$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

OpenEAI API Introduction
Version 1.0

January, 2003

by

Tod Jackson (tod@openeai.org)
Steve Wheat (steve@openeai.org)

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Contents

Introduction

Work on the OpenEAI API began in March 2001 at the University of Illinois. The purpose of this work was to provide foundation components with which to implement integrations using the OpenEAI Message Protocol. While the benefits of describing data using XML are widely understood and generally accepted today, the fact remains that building and manipulating XML documents is programmatically challenging for many developers, and gets really boring very fast for nearly all developers. For these reasons, the work is highly prone to error.

It would be ideal if developers did not have to use their knowledge of XML to programmatically build or manipulate every XML message at the document level. It would be ideal if developers did not have to use their knowledge of and the Java Message Service (or any other transport an

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

organization may need to employ in interfacing specific applications) to send, receive, or consume messages. If developers had a business-object-oriented API, with objects that looked like the enterprise data objects defined by an organization in its messages (such as BasicPerson, BasicEmployee, InstitutionalIdentity, DriversLicense, EmergencyContact, HonorAward, LicenseCertification, WorkHistory, AdmissionApplication, TestScore, or whatever else), they could simply focus on implementing the business logic provided to them by analysts (or that they came up with by working with the design team, depending upon how the organization performs analysis and development).

At its heart, the OpenEAI API ***is*** this business-object-oriented API for building integrations. Specifically, this business-object-oriented API is the OpenEAI Message Object API, referred to as the MOA. OpenEAI Java Message Objects know how to build themselves from XML and serialize themselves to XML. They know how to build all the messages specified by the OpenEAI Message Protocol and send them using a JMS provider. They know how to translate the application-specific values for data fields for application X and translate them to enterprise values. They know how to take enterprise values and translate them to appropriate application-specific values for application Y. They know how to enforce the presence of required fields. They know how to enforce formatting masks. They know how to execute scrubbers on data to address complex transformation and formatting that may not be so easily generalized. And they know how to do a lot more.

Since business objects are most often specific to an organization, each organization is going to have its own message object definitions and Java Message Objects or MOA. Sure, some organizations and industries will standardize and agree to use common definitions for some business objects, and some already have. However, for the vast majority of data that organizations need to pass between their own systems, there are no standard definitions, and organizations need to work quickly to meet their own requirements and internal deadlines. It would be great if all the complex EAI and JMS capabilities of message objects could be inherited from OpenEAI foundation objects, making OpenEAI Java Message Objects easy to implement, matching each organization's message object structures.

Well, this is exactly how it works! As it turns out, it's even better. All of the JMS and EAI behavior of Java Message Objects is implemented in ancestor classes provided in the OpenEAI foundation. The Java Message Objects that must be developed for an organization's own MOA are so straightforward that they can be programmatically generated from XML message definitions by an MOA generation application provided by the OpenEAI Project. So, the development of a business-object-oriented API becomes a *non-step* in the integration project lifecycle. It's simply a byproduct of defining enterprise data objects in XML.

The rest of the OpenEAI foundation components were developed over time to facilitate the use of these business objects in one way or another. Layout managers were developed to implement serialization of these objects to and from various formats such as XML, flat files, and stored procedure calls. Enterprise field foundation was implemented to allow analysts to specify enterprise values and application-specific values and to perform translations to and from these values. Scheduled application and message consumer client foundation was implemented to provide generic, runnable applications that serve as containers for specialized commands. These commands are where the business logic of integrations is implemented using the XML-, JMS-, and EAI-aware business objects. Application configuration foundation and a common deployment descriptor were developed to allow developers and deployers to configure messaging applications systematically, and to enable contemporary deployment patterns, such as storing the configuration files for distributed applications that may run in many instances on many servers in a central directory server or web server and retrieving them via a secure protocol at runtime.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

The OpenEAI API is an especially critical aspect of the OpenEAI Project. The OpenEAI Message Protocol, Methodology, and Implementation Strategies could be used by themselves. They are concepts that have proven useful in a number of ways for thinking about the dynamics of integration applications, for performing integration analysis, and for organizing an enterprise that is integrated using messaging. However, the fact that the OpenEAI Project takes these ideas and provides a set of APIs to implement them makes the protocol, methodology, and implementation strategies so effective. We can learn much from actually implementing these concepts and building real integrations to run our organizations, and what we learn can be used to refine and extend the underlying theory.

