PONTON X/P Adapter Programming Guide

Version: 9
Date: 02-Apr-2019

Table of Contents

1. Introduction

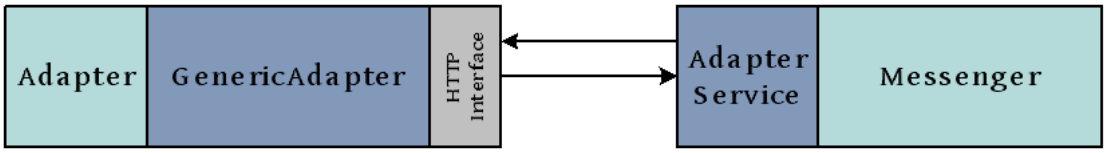
This document describes the basics how to write an Adapter for the Ponton X/P Messenger.

If you plan to implement your own Adapter, you should be familiar with object-oriented programming, inheritance, interface implementation, and the Java programming language.

Every Adapter has to implement the interface **ISpecificAdapter** from the package **de.pontonconsulting.xmlpipe.adapter**. This interface contains several methods that have to be implemented by the Adapter programmer.

Other essential classes are **GenericAdapter**, **MessageResult** from the package **de.pontonconsulting.xmlpipe.adapter** and **BackEndMessage** from the **package de.pontonconsulting.xmlpipe.message**.

1.1. Message Flow



The communication between the Messenger and an Adapter are standard HTTP calls.

All the data sent from the Adapter to the Messenger is sent by the GenericAdapter to the AdapterService servlet running in the same webserver as the Messenger. The Adapter calls methods in its GenericAdapter instance to invoke the sending of data.

Data from the Messenger to the Adapter is sent to the HttpInterface started by the GenericAdapter instance of the Adapter. The GenericAdapter reconstructs the data and calls the defined methods of the Adapter.

2. Important classes

Each adapter for Ponton X/P must implement the interface ISpecificAdapter, additionally there are some optional interfaces that can also be implemented to use more functionality. The outbound communication is done using an instance of the GenericAdapter.

2.1. Interface ISpecificAdapter

The following methods need to be implemented by the Adapter programmer.

```
public String getID()
```

This method should return the identifier of the Adapter. It will be passed to the messenger on Adapter registration. The identifier should be like 'AdapterName-IdNumber' and has to be unique in the whole messenger pipeline.

```
public String getStatus()
```

This method should return the status of the Adapter in human readable text form – like 'The Adapter is ready to receive messages'.

```
public boolean supportsAcknowledgements()
```

If the Adapter is able to handle acknowledgement, this method has to return the boolean true. The messenger will forward all incoming acknowledgements to the Adapter in that case. The Adapter must be able to associate the incoming acknowledgements to the previous sent message, since this is an asynchronous callback.

```
public boolean supportsAttachments()
```

If the Adapter supports attachments, this method should return the boolean true. The messenger will strip off any attachments from messages containing attachment, when this method return false.

```
public int getNumberOfParallelThreads()
```

This method should return the maximum number of parallel receiving threads, if the Adapter supports multithreaded receiving. If the Adapter does not support multithreaded receiving, this method should return the int 1.

```
public MessageResult receiveMessage(BackEndMessage message)
```

This method is called when a new Message is received by the GenericAdapter.

```
public MessageResult receiveTestMessage(BackEndMessage message)
```

This method is called when a new test message is received by the GenericAdapter. It is not required that the adapter performs any special treatments to test messages – instead the usual receiveMessage method might be called. This method just helps to recognise test messages without parsing them.

```
public MessageResult receiveAcknowledgement(BackEndMessage message)
```

This method is called when an acknowledgement is received by the GenericAdapter. But the method is only called if supportsAcknowlegdements returns true.

```
public File getWorkFolder()
```

This method returns the work-folder of the adapter. When a message contains attachments, the GenericAdapter will save them into a subfolder (named like the message ID) of the work folder. HttpRequests from the messenger will be saved into this folder temporary.

```
public boolean doSelfCheck()
```

This method is called to test the adapters special functions. It should return true, if the adapter is working properly.

```
public String shutdown()
```

When this method is called, the adapter should perform a clean shutdown and free all used resources. It can return a shutdown message to the messenger.

