Banter Relationship Objects API User's Guide

Relationship Manager Version 4.5

banter

Banter Relationship Objects API User's Guide
May 17, 2001

Banter Relationship Manager Version 4.5

Banter Inc.
60 Federal Street
Suite 550
San Francisco, CA
94107

Tel: 1-415-247-2600
Fax: 1-415-247-2626
E-mail: info@banter.com

Table of Contents

About this Document

The *Banter Relationship Objects API User's Guide* provides applications programmers with the information they need to effectively use the Banter Relationship Objects API for Relationship Manager, version 4.5. This book is class-based; chapters are arranged in logical order, providing you with appropriate descriptions, as you require them. Read each chapter thoroughly to effectively use and attain maximum benefit from the Banter Relationships Objects API (BRO API).

This book contains the following chapters:

- **Chapter 1 BRO API Overview** – A general introduction to the Relationship Manager system and a brief discussion of the Banter Relationship Objects API
- **Chapter 2 BRO API Applications** – A general description of the three main applications that you will create using the BRO API
- **Chapter 3 Class Types and Descriptions** – A discussion of each of the class types in the BRO API
- **Chapter 4 Examples** – Examples of the three main applications that you will create using the BRO API

Purpose of this Document

The purpose of this document is to enable the reader to make effective use of the Banter Relationship Objects API (BRO API). This API is a C++ object-oriented interface to the Relationship Manager. After having read this document, the user should be able to write applications that communicate with the Relationship Manager in connection with other enterprise systems.

The *Banter Relationship Objects API User's Guide* provides an overview of the BRO API and is a companion to the *Banter Relationship Objects API Object Dictionary*, which documents each of the classes in the API.

Conventions

The following conventions are used in this document:

Example	Description
GO TO	Directs the reader to a different point in a procedure or document.
raMessage	Italics indicate document names, function names, and class names.
[minutes]	Square brackets indicate an optional value.
Message Center	Application names and Queue names are capitalized.
Note:	Indicates important additional information.

Diagrams Shown in this Document

The diagrams in figures 2 through 8 are Universal Modeling Language (UML) diagrams. Relationships between objects in the diagrams follow UML standards.

CHAPTER 1

BRO API Overview

This first chapter contains the following sections:

- Welcome to Banter Relationship Manager
- Purpose of the BRO API
- Relationship Manager Data Flow
- Other Documentation

Welcome to Banter Reply

Banter Reply is a self-tuning solution for managing e-mail and web form communications. *Reply* is a component of Relationship Manager—Banter's enterprise-class suite of e-communications products.

Banter's core technology—the Relationship Modeling Engine™ (RME)—enables *Reply* to accurately analyze the language, intent, and context of customer communications. *Reply* harnesses the power of the RME to determine an appropriate automatic or semi-automatic response to each communication. Furthermore, *Reply* uses Banter's Adaptive Knowledge Base to track and learn from normal message activity. *Reply* employs this acquired knowledge to tune the system and improve the accuracy of its responses to future customer interactions.

Banter Reply's analysis, tracking, and self-tuning abilities, enable the system to handle large quantities of customer communications and respond with unprecedented accuracy.

Relationship Manager Environment

The Relationship Manager environment includes server and client-side applications. The services run on either Sun® Solaris™ with an Oracle® server or on Windows® NT™ with SQL Server™. The client-side application (Message Center) runs on Windows NT with either an Oracle® or SQL Server™ database. There is also a web-based version of the Message Center, which runs under Internet Explorer version 4.0 and above.

Purpose of the BRO API

The Banter Relationship Objects API (BRO API) is a collection of C++ classes. These classes:

- Enable users to build three types of applications that connect to the Relationship Manager system: input connectors, output connectors and active queue applications. (See "Applications that Connect to Relationship Manager" on page 3).
- Enable users to view the properties of Relationship Manager configuration objects, and to view and manipulate messages in a Relationship Manager system.

Relationship Manager Data Flow

Relationship Manager's message processing data flow has three basic stages:

- **Receiving inbound messages** – Relationship Manager receives inbound messages from various data channels. Customer applications, known as input connectors, capture incoming messages and send them to Relationship Manager for processing. Input connectors place these messages in an area known as the input tray. When a message is placed in the input tray, it is ready to be processed by the Relationship Manager.

Note: Input connector applications must handle incoming messages arriving from inbound channels and place them in the input tray.

- **Managing and processing messages** – When messages have been put in the input tray, Banter's Relationship Modeling Engine (RME) analyzes them, and the Relationship Manager processes and routes the messages. From this point, if the message is designated for manual handling, the Message Center can help agents manage, categorize, track, and process messages.
- **Sending outbound messages** – When message processing is complete, Relationship Manager places messages in the output tray. The message is now available for processing by an output connector. Output connectors extract a message from the Relationship Manager, continue processing the message in an external application space, and send it on to outbound message channels.

Figure 1 below illustrates how data flows into and out of Relationship Manager.