The OpenEAI API for implementing enterprise integrations is based on several other architectures, concepts, and foundation. These include:

- Java 2 Platform Enterprise Edition, specifically core Java itself and specifications within the J2EE platform such as the Java Naming and Directory Interface (JNDI) and the Java Message Service (JMS).

- The Extensible Markup Language (XML) and specific XML parsing and manipulation foundation such as Xerces, Xalan, and JDOM.

Throughout this document, there are both code examples and XML examples. The OpenEAI Project also maintains reference implementations of real-world messaging applications that are used by one or more organizations to integrate production enterprises, as well as a complete sample messaging enterprise. The sample messaging enterprise is intended to serve as a leaning tool and pattern reference. It is intended for use in conjunction with the Getting Started with OpenEAI document. If you have not already read this document and followed through the example enterprise, you may want to peruse it quickly. It will help you focus in on the area of OpenEAI you want to explore first, depending on your role in your organization. The Getting Started document also serves as a guide to the OpenEAI Core Documentation Suite. You probably need to have a general understanding of the OpenEAI Message Protocol and OpenEAI methodology to best understand OpenEAI API.

The OpenEAI API can be classified into ten general areas of foundation. These are the areas and their corresponding package names.

1. Application foundation (org.openeai.afa)
2. Application configuration (org.openeai.config)
3. Enterprise Message Object API foundation (org.openeai.moa)
4. JMS Foundation (org.openeai.jms)
5. Enterprise Layout Manager foundation (org.openeai.layouts)
6. Enterprise Scrubber foundation (org.openeai.scrubbers)
7. Enterprise Database Connection pool foundation (org.openeai.dbpool)
8. ThreadPool foundation (org.openeai.threadpool)
9. XML Utilities (org.openeai.xml)
10. Reference implementations (org.openeai.implementations)

The official API documentation (javadoc) is available for download and online browsing. This document describes how components from these packages are used, and provides examples.

As we write, the current release of the OpenEAI API is 3.0 beta 2. The precursor of the OpenEAI API, the University of Illinois EAI API, was released internally at the University of Illinois in

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

December 2001 (release 1.0) and again in June 2002 (release 2.0), and is currently used in production at the University of Illinois. Both the University of Illinois and SCT Corp. are testing the 3.0 beta release of OpenEAI. As soon as a site that practices OpenEAI uses the 3.0 release satisfactorily in production, the project will provide a final 3.0 release. The OpenEAI Project promotes software from 'beta' to an official release only after it has both completed rigorous testing by the OpenEAI project and been successfully used in production by at least one site that practices OpenEAI.

The OpenEAI API is maintained by the OpenEAI Project (info@OpenEAI.org). This document was originally written by Tod Jackson (tod@openeai.org) and Steve Wheat (steve@openeai.org) in 2001 and 2002 and published in January 2003.

Helpful Definitions

This section provides brief definitions of some key terms that will be used throughout this document. Each of these will be described in more detail later on.

Application

An application will be involved in the production and/or consumption of enterprise messages. It will typically be the initiator of a messaging conversation. For example, an employee self-service application that requests emergency contact information from the enterprise's ERP system.