2.2. Interface IMessageStatusHandler

The EbXML standard allows querying a message service for the status of specific messages. If the adapter wants to use the sendStatusRequest() method of GenericAdapter, it has to implement the IMessageStatusHandler interface to be able to receive the statusResponse.

```
public MessageResult receiveStatusResponse(BackEndMessage message)
```

This method is called when a Status Response is received from the remote message service.

Please note that the Status Request feature is optional in the EbXML specification, so there could be an error returned if the partners message service does not support Status Requests.

The possible result codes of a Status Request could be:

- **UnAuthorized** – the Message Status Request is not authorized or accepted
- **NotRecognized** – the message is not recognized
- **Received** – the message has been received by the MSH
- **Processed** – the message has been processed by the MSH
- **Forwarded** – the message has been forwarded by the MSH to another MSH

2.3. Interface IPartnerEventListener

Adapters can be notified about changes in partner settings. If the adapter should be notified, it has to implement the IPartnerEventListener interface.

```
public void partnerAdded(String partnerId, String displayName, boolean
isLocalPartner)
```

This method is called whenever a new partner profile was created in the user interface.

```
public void partnerDeleted(String partnerId, String displayName, boolean
isLocalPartner)
```

This method is called whenever a partner profile was deleted in the user interface.

```
public void partnerModified(String partnerId, String oldPartnerId, String
displayName, boolean isLocalPartner)
```

This method is called whenever a partner profile changed. This can be caused by manual updates in the user interface, but also due to automatic profile updates from the registry.

2.4. Interface IAgreementEventListener

The Messenger can notify adapters about changes in agreements. If an adapter should get these notifications, it has to implement the IAgreementEventListener interface.

```
public void agreementAdded(String localPartnerId, String remotePartnerId)
```

This method is called whenever a new agreement is created in the user interface. The internal partner Ids of the involved parties are provided.

```
public void agreementDeleted(String localPartnerId, String remotePartnerId)
```

This method is called whenever an agreement is deleted in the user interface. The internal partner Ids of the involved parties are provided.

```
public void agreementModified(String localPartnerId, String
remotePartnerId)
```

This method is called whenever an agreement is modified in the user interface. The internal partner Ids of the involved parties are provided.

2.5. GenericAdapter

When starting an adapter, it has to create an instance of the GenericAdapter. The GenericAdapter handles the communication between the adapter and the Messenger. On start-up it will register the Adapter at the Messenger.

2.5.1. Initialisation

To create the GenericAdapter instance it is recommended to use the following constructor: public GenericAdapter(String logCategory)

```
// Create the GenericAdapter instance which manages connections between the
// Messenger and the Adapter.
// The Adapter will be registered with the Messenger when the
// addMessengerConnection() method is called.
// The Messenger is contacted on the given port number

GenericAdapter ga;

ga = new GenericAdapter( getID() );
ga.setEndAdapter( this );
ga.setServerPort( adapterPort );
ga.addMessengerConnection( "localhost", 8080, "/pontonxp/AdapterService");
```

To stop the GenericAdapter use the method **GenericAdapter.shutdown()**.

2.5.2. Partner Queries

To check if a partner ID exists in the messenger partner configuration the method **partnerExists(String partnerId)** can be used. It will return a boolean.

To obtain the complete list of partner IDs the method **getFullPartnerList()** can be used. It will return a String-array with all configured partner IDs.

To get only the local partners use the method **getLocalPartnerList()**. To get only the remote partners use the method **getRemotePartnerList()**.

2.5.3. Sending of Messages

With the method **sendPing(String senderId, String receiverId)** it is possible to send an ebXML-Ping to a remote partner. This will only work, if the adapter supports acknowledgements, because the resulting ebXML-Pong message will be passed to the adapter as an acknowledgement. This method returns an instance of the MessageResult class. See the JavaDoc for information on the MessageResult class.

To send a message to a remote partner use the method **sendMessage(BackEndMessage message)**. This method will return a MessageResult, which contains the information about the local processing. If the message was processed successfully by the receiver is returned in the asynchronously received acknowledgement.

2.5.4. Querying the Message Archive

It is possible to fetch message content from the Messengers archive by using either

writeXmlDocumentTo(String messageId, OutputStream out)

or

writeHtmlDocumentTo(String messageId, OutputStream out).

Both methods will send a query based on the message id to the Messenger. The result will be written to the output stream that has to be provided as a parameter.

Depending on the method, the result will either be the raw payload (usually XML) or the HTML representation based on the raw payload.

2.6. BackEndMessage

When sending or receiving a message an instance of the BackEndMessage class is used. This class contains getter- and setter-Method for all elements and attributes in the BackEndMessage.