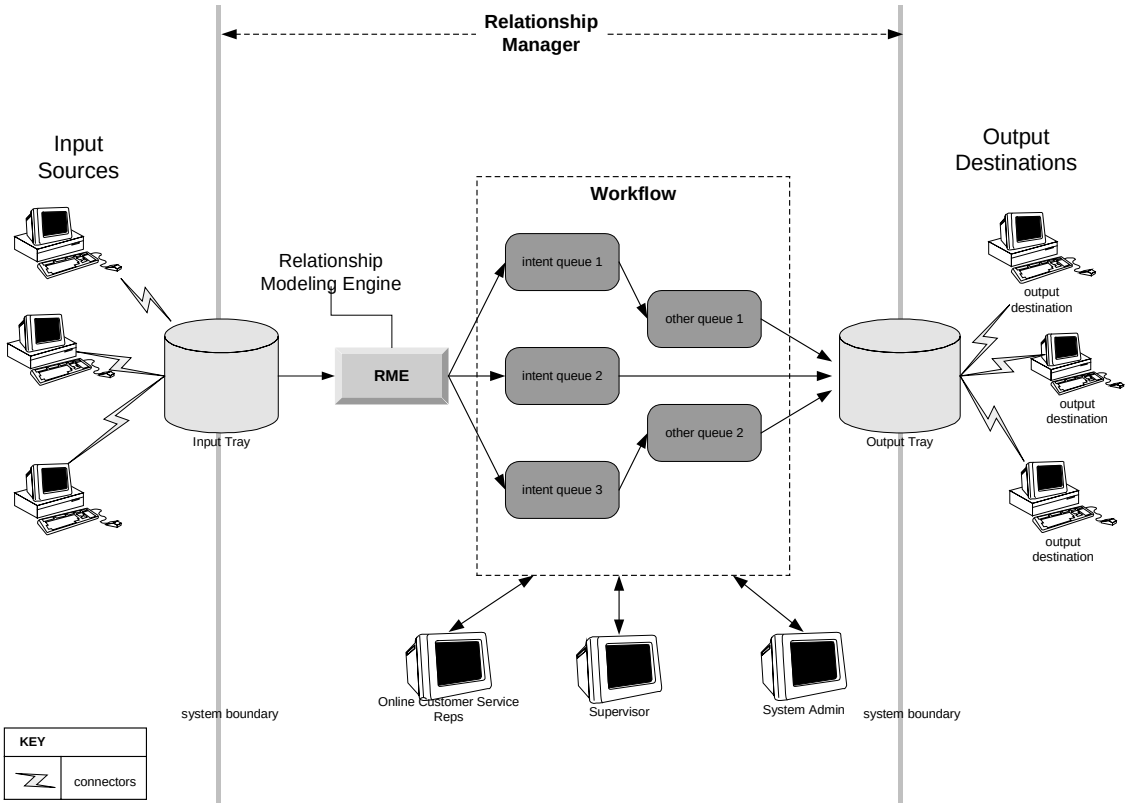

Figure 1 – Relationship Manager Workflow

Formatting Messages in Input and Output Trays

Inbound messages are received from various data channels (e-mail, XML sources, web forms, chat, etc.) and are formatted into Name Value Pairs (NVPs); some NVPs are pre-defined (internal NVPs), and some may be user-defined. All NVPs defined in Relationship Manager's NVP Dictionary may be used by the BRO API. Moreover, all user-defined NVPs must exist in the NVP Dictionary before they can be used by the Banter Relationship Objects API (BRO API). User defined NVPs are created with Relationship Manager's Configuration application.

Applications that Connect to Relationship Manager

There are three types of applications that connect to Relationship Manager.

- **Input connectors** – An input connector application delivers messages to Relationship Manager from an external source by inserting one or more messages into the input tray. The Relationship Manager is not aware of sources of messages, which can include email services. However input connectors may know about sources of messages (e.g., email services).
- **Output connectors** – An output connector application delivers messages to an external source from Relationship Manager by extracting one or more messages from

the output tray. Since the Relationship Manager is not aware of e-mail services, it is the job of the output connector to format the messages it has extracted from the output tray and send them onwards to e-mail services.

- **Active queue applications** – Because there is often a need to customize message processing within Relationship Manager, we provide the ability to create active queue applications using the BRO API.

  Active queue applications are programs that perform customized, site-specific processing on messages routed to specially configured (or *active)* queues. The actual processing may vary depending on the content of these messages.

Other Documentation

The following documents may provide helpful background information about Relationship Manager:

- *Banter Reply Guide for Agents*
- *Configuration User's Guide*
- *Relationship Manager Installation for Windows NT/SQL Server*
- *Relationship Manager Installation for Windows NT/Oracle Client*
- *Relationship Manager Installation for Sun Solaris/Oracle Server*

CHAPTER 2

BRO API Applications

This chapter provides a general description of the following BRO API applications:

- Active Queue Applications
- Input Connector
- Output Connector

Active Queue Applications

Active queue applications process messages that are pending in a queue that has been configured as active. Each active queue application has the name of a configured active queue as one of its parameters. All messages in an active queue undergo active queue processing. To write an active queue application, create a class derived from *raActiveQueueFramework.*

Class raActiveQueueFramework

The *raActiveQueueFramework* class is an abstract base class for active queue applications.

Note: For an example of an active queue application, see the example on page 31.

How to use raActiveQueueFramework

- Create an active queue class derived from *raActiveQueueFramework*
- Write your specific active queue code for handling one message in the function *ProcessMessage()* of your derived active queue class.

Note: The method *Work()* is called periodically from the method *Run()* iterates through all pending messages, and send each of them to *ProcessMessage()*.

For an example of an active queue application, see page 31.

Input Connector

Overview

Input connectors deliver messages to Relationship Manager from external sources.

The input connector is responsible for:

- Fetching data from external sources (e.g., SMTP, WEB forms, other databases)
- Dividing the data into pre-defined and optional customer-specific NVPs and submitting the NVPs to the input tray.

Class raInputConnectorFramework

The *raInputConnectorFramework* class is an abstract base class for input connector applications.

