ-Interface uses a decorator for defining the storage of the action selector. The MQ-interface has a variable that contains a decorator object. An application can set the decorator individually for each message. For convenience reasons the class MQMessage contains a class method where an application can define a decorator that should be used for all messages.

The messages action, action:, data, and data: call the same methods in the decorator. The methods in the decorator know how to store the action selector and how to retrieve it. The two methods below show the messages action and action: for a decorator that stores the action selector into the parameter applicationName of the message descriptor.

action
```
"Answer the action selector coded into a parameter
of the message descriptor"

^message applicationName
```

action: aString
```
"Answer the action selector coded into a parameter
of the message descriptor"

message applicationName: aString
```

The messages data and data: implemented in the base class ActionDecorator just store the application data into the message. They support decorator that store the selector in the application data. The concrete subclass DefaultActionDecorator does this.

Asynchronous message

An asynchronous message represents a message that is used by an application to send information to another application but does not require a reply. An application may define reports to be sent back. A developer creates a new asynchronous message by sending the message newAsynchronousMessage to a sender queue.

```
| message|
message := sender newAsynchronousMessage.
message action: #tick.
message data: #[ 84].“Integer value for T”
sender put: message.
```

The above code fragment send a time tick with a Boolean value (true) to another application. We are not interested what the other application is doing with it or even if it is reading it.

If we want to know if the message is delivered or if the other application accepts the message we have to define reports.

```
| message report |
message := sender newAsynchronousMessage.
message action: #processOrder.
message data: anOrder asMQData.
message reportCOD.
sender put: message.
report := receiver getReportFor: message.
```

This code fragment sends an order to another application. We want to know if the other application read the message from the queue. We are not interested what exactly the other application is doing with the order. We just want to know it has it.

The code fragment waits for a report after it has sent the message. The queue manager creates that report when the receiver application reads the message from the queue. In this example the default wait interval of the queue is used. If no report arrives within this time interval the queue raises an exception. If you want to use a different wait interval you have to use the message getReportFort:wait:. The second parameter of the message defines the wait interval.

In the above example we assume that the sender queue is linked to a receiver queue. The sender queue copies the name of the receiver queue and the name of the queue manager into the message before it send it. If the queues are not linked then the developer has to set the queue parameter manually as shown in the fragment below.

```
| message report |
message := sender newAsynchronousMessage.
message action: #newOrder.
message data: anOrder asMQData.
message reportCOD.
message replyQueue: receiver.
sender put: message.
report := receiver getReportFor: message.
```

The queue will create the message and correlation ids. After the message is sent the parameter can be read from the message description. The receiver now can associate a report it receives to the message. If you want to define your own message and correlation id you can do so by using the messages messageID: and correletionID:.

You may define several reports. Based on the report you define you may have to read several reports. The reports will arrive in the following order:

COA	Confirmation of arrival	the message is place into the destination queue
COD	Confirmation of delivery	the message is read from the destination queue
PAN/NAN	Positive or negative action notification	the retrieving application accepts or rejects the message. They should always used together.

An error report may arrive anytime. The queue manager creates an error report when the message cannot be delivered or an application send an error report if the action triggered by the message failed.

If you define all report types you have to retrieve three reports: COA, COD, PAN/NAN. You only break this if you retrieve an error report.

Request

Use should a request if you want to receive a response to a message you sent to another application. An application that retrieves a request has to create a reply and send it back to the sender application.

```
| request reply |
request := sender newRequest.
request action: #saveAddress.
request data: anAddress asMQData.
sender put: request.
reply := receiver getReplyFor: request.
```

The above code fragment creates a request to store an address into a server. The server will process the request and send a reply back to the sender. In the example we assume that the sender and receiver queues are linked – the sender knows the queue where the request will arrive. The sender copies the name of the queue and the name of the queue's queue manager into the message. Since we do not define a message or correlation id WebSphere is doing it for us.