See the supplied BackEndEnvelope.xsd file for the complete structure of the BackEndMessage.

To create a BackEndMessage there are two constructors available. One takes a **java.io.File** referencing the XML message to be sent, the second takes also a **java.io.File** and a **Boolean** to define if the file is actually XML or something else.

The older constructors that take byte arrays as parameter are deprecated, since they are not compatible with very large messages; however they are still fully functional.

If the given XML file already contains a BackEndEnvelope, the BackEndMessage class will recognise it and use the value provided. If there is no BackendEnvelope in the File, then it might be needed to provide some data via method calls.

In general all values that are provided via method calls override data that were fetched from the optional BackEndEnvelope.

To add an attachment to the BackEndMessage, use the method **addAttachment(java.io.File file)**. This method will determine the mime-type of the attachment automatically. It is also possible to set the mime-type manually with the method **addAttachment(java.io.File file, String type)**. Additionally you can add a description and

specify the language of the attachment if you use the method **addAttachment(java.io.File file, String type, String description)** or **addAttachment(java.io.File file, String type, String description, String language)**.

When a message is received, you can access all fields in the BackEndEnvelope with the getter-methods. For all elements there are methods available which return the element content text, e.g. the method **getConversationIDText()** returns the content text of the element ConversationID as a String.

You can obtain the message content by calling the method **writeMessageDocumentTo(OutputStream)**. This method returns only the payload document. If you want to process the BackEndEnvelope externally, you can get the whole BackEndMessage, including the payload document with the BackEndEnvelope wrapped around it, by calling the method **writeBackEndMessageTo(OutputStream)**. There is also the possibility to store the BackEndEnvelope without the payload by using the method **writeBackEndEnvelopeTo(OutputStream)**.

If the received document contains attachments, you can get them by using the method **getAttachment(String filename)**. This method returns the file reference to the attachment in the work folder. The attachment has to be moved or copied out of the work folder before the receive-method of the adapter is returning, because the GenericAdapter will remove the temporary folder for this method afterwards. To get all filenames of attached files, use the method **listAttachments()**. You will get a String-array with the filenames of all attachments of this message.

2.7. MessageResult

The class MessageResult is used to pass send/receive results between the Messenger and an Adapter.

2.7.1. Outbound transmissions

When the Adapter sends out a message it will get a MessageResult back containing the result of the local processing. Additionally the MessageResult will contain the conversation ID, message ID and other message specific data. See the JavaDoc of the MessageResult class for complete information.

2.7.2. Inbound transmissions

When the Adapter receives a message, it has to provide a MessageResult so that the Messenger knows if the message was correctly processed.

To create a MessageResult you have to use the only constructor of the MessageResult, which takes a MessageResultIdentifier as argument. The MessageResultIdentifier is an inner class of the MessageResult.

```
MessageResult result = new
MessageResult(MessageResult.MSG_SUCCESSFULLY_RECEIVED);
```

In case of errors, the Messenger will try to resend the message.

If that is not wanted, for example because the adapter is not able to process the message due to the message contents itself, then it makes no sense to retry the transmission.

To prevent retransmission of messages the adapter has to return

```
MessageResult.ADAPTER_REJECTED_MESSAGE
```

This will result in a final state of FAILED. However it is still possible to manually retransmit the message using the MessageMonitor.

2.8. Acknowledgements

For any outbound transmission an asynchronous XpAcknowledgement XML response will be generated by the X/P Messenger.
This XpAcknowledgement is delivered as BackendMessage Object parameter of the **receiveAcknowledgement(BackEndMessage message)** method.

The XpAcknowledgement contains one overall ResultCode at the XPATH **/XpAcknowledgment/OverallResultCode**

It also contains one or more detail results at XPATH **/XpAcknowledgement/Results/Code**
The possible value range is:

- Success
- Warning
- Error
- Pong
- StatusResponse

A describing Text is available at XPATH **/XpAcknowledgement/Results/Description**

3. MessageResult Codes

3.1. GenericAdapter.sendMessage()

3.1.1. MSG_SUCCESSFULLY_SEND

The Messenger completed all processing steps and the message is queued for delivery. It is not transmitted to the receiver at this point !

From the returned MessageResult object the adapter can get several information:

String getMessageID() this is the message id (transfer id) that is used for transmission. If it was defined in the backend message, then it should be unchanged.