How to use raInputConnectorFramework

- Create an input connector class derived from *raInputConnectorFramework*
- Implement *virtual void BuildMessage(raInMessage& inMessage)* to build a new message that will be added to the Relationship Manager input tray by *Work()* (i.e., insert into the message the necessary NVPs by using *raMessage*'s inherited methods, such as *AddNvp*, *SetNvp*).
- Implement *virtual bool ExistNewMessage()* to check if there are messages waiting to be added to the Relationship Manager input tray. The function should return true if messages are currently waiting to be insert into the input tray. The function should return false if no messages are waiting to be inserted into the input tray.

Note: The method *Work()*, called periodically from the method *Run()* iterates through all pending messages, and sends each of them to *ProcessMessage()*.

For an example of an active queue application, see page 31.

Class raInputTray

Class *raInputTray* is the interface for the input tray. The input tray is the gateway through which external applications may insert messages into a Relationship Manager installation. To access the input tray, use *raInstallation::GetInputTray().*

If you wish to handle the insertion of the messages into the input tray yourself, you can rewrite the virtual function *Work()*.

To insert a message into the input tray:

1. Construct an object of type *raInstallation*
2. Get the input tray by using *raInstallation::GetInputTray()*

3. Create a new *raInMessage* by calling *raInputTray::CreateNewMessage()*
4. Insert the necessary NVPs by using *raMessage*'s inherited methods (e.g., *AddNvp*, *SetNvp*)
5. Put the message into the input tray by calling *raInputTray::Put*

Note: For an example of an input connector, see page 23.

Output Connector

Overview

After the Relationship Manager system processes a message, an output connector delivers the message to the mail server for delivery to the recipient.

The output connector is responsible for:

- Polling the output tray for messages that are ready for dispatch
- Formatting and delivering messages to an external entity

Class raOutputConnectorFramework

The *raOutputConnectorFramework* class is an abstract base class for output connector applications.

How to use raOutputConnectorFramework

- Create an output connector class derived from *raOutputConnectorFramework*
- Implement *virtual void ExtractMessage (const raOutMessage& outMessage)* to process the message extracted from the Relationship Manger output tray, according to your specific output connector needs (e.g., you might format the message and deliver it to an external entity).

Note: For the method *Work()*:
- *Work()* is called periodically from the method *Run()*
- *Work()* extracts messages that are ready from the output tray
- *Work()* extracts each message one by one and sends each message to be processed by *ExtractMessage()*.

For an example of an output connector application, see page 26.

Class raOutputTray

When message processing by Relationship Manager is complete, and the message is ready to be sent outside the system, Relationship Manager places the message in the output tray. Once the message is in the output tray, it is available to an output connector. The output connector will extract the message from the Relationship Manager, do further

processing outside of the Relationship Manager, and send the message to outbound message channels.

If you want to handle the polling of the messages from the output tray manually, you can rewrite the virtual function *Work()*.To obtain the output tray, use *raInstallation::GetOutputTray()*.

To extract a message from the output tray

1. Construct an object of type *raInstallation*
2. Get the output tray by using *raInstallation::GetOutputTray()*
3. Retrieve an *raOutMessage* by calling *raOutputTray::Get*
4. Process the message by using *raMessage's* inherited methods (e.g., *GetNvp*, *GetMultipleNvp*)

To complete the message processing, do one of the following:

- Mark the outgoing message as closed by calling *raOutMessage::Close()*.
  This will leave the message in the database, making it unavailable to the output tray.
- Remove the message by calling *raOutMessage::Delete()*.
  This will remove the message physically from the database.

You can also delete a message that has been closed earlier.

Note: For an example of an output connector, see page 26.

CHAPTER 3

Class Types and Descriptions

The Banter Relationship Objects API (BRO API) is a collection of C++ classes. These classes are organized according to the following class types:

- Navigation
- Containers
- Iterators
- Configuration Objects
- Message Hierarchy
- Exceptions
- Miscellaneous/Helpers

Navigation

The following are navigation classes:

- Class *raInstallation*
- Class *raApplicationFramework*
- Class *raActiveQueueFramework*
- Class *raInputConnectorFramework*
- Class *raOutputConnectorFramework*

Class raInstallation

For applications using the BRO API, *raInstallation* is a gateway to an installation of the Relationship Manager.

Class *raInstallaton* is used to obtain Relationship Manager objects: Queues, Categories, Canned Texts and Users. It is also used to obtain, create, and manipulate messages and to put Trace logs.

This is the first BRO object that should be used. All other objects can be obtained by navigating from it.

Use raInstallation to do the following:

- Get the containers of Relationship Manager objects (i.e., Queues, Categories, Canned Texts, Users and System Messages). Use the containers to get single objects and messages that you need to access.
- Get the *raTrace* object and use it to put traces into Banter Monitor to help you follow the course of your application run.

Transaction Control

A transaction represents a group of database changes treated as one unit. Either all the changes in a transaction are saved together, or none of the changes are saved.

To ensure that a series of database changes are saved within one transaction, use the following pattern example:

```
raInstallation::BeginTransaction();
```

change 1

change 2

change 3

...

last change

```
raInstallation::CommitTransaction();
```

If an error occurs during the transaction, the transaction can be rolled back, and none of the changes will be saved. To roll back a transaction use the following:

```
raInstallation::RollbackTransaction();
```

Note: Using transaction control is optional. If you do not call *BeginTransaction()* each Save operation will be treated as a single transaction (See `raActiveMessage` synopsis).

Nested Transactions :

If you nest one transaction inside another transaction, only the last `CommitTransaction()` will commit a database transaction.