Scheduled Application

A scheduled application can start, execute some logic and exit, or can run as a daemon application that runs continuously and executes business logic on a configurable schedule. This is a common requirement for integration applications. For example, you may be familiar with requirements to check a directory for a URI for a file at a specified interval to see if a mainframe extract file has been dropped off. You may need to check a database table for changes or pending transactions every so often. When it comes to integration tasks, there are about a million things you could have to do on an interval-based schedule. After doing quite a few, it becomes clear that some integrations become much more efficient (and less latent) if you do not have to rely upon scheduling things every minute or every second. Optimally, you need the flexibility to schedule some things to happen within the bounds of a single second. The OpenEAI Scheduled Application foundation provides the ability to encapsulate business logic in individual components (commands). These commands can be executed according to a defined schedule associated with the application. This serves several purposes:

- Allows a generic "main" class for all applications that need to run in this fashion. This generalizes some of the standard application startup code, but more importantly, it allows a generalized startup mechanism for all applications, such as a common start script, service deployer, or administration console. This is an important consideration when deploying applications. It can become very confusing to support a different startup mechanism for each application if they all have a unique way to be configured and started.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

- Execute immediately and exit (type=Application). This means the application will execute just like any "normal" application one might think of. The application will start, it will execute the business logic associated to it immediately, and it will exit. This is similar to a normal Java class with a "main" routine.

- Execute immediately and go back to sleep (type=Triggered). This means the application will start, execute the business logic associated to it immediately, and then wait for a signal to tell it to stop. This is useful if you want the business logic itself to have more control over the life of the application.

- Execute on a given day(s) at a given time(s) (type=Daemon). This means the application will start, then utilize a flexible scheduling facility (which is part of the OpenEAI API) to determine when business logic should be executed. After it has executed that business logic, it will wait until the next scheduled initiation to execute the business logic again. This is useful for long-running applications that need to execute the same, or different, business logic over and over again on a regular interval.

Message Gateway

A message gateway is a daemon application that consumes messages in the publish/subscribe model, point-to-point model, or both. It is used to expose an existing application that is authoritative for some data to the rest of a messaging enterprise through request/reply messages or to consume synchronization messages from authoritative application to keep itself up to date.

Specifically, a message gateway may consume a request message, execute or invoke appropriate business logic to process the request message, and return the reply appropriate to the request it consumed. It may also need to publish a sync message appropriate to the action of the request. Alternately or in addition, it may consume synchronization messages, execute or invoke the appropriate business logic to process each sync message, and publish an Error-Sync message if it encounters errors processing a sync message it consumed.

See the section entitled "Implementing Message Support" in the OpenEAI Message Protocol Document for details on which messages a gateway may need to support, depending on whether it is or is not the application authoritative for one or more enterprise message objects.

Similar to the scheduled application foundation, the message gateway foundation provides the ability to encapsulate business logic, or the invocation of application business logic, in individual components (commands). Then, when messages are consumed, the appropriate command is executed to process each message, and it replies in the case of requests or publishes a Sync-Error message if a sync message is unsuccessfully processed.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Message Relay

A message relay is a useful infrastructure component for making applications message-aware that are JMS-unaware and/or XML-unaware. A message relay is typically a daemon application or servlet that serves as an intermediary between a JMS-unaware application and the rest of a JMS-aware messaging enterprise. It can relay messages between applications that are JMS-aware and those that may only be able to send or receive messages with other, more traditional transport protocols such as TCP, HTTP, and HTTPS. In addition to transport bridging, a relay can also be useful in bridging message protocols. Some technologies that cannot easily be made JMS-aware also cannot easily be made XML-aware. For more details on the concept and implementation of message relaying, see the OpenEAI Implementation Strategies Document.

Messaging Enterprise

The messaging enterprise is the combination of all messaging applications, gateways, relays, proxies, and other messaging infrastructure applications that are deployed to integrate and manage messaging within an organization.

Analysis Template

The analysis template is used to document integration analysis and define the enterprise messages needed for a particular integration. Additionally, it defines the production and consumption logic for those messages. This document must be completed before any serious development work can begin. See the OpenEAI Methodology Document for more details on the OpenEAI analysis template.