String getConversationID() this is the conversation id that is used for transmission. If it was defined in the backend message, then it should be unchanged.

String getMessageType() the message type that was identified by the messenger

String getSchemaVersion() the schema version that was identified by the messenger.

String getSchemaSet() the schema set that was used for validationString String getMessageTime() the creation timestamp that is transmitted in the transport envelope

String getTransmissionProtocol() the protocol that is used to send the message

3.1.2. ADAPTER_REGISTRY_COULD_NOT_BE_ACCESSED

This error is returned if an adapter tries to send a message. It can only happen if the Messenger was restarted while the adapter kept running. If the adapter cannot be re-registered in the messenger database this error will occur.

3.1.3. BACKENDMSG_COULD_NOT_BE_RECONSTRUCTED

When unexpected or corrupted data is send to the Messenger, this error will be returned.

String getResponseMessage() this contains further details about the error

3.1.4. MESSAGE_COULD_NOT_BE_REGISTERED

The message ID is already registered at the Messenger. It is not allowed to send two messages with the same ID. if the previous transmission failed for any reason, then the adapter is allowed to resend the same message (with the same message ID) so that the transmission is triggered again.

A second cause for this error is a database communication problem.

3.1.5. PARTNER_IS_DISABLED

The receiver or sender of this message is deactivated in the partner config of the Messenger.

String getResponseMessage() contains the local id of the blocked partner. And also states if it is the receiver or the sender.

3.1.6. PARTNER_IS_NOT_KNOWN

The local id of the receiver or sender of this message is unknown. The partner settings should be checked for the correct local id.

String getResponseMessage() contains the local id that was not found. It also states if it is the receiver or the sender.

3.1.7. PARTNER_STORE_COULD_NOT_BE_ACCESSED

The partners.xml file got corrupted or is not accessible.

String getResponseMessage() contains additional error information.

3.1.8. TRANSPORT_PROVIDER_NOT_FOUND

The URL defined for the receiver specifies an unknown protocol. Currently only http:// https:// mailto:// smime:// are supported.

This has to be fixed in the partners configuration.

3.1.9. COULD_NOT_DECRYPT_PRIVATE_KEY_PASSWORD

The private key to sign messages is not found or cannot be decrypted. This is either because the specified sender is not a local partner or some problem with the private key password exists.

String getResponseMessage() contains additional error information.

3.1.10. COULD_NOT_INITIALIZE_PIPELINE

There was a problem while initializing the processing pipeline.

String getResponseMessage() contains additional error information.

3.1.11. ENCRYPTION_FAILED

Problem while encrypting the payload.

String getResponseMessage() contains additional error information.

3.1.12. SIGNING_FAILED

Problem while signing the payload.

String getResponseMessage() contains additional error information.

3.1.13. COMPRESSION_FAILED

Problem while compressing the payload.

String getResponseMessage() contains additional error information.

3.1.14. VALIDATION_FAILED

Problem while validating the payload.

String getResponseMessage() contains additional error information.

3.1.15. COULD_NOT_PROCESS_MESSAGE

The message type and or version are not allowed by either the receiver or sender. Check the partner configuration to see if the correct schema-sets are activated for both partners.

Any other problem while processing the message can also result in this error.

String getResponseMessage() contains additional error information.

3.1.16. LOGGING_INTO_DATABASE_FAILED

The message was completely processed but an error occurred when the message was queued for delivery.

String getResponseMessage() contains additional error information.

3.1.17. COULD_NOT_INITIALIZE_PACKAGER

A problem with the messenger configuration caused this internal problem.

String getResponseMessage() contains additional error information.

3.1.18. COULD_NOT_PACKAGE_MESSAGE

The transmission format could not be created. This can be caused by invalid or missing message properties. It can also be caused by file system errors.

String getResponseMessage() contains additional error information.

3.1.19. DUPLICATE_MESSAGE_ID

The message id that was provided was already used for a successful message. Sending a second message with the same ID would result in unpredictable message transfers. So if for some reason the same ID needs to be transmitted, the old message with the same ID has to be manually deleted in the MessageMonitor first.

3.2. GenericAdapter.sendPing()

Same as for sendMessage()

3.3. GenericAdapter.shutdown()

3.3.1. ADAPTER_SUCCESSFULLY_UNREGISTERED

Everything is ok.

3.3.2. ADAPTER_COULD_NOT_BE_UNREGISTERED

A communication problem with the messenger occurred.