Example

```
void CloseDueToCustomerRequest (raActiveMessage& message)
{
    try {
        GetInstallation().BeginTransaction();
        message.Close();
```

```
        message.WriteLogEntry("Close message due to customer request");
        GetInstallation().CommitTransaction();
    } catch (raxException) {
        GetInstallation().GetTrace().Put(raTrace::S_ERROR,
                                                "Problem while closing message");
        GetInstallation().RollbackTransaction();
        throw;
    }
}
void ClosePendingMessages(raQueue& queue)
{
    try {
        GetInstallation().BeginTransaction();
            raActiveMessageSet::Iterator iter;
        iter = queue.GetPendingMessages();
        For ( ; !iter.IsAtEnd() ; ++iter ){
            raActiveMessage message(*iter);
            CloseDueToCustomerRequest(message);
          }
        message.WriteLogEntry("Close Pending messages due to customer

              request");
        GetInstallation().CommitTransaction();
    } catch (raxException) {
        GetInstallation().GetTrace().Put(raTrace::S_ERROR,
                                                "Error: Roll back transaction.");
        GetInstallation().RollbackTransaction();
    }
}
```

Note: The behavior of the transaction control in cases of non-standard use (e.g., call to *CommitTransaction()* without calling *BeginTransaction()* first) defers from one database to another.

If you have called *RollbackTransaction*, the entire transaction will be rolled back. However, calling *RollbackTransaction* a second time without beginning another transaction will have no effect. This situation can occur as a result of nested transactions.

BeginTransaction

```
void BeginTransaction () throw (raxException);
```

Description

Begin a database transaction.

Exceptions

raxException – is thrown if the operation fails.

CommitTransaction

```
void CommitTransaction () throw (raxException);
```

Description

Commits a database transaction.

Exceptions

raxException – is thrown if the operation fails.

RollbackTransaction

```
void RollbackTransaction () throw (raxException);
```

Description

Rollback database transaction.

Note: If you have called RollbackTransaction, the entire transaction will be rolled back. However, calling RollbackTransaction a second time without beginning another transaction will have no effect. This situation can occur as a result of nested transactions.

Exceptions

raxException – is thrown if the operation fails.

Class raApplicationFramework

An *raApplicationFramework* class is an abstract base class designed to encapsulate BRO add-on applications – as active queues and connectors.

Any application based on *raActiveQueueFramework* will run as a Windows service or as a Unix background process.

A BRO add-on works with only one Relationship Manager installation.

The *raActiveApplicationFramework* class handles:

- Connection and disconnection to the database
- Automatic reconnection to the database if there is a database problem
- Bringing the program down when configuration has been changed
- Bringing the program down in an orderly fashion
- Ensuring that no more than one occurrence of an active queue application will run on the same queue
- Printing messages to the system log and the trace log

When using one of the *raApplicationFramework* derived classes, do the following:

- Create an application class derived from *raActiveQueueFramework, raInputConnectorFramework* or *raOutputConnectorFramework*
- Write your specific application code in the virtual functions *raActiveQueueFramework::ProcessMessage(), raInputConnector::BuildMessage() and raInputConnector::ExistNewMessage(),* or *raOutputConnector::ExtractMessage()*

In the* main() *function of your application do the following:

- Construct an *raInstallation*
- Construct a new application of your derived class type
- Call the *Run()* method, for example *activeQueue.Run()*

Note: *Optional* – To create an application that runs once and quits, call the function *SetStopping()* at the end of your *Work()* function.

Class raActiveQueueFramework

An *raActiveQueueFramework* class is an abstract base class for active queue applications. This class is derived from *raApplicationFramework*.

Note: For more information see “Class raActiveQueueFramework” on page 5.

Class raInputConnectorFramework

An *raInputConnectorFramework* class is an abstract base class for input connectors. This class is derived from *raApplicationFramework*.

Note: For more information see “Class raInputConnectorFramework” on page 6.

Class raOutputConnectorFramework

An *raOutputFramework* class is an abstract base class for output connectors. This class is derived from *raApplicationFramework*.

Containers

A BRO API container is a collection of Relationship Manager objects of the same type.

Note: The name for each container consists of the name of the object held by the container, followed by "Set". For example, *raQueueSet* and *raCannedTextSet* are containers for queue and canned text objects.

To obtain a container use *raInstallation::GetXxx()* function.

Note: No container class in the BRO API has a default constructor therefore you must use *raInstallation* to obtain it.

For example:

```
raInstallation currentInstallation ("dbSource", "dbName",
                                    "username", "password",
                                    "installationName",
                                    "processName");

raQueueSet queueSet = currentInstallation.GetQueueSet();
```

To sequentially go over the container items (or a subset of them), use iterators (see "Iterators" on page 15).

Containers have two kinds of Get functions:

- Get functions returning an iterator, enabling your application to iterate through a subset of the container's objects defined by each specific function.

  The following example returns an iterator that can be used to iterate through all *raCannedTexts* associated with the *raCategory* that was sent as a parameter:

  raCannedTextSet::iterator raCannedTextSet::GetByCategory(raCategory& category)

- Get functions returning one specific item form the container.

  The following example returns the category with the name sent as parameter to the function:

  raCategory raCategorySet::GetByName(const raString& name) const

The following container classes are obtained from *raInstallation*:

- *raCannedTextSet*
- *raCategorySet*
- *raQueueSet*

- *raSystemMessageSet*
- *raUserSet*

Iterators

An iterator is an object that allows the application to go over items in a container sequentially.

Note: At any given time, an iterator references one item in the container. To reference a different item, the iterator must be instructed to move.