You can define the same reports for a request as for an asynchronous message. They will be handled the same as described above. The following example shows a code fragment where we want to know if the message is delivered.

```
| request reply report |
request := sender newRequest.
request action: #saveAddress.
request data: anAddress asMQData.
message reportCOD.
sender put: request.
report := receiver getReportFor: request.
reply := receiver getReplyFor: request.
```

The queue manager creates the report when the receiver read the request from the queue. The application process the message – stores the address – and sends a reply back to the same queue as the queue manager sent the report. Thus the report will arrive before the reply and we have to retrieve them in this order.

Reply

If an application retrieves a request it must send a reply back to the queue defined in the request.

```
listenForIncommingMessages
“A simple WebSphere MQ listener”

| message reply answer |
[message := listener getIncommingMessage.
sender := self queueManager
getSenderQueue: aMessage replyQueueName.
answer := self
        processAction: message action
        with: message data.
reply := message createReply.
reply data: answer asMQData.
sender put: reply.
true] whileTrue: [].
```

This sample method can be use as a simple listener that route any incoming message to its own class. The class process the action coded into the request. The listener then creates a reply, copies the response into the reply and sends it back.

A request has to carry information – the name of the receiver queue and the name of its queue manager - where to send the reply. The receiving application uses this information to send the request back to the sending application. The message createReply will copy all necessary parameter from the request to the reply.

In this example we assume that we only have one queue manager. Thus we don’t have to handle the queue manager parameter.

This message can be forked – maybe in an initialize - as a separate process running in the background. Then it does not block the rest of the image.

```
initialize
    “Fork a process for a listener”

    listener := self queueManager
    getReceiverQueue: ’app1.rec.queue’.
    process := [self listenForIncommingMessages].
    process forkAt: Processor userInterruptPriority.
```

Report

An application may define reports it want to receive when sending an asynchronous message or a request to another application. Some of the reports must be created by the receiving application.

```
listenForIncommingMessages
“A simple WebSphere MQ listener”

| message reply report answer |
[message := listener getIncommingMessage.
sender := self queueManager
getSenderQueue: aMessage replyQueueName.
(message requiresPANReport and: [self accept: message action])
ifTrue: [sender put: (message createReport: #PAN)].

answer := self
            processAction: message action
            with: message data.
reply := message createReply.
reply data: answer asMQData.
sender put: reply.
true] whileTrue: [].
```

The above sample work the same as the example before except it checks if it has to send a PAN back to the sending application before it process the action triggered by the message. The sample sends the report through the same queue as the reply. The sending application first has to retrieve the report and then it has to retrieve the reply.

The message createReport: creates a new report and copies all necessary parameter to the report. The parameter defines what report is send back. Currently we support three report types:

- PAN is sent when the application accepts the message.
- NAN is sent when the application rejects the message.
- Error is sent when the action triggered by the message raised an error.

2

Class Reference

This chapter describes all classes and public methods of the MQ-interface.

QueueManager

QueueManager is an abstract class that represents a WebSphere MQ queue manager. It instantiate a concrete class based on parameter in the instance creation methods. A queue manager keeps two lists of all queues an application requests. One list for receiver queues and one list for sender queues. The manager uses this two lists to maintain the life cycle of its queues.

Super class

Object

Class methods

new: *aSymbol* **threaded:** *aBoolean*

This factory method create a concrete subclass of QueueManager based on the parameter of the message. WebSphere MQ supports two different setups for the queue manager. Additionally it provides libraries for threaded and none threaded API calls. Therefore there are four subclasses that are defined by the two parameter of the message.

aSymbol	aBoolean	
#local	true	LocalThapiQueueManager
#local	false	LocalQueueManager

aSymbol	aBoolean	
#remote	true	RemoteThapiQueueManager
#remote	false	RemoteQueueManager