3.3.3. ADAPTER_REGISTRY_COULD_NOT_BE_ACCESSED

A database problem prevented the deregistration

3.4. ISpecificAdapter.receive…()

All receive methods can return the same MessageResults:

3.4.1. MSG_SUCCESSFULLY_RECEIVED

The Messenger will remove the inbound message from the queue and flag it as successfully completed.

Before returning the MessageResult object it is possible to give additional text to the Messenger using **appendToDescription()** or **setDescription()**.

The later will override the default text that comes with the selected MessageResult.

3.4.2. COULD_NOT_ACCESS_ATTACHMENT_FILE

The Messenger will store the reported error but it will try to send the same message again.

3.4.3. CUSTOM_ERROR

The Messenger will store the reported error but it will try to send the same message again.

3.4.4. ADAPTER_REJECTED_MESSAGE

The Messenger will remove the inbound message from the queue and flag it as ERROR.

Another try to deliver this message is possible by selecting “change adapter” on the message monitor.

Before returning the MessageResult object it is possible to give additional text to the Messenger using **appendToDescription()** or **setDescription()**.

The later will override the default text that comes with the selected MessageResult.

PONTON GmbH
Dorotheenstraße 64
GERMANY 22301 Hamburg
Web:http://www.ponton.de

4. Sample Adapter Source Code

```
package de.pontonconsulting.xmlpipe.adapter.sampleadapter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import de.pontonconsulting.xmlpipe.adapter.AdapterException;
import de.pontonconsulting.xmlpipe.adapter.GenericAdapter;
import de.pontonconsulting.xmlpipe.adapter.ISpecificAdapter;
import de.pontonconsulting.xmlpipe.adapter.MessageResult;
import de.pontonconsulting.xmlpipe.message.BackEndMessage;
import de.pontonconsulting.xmlpipe.message.BackEndMessageException;


/**
* This is a sample implementation of an adapter for Ponton X/P.
*
* Copyright © 2016 Ponton Consulting GmbH. All rights reserved.
*/
public class SampleAdapter implements ISpecificAdapter {

private Log _log;
private GenericAdapter _ga;

public static void main(String[] args) {
     SampleAdapter adapter = new SampleAdapter();
     adapter.partnerExistenceTest();
     adapter.getPartnerlistTest();
     adapter.sendTest();
     adapter.shutdown();
}



/**
 * Create the SampleAdapter instance.
 */
public SampleAdapter() {
     try {
            System.setProperty("org.apache.commons.logging.LogFactory",
"de.pontonconsulting.common.log.PontonLogFactory");
            _log = LogFactory.getFactory().getInstance(getID() +
"/Main/console=DEBUG");
            _log.debug("Initializing SampleAdapter");
            // Create the GenericAdapter instance which provides the connection
            _ga = new GenericAdapter(getID());
            _ga.setServerPort(0); // Use -1 if you don't want to receive any
messages.
            // Use 0 if any free port should be used. Default is 0.
            _ga.setProcessingTimeout(1200000); // Default is 600000 (10
minutes).
            _ga.setAdapterIP("127.0.0.1"); // This IP address is used by the
messenger to // send incoming messages to the adapter.
            // If the IP address is not explicitly set, // the first found IP
```

```
will be used.
            _ga.setEndAdapter(this);
            _ga.addMessengerConnection("localhost", 8080,
"/pontonxp/AdapterService"); // Add more messenger connections, if you use
cluster mode.
      } catch (AdapterException ae) {
            _log.fatal("Error while initializing SampleAdapter", ae);
      } catch (IOException ioe) {
            _log.fatal("Error while initializing SampleAdapter", ioe);
      }
}


public int getNumberOfParallelThreads() {
      return 1;
}


public MessageResult receiveMessage(BackEndMessage message) {
      // There are several getter methods to obtain the data from the
BackEndMessage.
      // For attributes, these methods return a java.lang.String.
      // For elements, there are methods that return an org.dom4j.Element and
methods
      // that return a java.lang.String.
      _log.info("*** Receiving message: " + message.getTransferIDText());
      _log.info(" ** Conversation id: " + message.getConversationIDText());
      _log.info(" ** Local sender id: " + message.getSenderOrganisationText());
      _log.info(" ** Local receiver id: " +
message.getReceiverOrganisationText());
      _log.info("*** # of attachments: " + message.getNumberOfAttachments());
      String[] attachments = message.listAttachments();
      for (int i = 0; i < attachments.length; i++) {
            File attachment = message.getAttachment(attachments[i]);