It is important to know the following information about BRO API iterators:

- Each iterator is designed to iterate through a specific container.
- An iterator is initialized to iterate through a specific subset of items of the container.
- Each BRO API container class has at least one iterator associated with it.
- Each iterator class is a nested class of its associated container class.

To initialize an Iterator

- Call one of the Get functions of the containers.

  In some cases, you can get an iterator by using a Get function from a different class. For example:

  ```
  raActiveMessageSet::iterator raQueue::GetPendingMessages() const
  ```

  The above example returns an iterator to iterate through all the *raActiveMessages* pending in the calling *raQueue*.

NVP Iterators

A message consists of a collection of NVPs. Since the NVPs are components of a message, there is no specific container defined to contain them. For this reason, NVP iterators, can be obtained using *Get* functions of message classes (e.g., *GetAllNvps()*, *GetMultipleNvp()*).

Iterator Operations

The following methods are defined for all iterators:

- **default constructor** – creates an invalid iterator. To use it, you should assign to it a valid iterator obtained by a container *Get* function.
- ***operator=()*** – assignment operator
- ***operator*()*** – returns the container item referenced by the iterator
- ***IsAtEnd()*** – returns “true” if the iterator points beyond the end of the container
- ***operator++()*** – advances the iterator to the next item

Use the iterator as follows:

1. Construct a container (*raXxxSet* object).
2. Construct an iterator for that container.
3. Initialize the iterator by using one of the container's Get functions.
4. To go over the container items, do the following for each item:
   i. Use operator*() to get the item referenced by the iterator.
   ii. Check if the iterator reached the end of the container (using IsAtEnd()), if so – exit.
   iii. Advance the iterator using *operator++()*.
   iv. Return to step i.

Code Example

```
// create the installation
raInstallation thisInstallation ("dbSource", "dbName", "username",
                                 "password", "installationName",
                                 "processName");

// stage 1 - Construct  a container (raXxxSet object), and initialize it.
// Get the categories container from the raInstallation object
raCategorySet allCategories = thisInstallation.GetCategorySet();

// stage 2 - Construct  an iterator for that container.
raCategorySet::iterator i;

// stage 3 - Initialize the iterator by using one of the container's Get
functions.
// stage 4 - go over the container items
for (i = allCategories.GetAll();
     !i.IsAtEnd() ;       // stage 4ii - Check if the iterator reached the
end of the container
     ++i)                 // stage 4iii - Advance the iterator
{
      // stage4i - get the item referenced by the iterator.
      raCategory currentCategory = (*i);

      // here you can use currentCategory as you like
}
```

Configuration Objects

The following configuration objects are represented in the BRO:

- Queues
- Categories
- Canned Texts
- Users

Each of the above has an object that represents it:

- *raQueue*
- *raCategory*
- *raCannedText*
- *raUser*

These objects can be viewed, but not changed. For more details about each class role in the system, see the *Banter Relationship Objects API Dictionary*. For general information on Queues, Categories, Canned Texts and Users, see the *Configuration User's Guide*.

Each configuration object also has a container to represent a set of objects of that type:

- *raQueueSet*
- *raCategorySet*
- *raCannedTextSet*
- *raUserSet*

For more information about these containers, see "Containers" on page 14.

Message Hierarchy

The following are message hierarchy classes:

- Class *raActiveMessage*
- Class *raSystemMessage*
- Class *raInMessage*
- Class *raMessage*
- Class *raOutMessage*

Message Hierarchy Overview

This section describes the hierarchy of message classes and the relationship between all the message classes and NVPs.

Class raMessage

Class *raMessage* is an abstract base class of all messages classes in this library. It provides an interface for handling NVPs (both single and multiple-valued). The methods of this class enable the adding, setting, and modifying of the NVPs of a message object. For the BRO API, a message contains a collection of NVPs.

Note: There are no methods for removing NVPs from a message.

Relationship Manager supports three NVPs value types: integer, string (represented in the BRO API as *raString*) and blob (binary large object, represented in the BRO API as *raBuffer*).

Class raSystemMessage

Class *raSystemMessage* represents a message that has been processed by Relationship Manager.

In addition to providing access to NVPs, this class provides retrieval methods for different properties of a message in the system, such as its priority, and the arrival time, etc.

Note: Modifications to a system message are not allowed.

Class raActiveMessage

Class *raActiveMessage* represents an active message in the system. Active messages are system messages that can be modified by a BRO API user application.

In addition to *raSystemMessage* methods and properties, *raActiveMessage* has methods that enable it to change message NVPs and other properties (e.g., SLA times) and its state in Relationship Manager.

Class raInMessage

Class *raInMessage* represents a message created to be inserted into the input tray of a Relationship Manager system.

In addition to the NVP manipulation methods (inherited from *raMessage,)* this class provides the method *ForceToQueue()*, that enables the BRO API application to override a Relationship Manager's regular queue assignment for the message.

To insert a message into the input tray, the input connector must do the following:

1. Construct an object of type *raInstallation*
2. Get the input tray by using *raInstallation::GetInputTray()*
3. Create a new *raInMessage* – using *raInputTray::CreateNewMessage*
4. Insert the necessary NVPs – using *raMessage's* inherited methods *AddNvp*, *SetNvp* etc.
5. Put the message into the input tray – call *raInputTray::Put*

Class raOutMessage

Class *raOutMessage* represents a message created by Relationship Manager, intended to be sent outside the system. The message is placed in the output tray, from which it may be retrieved by calling *raOutputTray::Get(). raOutMessage* cannot be modified.