Instance methods

name
Return the name of the queue manager

name: *aString*
Set the name of the queue manager

connect
Connects an instance of a concrete subclass to the WebSphere MQ queue manager. The application now can use other WebSphere MQ services.

disconnect
Disconnect the application from the WebSphere MQ queue manager. The application cannot use WebSphere MQ services anymore.

getReceiverQueue: *aString*
Request a ReceiverQueue instance for a queue name. If the queue manager has already a receiver queue registered for this name it just answer the queue. If it does not have a receiver queue registered for it, it will create one and register it.

getSenderQueue: *aString*
Request a SenderQueue instance for a queue name. If the queue manager has already a sender queue registered for this name it just answer the queue. If it does not have a sender queue registered for it, it will create one and register it.

LocalQueueManager

LocalQueueManager is a concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on the same machine as the image. It is using the WebSphere MQ none threaded server library. This class does not has an own public protocol. It just defines the DLLCC class for the threaded server library.

Super class

QueueManager

LocalThapiQueueManager

LocalThapiQueueManager is a concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on the same machine as the image. It is using the WebSphere MQ threaded server library. This class does not has an own public protocol. It just defines the DLLCC class for the threaded server library.

Super class

LocalQueueManager

RemoteQueueManager

RemoteQueueManager is a concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on a different machine using the WebSphere MQ none threaded client library.

Super class

QueueManager

Instance methods

host
Returns the name or the ip-address of the host where the queue manager is installed.

host: *aString*
Set the name or the ip-address of the host where the queue manager is installed. When using the host name to define the host then the local machine must be able to resolve the ip-address for the host name.

port
Returns the port of the listener.

port: *aNumber*
Set the port of the listener. This must only be set if there are more then one queue manager installed on the remote server.

channel
Return the name of the channel that connects the application to the remote queue manager.

channel: *aString*
Set the name of the channel that connects the application to the remote queue manager.

RemoteThapiQueueManager

A concrete queue manager class that connects a VisualWorks image to a queue manager that is installed on a different machine using the WebSphere MQ threaded client library. This class does not has an own public protocol. It just defines the DLLCC class for the threaded client library.

Super class

RemoteQueueManager

MessageQueue

MessageQueue is an abstract class that represents the link to a queue in a queue manager. In WebSphere MQ a queue can only be opened for writing or for reading. It is not possible to open a queue for both. Therefore the MQ-Interface contains two subclasses of this class that represent the different behavior for sender and receiver queues.

Initially an instance of a MessageQueue subclass is not linked - by opening - to its real WebSphere MQ queue. The object opens the queue when the first get: or put: message is sent to the object. A developer may force a queue to establish the link by sending the message open to it. A developer may define a time frame after which a queue will be closed if no get: or put: message is send to the queue within this time frame.

Super class

Object

Instance methods

name
Answers the name of the queue

open
Opens a WebSphere MQ queue. Now an application can retrieve messages from it or send messages to it. This method is automatically called when an application sends the first get: or put: message to it.

close
Close the WebSphere MQ queue. An application may send this message manually. It also can leave it to the queue manager that maintains the live cycle of its queues.

timeout: anInteger
Define the time interval for closing the queue resource. If no get: or put: is issued in this frame the queue is closed but the instance remains in the same state as being just created

ReceiverQueue

ReceiverQueue is a concrete message queue class that is or will be opened for reading messages from a queue.

Super class

MessageQueue

Class methods

defaultMatchingParameter: *aStringCollection*
Define the default parameter that will be used to retrieve selected messages from the queue. By default there is none. The first message in the queue is received from the queue

Instance methods

matchingParameter: *aStringCollection*
Define the parameter that will be used to retrieve selected messages from the queue. If this message is not send then the default parameter will be used to read messages from the queue.

getReportFor: *aMessage*
Try to receive a report from the queue. This method uses the default wait interval of the queue. If no report arrives in time then the queue raise an error. If a matching report arrives the method answers this report.