            _log.info(" --> Attachment: " + attachment.getAbsolutePath());
            // All attachments have to be moved to a different folder before
the MessageResult
            // is returned, because the GenericAdapter will delete these from
the work folder.
      }
      /* get the business document as input stream */
      message.getMessageDocumentInputStream();
      /* save the business document as file */
      message.writeMessageDocumentTo(new File("payload.xml"));
      /* save the business document including the backend envelope as file */
      message.writeBackEndMessageTo(new File("full.xml"));
      /* save the backend envelope as file */
      message.writeBackEndEnvelopeTo(new File("envelope.xml"));


      return new MessageResult(MessageResult.MSG_SUCCESSFULLY_RECEIVED);
}


/**
 * This method will call the method receiveMessage(BackEndMessage), because
this
```

```
 * Adapter does not distinguish between 'test' and 'production' messages.
 */
public MessageResult receiveTestMessage(BackEndMessage message) {
     return receiveMessage(message);
}


/**
 * Get the MessageResult (XML acknowledgement for the message).
 */
public MessageResult receiveAcknowledgement(BackEndMessage message) {
     /*
     * The XML acknowledgement as a byte array is accessible via the method
     * message.getMessageDocumentBytes()</code>
     */
     return new MessageResult(MessageResult.MSG_SUCCESSFULLY_RECEIVED);
}


public boolean doSelfCheck() {
     return true;
}


public String getID() {
     return "sample-adapter";
}


public String getStatus() {
     return "SampleAdapter is ready to receive Messages.";
}


public String shutdown() {
     _ga.shutdown();
     return null;
}


public File getWorkFolder() {
     return new File("");
}


public boolean supportsAcknowledgements() {
     return false;
}


public boolean supportsAttachments() {
     return true;
}


/**
 * Send a message to a partner.
```

```
 */
public void sendTest() {
     try {
            // Create a new BackEndMessage from the XML file.
            BackEndMessage bem = new BackEndMessage(new
File("webroot/.../BE_PN21_InfoRequest_mini.xml"));
            // Set the sender and receiver organisation. These values have to
match
            // internal IDs of partner profiles.
            // If the specified XML file already contains a BackEndMessage and
the
            // sender and receiver information is set correctly,
            // these values don't have to be set manually.
            bem.setSenderOrganisation("sender");
            bem.setReceiverOrganisation("receiver");
            // optionally set specific Message Type.
            bem.setDTDSet("papinet2.1");
            bem.setMessageName("PurchaseOrder");
            bem.setDTDVersionNumber("V2R10");
            MessageResult result = _ga.sendMessage(bem);
            if (!result.equals(new
MessageResult(MessageResult.MSG_SUCCESSFULLY_SEND))) {
                  _log.error("transmission failed:" + result.toString());
            }
     } catch (FileNotFoundException e) {
            e.printStackTrace();
     } catch (AdapterException e){
            e.printStackTrace();
     } catch (BackEndMessageException bme) {
            bme.printStackTrace();
     }
}


/**
 * This test checks whether partners exist.
 */
public void partnerExistenceTest() {
     try {
            _log.info("Partner 'papitest' exists: " +
_ga.partnerExists("papitest"));
            _log.info("Partner 'papitest2' exists: " +
_ga.partnerExists("papitest2"));
            _log.info("Partner 'papitest234' exists: " +
_ga.partnerExists("papitest234"));
     } catch (AdapterException e) {
            e.printStackTrace();
     }
}


/**
 * This test gets the configured partner list from the Messenger.
 */
public void getPartnerlistTest() {
     try {
            String[] partners = _ga.getFullPartnerList();
```

```
            for (int i = 0; i < partners.length; i++) {
                  _log.info("*** (all) Partner " + i + " has local id: " +
partners[i]);
            }
            partners = _ga.getLocalPartnerList();
            for (int i = 0; i < partners.length; i++) {
                  _log.info("*** (own) Partner " + i + " has local id: " +
partners[i]);
            }
            partners = _ga.getRemotePartnerList();
            for (int i = 0; i < partners.length; i++) {
                  _log.info("*** (remote) Partner " + i + " has local id: " +
partners[i]);
            }
      } catch (AdapterException e) {
            e.printStackTrace();
      }
}
}
```

PONTON GmbH

Dorotheenstraße 64

GERMANY 22301 Hamburg

Web: http://www.ponton.de