To extract a message from the output tray:

1. Construct an object of type *raInstallation*
2. Get the output tray by using *raInstallation::GetOutputTray()*
3. Retrieve an *raOutMessage* by calling *raOutputTray::Get()*
4. Process the message by using *raMessage's* inherited methods *GetNvp*, *GetMultipleNvp* etc.

To complete the message processing, do one of the following:

- Mark the outgoing message as closed by calling *raOutMessage::Close()*
  This will leave the message in the database, making it unavailable to the output tray.

– or –

- Remove the message by calling *raOutMessage::Delete()*
  This will remove the message physically from the database.

You can also delete a message that has been closed earlier.

Exceptions

The BRO library makes use of the standard C++ exception-handling mechanism.

It includes several exception classes, and most of the methods in the library throw exceptions.

To handle the exceptions, the customer application should use the try/catch blocks.

Note: If an exception is thrown, and the customer application does not catch it, the application will abort (immediately exit).

All of the BRO API exception classes are derived from *raxException*, and have the prefix "*rax*".

The method *raxException*::*GetText()* returns the text describing the error condition that caused the exception.

The following is a code example for handling an exception:

```
try {
      // The code calling the functions that might throw the exceptions
}   catch (raxException& x) {
      // You can use the string specified in the exception, here :
    raString str = x.GetText()
      // Handle the exception
}
```

For details about each exception, each function type and each cause for an exception throw, refer to the object dictionary.

Miscellaneous/Helpers

The following are miscellaneous/helper classes

- Class *raTrace*
- Class *raString*
- Class *raBuffer*

Class raTrace

The *raTrace* is used for creating records that describe events in the application run. The severity and the textual description of the event are parts of the event information defined by the *raTrace* user.

The trace object can be viewed using the Banter Monitor, provided with Relationship Manager.

A trace consists of:

- Severity – Defined by the *raTrace* object user
- Event text – Defined by the *raTrace* object user
- Time – Time and date that the *trace* was created
- Thread ID – ID of the thread (produced by the *raTrace* object)
- Trace ID – ID of the trace
- Subsystem – For every trace produced using the BRO API, the subsystem name is *Bro*

Enum raTrace::TraceSeverity

This enumeration represents the severity of the trace.

Use one of these enum values when putting a trace message (i.e., `raTrace::put([s_value],"Error while…")`) where `[s_value]` is one of the following:

S_FATAL

S_ERROR

S_WARNING

S_INFO

S_DEBUG

Class raString

Class *raString* is used to represent string data and is one of the three types of NVP values that can be added to a message.

Class raBuffer

Class *raBuffer* is used to represent binary data and is one of three types of NVPs values that can be added to a message.

CHAPTER 4

Examples

The following examples are included in this chapter:

- Input connector
- Output connector
- Active queue application

Input Connector

```
#include "Bro.h"
#include "raInputConnectorFramework.h"
#include "raInstallation.h"
#include "raInssage.h"

// The input connector class :
//////////////////////////
//     - Derived from raInputConnectorFramework
//     - Includes implementation of virtual functions BuildMessage() &
ExistNewMessage().
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
class InputConnectorSample : public raInputConnectorFramework
{
public:
       // Constructor
       ///////////////
       ///////////////
       InputConnectorSample(raInstallation& inst) : raInputConnectorFramework(inst){};

       // Builds a new message.
       // To be added to the input tray by Work().
       ////////////////////////////////////////////////////////////////////////
       ////////////////////////////////////////////////////////////////////////
       virtual void BuildMessage (raInMessage& inMessage);

       // Indicates if there are messages waiting
       // to be added to the input tray
       ////////////////////////////////////////
       ////////////////////////////////////////
       virtual bool ExistNewMessage ();
};


// Builds a new message.
// To be added to the input tray by Work().
////////////////////////////////////////
```

```
// Should contain specific input connector code for building one message (inMessage).
// It must be implemented while using raInputConnectorFramework ,
// But this implementation is only an example.
// For different input connector the implementation will be different.
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
void InputConnectorSample::BuildMessage (raInMessage& inMessage)
{

	/*
		The code segment that fetches data from external sources, and divides it
		into pre-defined NVPs should be here
	*/

	raString name;
	raString value;


	// Build the message out of NVP's
	////////////////////////////////////
	////////////////////////////////////

	name = "Nvp To Name";
	value = GetToField();
	inMsg.AddNvp(name, value);

	name = "Nvp From Name";
	value = GetFromField();
	inMsg.AddNvp(name, value);

	name = "Nvp Subject Name";
	value = GetSubjectField();
	inMsg.AddNvp(name, value);

	raBuffer binValue;
	name = "Nvp Body Name";
	char *msgBody = GetBody();
	binValue.SetData((unsigned char*)msgBody, strlen(msgBody));
	inMsg.AddNvp(name, binValue);

}


// Indicates if there are messages waiting
// to be added to the input tray
//////////////////////////////////////////
//////////////////////////////////////////
bool InputConnectorSample::ExistNewMessage ()
{
	// the ExistNewMessage() function should look into
	// the external message source
	return (!ExternalMessageSource.IsEmpty());
}



const raString dataBaseName("db name");
const raString dataBaseSource("db source");
const raString userName("name");
const raString password("pwd");
const raString instalationName("inst");
```

```
// main :
////////////
//     The following must be done while using raInputConnectorFramework :
//     - Construct an raInstallation
//     - Construct a new input connecotr object (of a type InputConnectorSample)
//     - Call raApplicationFramework::Run()
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
void main(int argc, char* argv[])
{

       try {

                     // Construct an raInstallation
                     // (With database parameters and process name)
                     ////////////////////////////////////////////////////////////
                     ////////////////////////////////////////////////////////////
                     raInstallation inst ( dataBaseSource,
                                                 dataBaseName,
                                                 userName,
                                                 password,
                                                 instalationName,
                                                 "SampleCodeInputConnector");


                     // Construct an active input connector of type
                     // InputConnectorSample
                     ////////////////////////////////////////////////////////////
                     ////////////////////////////////////////////////////////////
                     InputConnectorSample inputConnector(inst);

                     // Call raApplicationFramework::Run()
                     ////////////////////////////////////////////////////////
                     ////////////////////////////////////////////////////////
                     inputConnector.Run();


       } catch (raxException &x) {

                     inst.GetTrace().Put(       raTrace::S_ERROR,
                            "Fatal error: %s\n",
                            x.GetText().GetString());
       }

       inst.GetTrace().Put(raTrace::INFO, "Quit Sample Code Input Connector");

}
```