getReportFor: *aMessage* **match:** *aStringCollection*
Retrieve the first report from the queue that match the parameter provided by the last parameter. This method overrides previous match settings in the queue.

getReportFor: *aMessage* **wait:** *aNumber*
Try to receive a report from the queue. This method uses a defined wait interval. The default value of the queue is ignored. If no report arrives in time then the queue raise an error. If a matching report arrives the method answers this report.

getReportFor: *aMessage* **wait:** *anInteger* **match:** *aStringCollection*
Retrieve the first report from the queue that match the parameter provided by the last parameter. The report is sent back by another application to an asynchronous message or request – the parameter of this method. This method is used to override the timeout and match setting of the queue.

getReplyFor: *aRequest*
Try to receive a reply from the queue. This method uses the default wait interval of the queue. If no reply arrives in time then the queue raise an error. If a matching reply arrives the method answers this reply.

getReplyFor: *aMessage* **match:** *aStringCollection*
Retrieve the first reply from the queue that match the parameter provided by the last parameter. The reply is sent back by another application to request – the first parameter of this method. This method overrides previous match settings in the queue.

getReplyFor: *aRequest* **wait:** *aNumber*
Try to receive a reply from the queue. This method uses a defined wait interval. The default value of the queue is ignored. If no reply arrives in time then the queue raise an error. If a matching reply arrives the method answers this reply.

getReportFor: *aMessage* **wait:** *anInteger* **match:** *aStringCollection*
Retrieve the first reply from the queue that match the parameter provided by the last parameter. The reply is sent back by another application to a request – the first parameter of this method. This method is used to override the timeout and match setting of the queue.

getIncommingMessage
Read any message from the queue. This method uses the default wait interval of the queue. If no message arrives in time then the queue raise an error. If a matching message arrives the method answers this message. We recommend to define an infinitive wait and to use this method in a separate process to avoid the blocking of the whole image.

The main purpose of this method is in implementing a listener that handles unexpected incoming messages.

waitInterval: *aNumber*
Define the default wait interval for an attempt to retrieve a message from a receiver queue. The parameter defines the wait interval in milliseconds.

waitForever
Define the default wait interval for an attempt to retrieve a message from a receiver queue to infinite. A get: message will wait forever.

noWait
Define the default wait interval for an attempt to retrieve a message from a receiver queue to zero. A #get: message will return immediately. If there is no message in the queue it will raise an error.

SenderQueue

SenderQueue is a concrete message queue class that is or will be opened for writing messages to a queue. A sender queue can be linked with a receiver queue. Asynchronous messages and requests send to another application through the sender queue carry the name of the linked receiver queue and the name of its queue manager. Reports and replies will later arrive in the linked receiver queue.

Super class

MessageQueue

Instance methods

newAsynchronousMessage
Create a new asynchronous message and preset it according to the queue's settings. The message should be sent through the queue that created it.

newRequest
Create a new request and preset it according to the queue's settings. The message should be sent through the queue that created it.

receiverQueue: *aReceiverQueue*
Links a receiver queue to the sender queue. The sender queue will use this setting to set the queue name and queue manager name in messages sent through this queue.

put: *aMessage*
Write a message to the queue.

MQMessage

An abstract class for the four different message types IBM defined for WebSphere MQ. We named it MQMessage to be sure that it does not collide with the base class Message.

Super class

Object

Instance methods

data
Answer the application data transferred by a message. WebSphere MQ only transfers raw data. Thus an application has to convert the byte array into its domain objects.

data: *aByteArray*
Set the application data transferred by the message. Since WebSphere MQ only transfers raw data the application has to convert its domain objects into a byte array

reportCOA
Set the flag for receiving a confirmation of arrival. This report is generated by the queue manager when the message is placed into the destination queue

reportCOD
Set the flag for receiving a confirmation of delivery. This report is generated by the queue manager when the message is read from the destination queue

flushReports
Reset the report flags. No report is expected.

replyQueue: *aQueue*
Define the queue where a receiving application has to send replies and reports. The method will copy the queue name and its queue manager name to the message

ActionMessage

ActionMessage is an abstract class for messages that are indented to trigger an action in the receiver of the message. These message types carry a symbol in one of the parameter of the message descriptor.