Output Connector

```
#include "Bro.h"
#include "raOutputConnectorFramework.h"
#include "raInstallation.h"
#include "raOutMessage.h"

// The output connector class :
//////////////////////////
//    - Derived from raOutputConnectorFramework
//    - Includes implementation of virtual functions ExtractMessage().
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
class OutputTest : public raOutputConnectorFramework
{
public:
      // Constructor
      ///////////////
      ///////////////
      OutputTest(raInstallation& inst) : raOutputConnectorFramework(inst){};

      // Process the message extracted from the output tray by Work()
      // according to the specific output connector needs.
      virtual void ExtractMessage(const raOutMessage& outMessage);
};


// The following function
// Process the message extracted from the output tray by Work(),
// according to the specific output connector needs.
////////////////////////////////////////////////////////////
// Should contain specific output connector code for extracting
// one message (outMessge)
// and usually format it and deliver it to an external entity.
//
// It must be implemented while using raInputConnectorFramework ,
// But this implementation is only an example.
// For different output connector the implementation will be different.
//
// This function demonstrate an NVP manipulation
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
void OutputTest::ExtractMessage (const raOutMessage& outMessage)
{
      raString nvpName;
      raMessage::NvpDataType nvpDataType;
      int intValue;
      raString stringValue;
      raBuffer binaryBalue;


      // Retrieve all NVP's from the outMessage
      ///////////////////////////////////////////
      ///////////////////////////////////////////
      raMessage::nvp_iterator iterNvp = outMessage.GetAllNvps();
```

```
// Message NVP's iteration
//////////////////////////
//////////////////////////
for ( ; !iterNvp.IsAtEnd() ; iterNvp ++)
{

      nvpName = iterNvp.GetName();
      nvpDataType = outMessage.GetNvpType(nvpName);


      // Multiple valued NVP
      ///////////////////////
      ///////////////////////
      if ( outMessage.IsNvpMultiple(nvpName) )
      {


            raMessage::multiple_value_iterator iterMulti;
            iterMulti = iterNvp.GetMultipleValue();


            // Multiple values iteration
            ////////////////////////////
            ////////////////////////////
            for (; !iterMulti.IsAtEnd(); ++iterMulti)
            {
                  switch (nvpDataType)
                  {
                  // Integer value
                  ////////////////
                  case raMessage::DT_DATATYPE_INTEGER :
                     {
                           iterMulti.GetValue(intValue);

                           printf("Ordinal : %d Value: %d /n",
                                 iterMulti.GetOrdinal(), intValue);
                           break;
                     }
                  // String value
                  ////////////////
                  case raMessage::DT_DATATYPE_STRING :
                     {
                           iterMulti.GetValue(stringValue);

                           printf("Ordinal : %d Value: %s /n",
                                 iterMulti.GetOrdinal(), intValue);
                           break;
                     }
                  // binary value
                  ////////////////
                  case raMessage::DT_DATATYPE_BINARY :
                     {
                           iterMulti.GetValue(binaryValue);

                           break;
```

```
                        }
                }
        }

        // Single valued NVP
        ////////////////////
        ////////////////////
        else
        {
                switch (nvpDataType)
                {
                      // Integer value
                      ////////////////
                case raMessage::DT_DATATYPE_INTEGER :
                   {
                         iterNvp.GetValue(intValue);

                         printf("Value : %d", intValue);
                         break;
                   }
                      // String value
                      ////////////////
                case raMessage::DT_DATATYPE_STRING :
                   {
                         iterNvp.GetValue(stringValue);

                         printf("Value : %s", stringValue.GetString());
                         break;
                   }
                      // binary value
                      ////////////////
                case raMessage::DT_DATATYPE_BINARY :
                   {
                         iterNvp.GetValue(binaryValue);

                         break;
                   }
        }
}


// Retrieve specific values from the outMessage
///////////////////////////////////////////////
///////////////////////////////////////////////

raString value;

// Retrieve "From" address from the outMessage
///////////////////////////////////////////////
///////////////////////////////////////////////
value  = outMessage.GetOutputFrom ();

printf("From: %s\n", value.GetString());
```

```
      // Retrieve "Subject" from the outMessage
      ////////////////////////////////////////////
      ////////////////////////////////////////////
      value = outMsg.GetOutputSubject ();

      printf("Subject: %s\n", value.GetString());



      // Retrieve "To" address from the outMessage
      // (Multiple Valued)
      ////////////////////////////////////////////
      ////////////////////////////////////////////
      printf("To: ");

      raMessage::multiple_value_iterator iterTos;
      iterTos = outMessage.GetOutputTos()

      for (; !iterTos.IsAtEnd(); ++iterTos)
      {
            iterTos.GetValue(value);
            printf("Ordinal : %d Value: %s /n",iterTos.GetOrdinal(),
                  value.GetString());
      }

      // Retrieve "Body"  from the outMessage
      ////////////////////////////////////////////
      ////////////////////////////////////////////
      raBuffer binValue;
      binValue = outMessage.GetOutputBody ();
      // Binary data cannot be displayed.


}


const raString dataBaseName("db name");
const raString dataBaseSource("db source");
const raString userName("name");
const raString password("pwd");
const raString instalationName("inst");


// main :
/////////
//    The following must be done while using raOutputConnectorFramework :
//    - Construct an raInstallation
//    - Construct a new Output connector object
//      (of a type OutputConnectorSample)
//    - Call raApplicationFramework::Run()
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
void main(int argc, char* argv[])
{

      try {
```

```
                // Construct an raInstallation
                // (With database parameters and process name)
                ////////////////////////////////////////////
                ////////////////////////////////////////////
                raInstallation inst ( dataBaseSource,
                                      dataBaseName,
                                      userName,
                                      password,
                                      instalationName,
                                      "SampleCodeOutputConnector");


                // Construct an output connector object
                // of type OutputConnectorSample
                ////////////////////////////////////////////////////////
                ////////////////////////////////////////////////////////
                OutputConnectorSample OutputConnector(inst);

                // Call raApplicationFramework::Run()
                ////////////////////////////////////////////////////
                ////////////////////////////////////////////////////
                OutputConnector.Run();


      } catch (raxException &x) {

                inst.GetTrace().Put(    raTrace::S_ERROR,
                                                "Fatal error: %s\n",

      x.GetText().GetString());
      }

      inst.GetTrace().Put(raTrace::INFO, "Quit Sample Code Output
Connector");

}
```