Super class

Object

Class methods

defaultDecoratorClass: *aClass*
Define the class of the default decorator that will be used to store and retrieve the action selector into the message.

Instance methods

actionDecorator: *aDecorator*
Set the decorator that codes the action message into the descriptor, into the application data or both.

action
Answers a string that identify the action an application has to perform when retrieving an action message.

action: *aSymbol*
Set the action to be performed when receiving an action message.

reportError
Set a flag for receiving error reports. These reports are created by the queue manager when a message cannot be delivered or by an application if the action triggered by a message failed.

reportNAN
Set the flag for receiving a negative action notification. This report is generated by the receiver application when the message is rejected.

reportPAN
Set the flag for receiving a positive action notification. This report is generated by the receiver application when the message is accepted.

requiresErrorReport
Answer true if the receiver want to receive error reports, answer false otherwise.

requiresPANReport
Answer true if the receiver want to receive a positive action notification, answer false otherwise

requiresNANReport
Answer true if the receiver want to receive a negative action notification, answer false otherwise

replyQueueName
Answer the name of the queue where a receiving application has to send replies and reports.

replyQueueManagerName
Answer the name of the queue manager that contains the queue where a receiving application has to send replies and reports.

createReport: *anActionMessage*
Create an Report instance for an action message. This method copies all necessary parameter from the action message to the report.

AsynchronousMessage

An AsynchronousMessage is a concrete message class for a message that simply carries some information to another application. It does not generate a reply. This class does not have an own public protocol

Super class

ActionMessage

Request

Request is a concrete action message class for a request message. You should use this message type if you want to receive a reply to a message.

Super class

ActionRequest

Instance methods

createReply: *anActionMessage*
Create a Reply instance for an action message. This method copies all necessary parameter from the action message to the reply. The application just has to add the data to the reply.

Reply

Reply is a concrete message class for a reply an application has to generate when receiving a request. This class does not have an own public protocol

Super class

MQMessage

Report

Report is a concrete message class for reports generated by WebSphere MQ itself or by an application about the state of a message or an error.

Super class

MQMessage

Instance methods

isError
Answers true if it is an error report. Answers false otherwise

isCOA
Answers true if it is an confirmation of arrival report. Answers false otherwise

isCOD
Answers true if it is an confirmation of delivery report. Answers false otherwise

isPAN
Answers true if it is a positive action notification report. Answers false otherwise

isNAN
Answers true if it is a negative action notification report. Answers false otherwise

ActionDecorator

ActionDecorator is an abstract class that stores the action selector somewhere in the message. A subclass has to implement the messages action and action: that implement the concrete storage of the action selector (i.e. in an unused parameter of the message selector).

Super class

Object

Instance methods

message: *aMQMessage*
Sets the message where the decorator stores and retrieves the action selector. This method is called when a decorator is assigned to a message.

data
Answers the data part of the message.

data: *aByteArray*
Copy the byte array into the data part of the message

DefaultActionDecorator

DefaultActionDecorator is a concrete class that stores the action selector together with the application data in the data part of the message. The selector is just stored before the application data. The first byte of the data part contains the size of the selector. This makes it easy to split the selector and the application data when an application receives the message. The class contains instance variables that cache the selector and application data. Only id both variables are set, then the decorator combines them and stores it as data part of the message.

Super class

Object

Instance methods

action
Answers the action selector of the message

action: *aString*
Sets the action selector of the message. If the data is set, too then combine them and store them in the data part of the message

data

Answers the application data of the message

data: *aString*

Sets the application data of the message. If the action is set, too then combine them and store them in the data part of the message