Active Queue Application

```
#include "Bro.h"
#include "raActiveQueueFramework.h"
#include "raInstallation.h"
#include "raActiveMessage.h"


// The active queue class :
//////////////////////////////////////
//    - Derived from raActiveQueueFramework
//    - Includes implementation of virtual function ProcessMessage();
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
class ActiveQueueSample : public raActiveQueueFramework
{
public:
      // Constructor
      ////////////////////
      ////////////////////
      ActiveQueueSample (raInstallation &conn, const raString &queueName)
                                          : raActiveQueueFramework (conn,
                                                                queueName) {};

private:
      // ProcessMessage()
      // the specific active queue code
      // for handling one message.
      ////////////////////////////////////////////
      ////////////////////////////////////////////
      virtual void ProcessMessage (raActiveMessage& message);
};



// Processes one message
// that was pending in a queue and retrieved by Work().
////////////////////////////////////////////////////////////////////////////
// Contains specific active queue code for handling one message (msg).
// It must be implemented while using raActiveQueueFramework ,
// But this implementation is only an example.
// For different active queue the implementation will be different.
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
void ActiveQueueSample::ProcessMessage (raActiveMessage &msg)
{
```

```
      // An example for ProcessMessage function implementation :
      // (For different active queue this function will look differently)

      // Put a Trace log
      ////////////////////////
      ////////////////////////
      GetInstallation().GetTrace().Put (raTrace::S_INFO,
                                  "In ProcessMessage(): Processing messages %s...",
                                  msg.GetTrackingId().GetString());

      // Add string NVP
      ////////////////////////
      ////////////////////////
      msg.AddNvp("NVP_FOR_ACTIVEQ","This is an example to NVP addition");


      raString closeOrSend;

      // Get string NVP
      ////////////////////////
      ////////////////////////
      msg.GetNvp("CloseOrSend", closeOrSend);


      if ( closeOrSend == raString("Close") ){

            // Close message
            ////////////////////////
            ////////////////////////
            msg.Close();
      }
      else if ( closeOrSend == raString("Close") ){

            // Send message
            ////////////////////////
            ////////////////////////
            msg.Send();
      }
      else {

            // Move message to queue, for agent handling
            ////////////////////////////////////////////////////////////////
            ////////////////////////////////////////////////////////////////
            msg.MoveToQueue("QueueForHumanHandling");
      }

}

const raString dataBaseName("db name");
const raString dataBaseSource("db source");
const raString userName("name");
const raString password("pwd");
const raString instalationName("inst");
```

```
// main :
////////
//    The following must be done while using raActiveQueueFramework :
//    - Construct an raInstallation
//    - Construct a new active queue object (of a type ActiveQueueSample)
//    - Call raApplicationFramework::Run()
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
void main(int argc, char *argv[])
{

      try {

                  // Construct an raInstallation
                  // (With database parameters and process name)
                  ////////////////////////////////////////////////////
                  ////////////////////////////////////////////////////
                  raInstallation inst ( dataBaseSource,
                                        dataBaseName,
                                        userName,
                                        password,
                                        instalationName,
                                        "SampleCodeActiveQueue");


                  // Construct an active queue object of
                  // type ActiveQueueSample
                  // (With the raInstallation constructed before,
                  //    and the name of the queue this active queue
                  //    application will poll)
                  ////////////////////////////////////////////////////
                  ////////////////////////////////////////////////////
                  ActiveQueueSample activeQueue (inst, queueName);



                  // Call raApplicationFramework::Run()
                  ////////////////////////////////////////////////////
                  ////////////////////////////////////////////////////
                  activeQueue.Run();



      } catch (raxException &x) {

inst.GetTrace().Put(    raTrace::S_ERROR,
"Fatal error: %s\n",
x.GetText().GetString());
      }

      inst.GetTrace().Put(raTrace::INFO, "Quit Sample Code Active Queue");
}
```