Deployment Descriptor

This is an XML document structure used to configure all messaging applications and gateways that use OpenEAI foundation components. This document is constrained according to the configuration options of the OpenEAI foundation components to provide a clear and uniform way to configure applications.

Enterprise Object Document

This is another XML document structure that OpenEAI Java Message Objects use to apply business rules to their data in their member fields. The rules are specified in enterprise object documents and implemented by the message objects when data is put into the member fields via setter methods.

Integration Process Overview

Details of the recommended OpenEAI integration process are covered in the OpenEAI Methodology Document. The following brief overview is intended to help place the use of the

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

OpenEAI foundation components in the context of the development, documentation, and testing phases of the integration process.

Perform Analysis

Once the systems that need to be integrated are identified, an analysis group or team comprised of business, functional and technical integration analysts complete the OpenEAI analysis template, or their organization's customized version of the template, for each application that must be interfaced. Among other things, the template documents general integration requirements, specifies which existing message objects will be used and which new message objects will be required, defines new message objects, specifies which message actions of the OpenEAI Message Protocol will be required for each message object, enumerates the messaging applications, gateways, and infrastructure that must be developed to implement the integrations, and enumerates detailed message production and consumption logic.

Define Messages

Once any new enterprise message objects and the message actions for them have been defined through the process of completing the OpenEAI analysis template, technical integration analysts create the XML message definitions for the new messages in the organization's message hierarchy and provide one sample message for each definition.

Generate Java Message Objects

Next, the message definitions are implemented as Java objects. A Java object must be created for every complex enterprise business object defined. These Java objects are automatically generated using the OpenEAI MoaGenApplication; message definitions are prepared by integration analysts.

Develop, Document, and Test Messaging Applications

The details of this phase will vary from organization to organization: many organizations already have application development and testing practices established. However an organization chooses to implement them, the following guidelines should be considered.

1. Developers and analysts prepare detailed, technical stories for each messaging application and gateway listed in the completed analysis. These stories will draw heavily on the message production and consumption logic prepared by the functional staff and analysts and included in the analysis template.

2. Developers implement the appropriate messaging applications and gateways listed in the template using OpenEAI foundation components, the message object API that was generated for the organizations enterprise message objects, and the enterprise object documents completed by the functional staff and analysts. When developing

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

an OpenEAI based application or gateway, this means developing the commands needed to support the processes defined in the analysis.

3. While steps one and two above are proceeding, integration analysis staff can prepare OpenEAI TestSuiteApplication test suite documents for testing the message gateways that are to be developed. Test suite documents are XML documents comprised of messaging test cases. The OpenEAI TestSuiteApplication can take a test suite document and execute all its test cases very rapidly: it sends test messages to a target application and compares the replies received and sync messages published by the target application with expected results, produces a detailed report, and can even tear-down any messaging artifacts or database entries created as a result of the TestSuite execution. Functional staff and analysts are typically the best equipped to prepare these test suite documents, because they are the people who specified the integration requirements and message production and consumption logic to begin with. Developers can perform this work, but if they do the functional staff and analysts should review and approve the test cases as being representative of the requirements.

   It's useful for developers to have these test cases available to them during the development process: as they implement message support they can iteratively execute the test suite using the OpenEAI TestSuiteApplication to check their progress and to verify that changes they make do not break anything.

   At this time, functional staff and analysts also prepare real-world online and batch scenarios to test the new messaging applications and gateways in integration with the rest of the messaging enterprise in a test environment.

4. All messaging applications and gateways pass both informal developer testing and all of the formal test suites executed by the TestSuiteApplication.

5. The new messaging applications and gateways are promoted from a development environment to a test environment for integration testing, and the real-world online and batch scenarios are executed until the functional staff and analysts are convinced the new applications are performing appropriately. Note that although the practice of preparing test suites and unit testing messaging applications can be very effective at ensuring that messaging applications perform as designed, it may not help validate that the design is correct. From time to time, integration testing with the rest of the messaging enterprise turns up new requirements, such as identifying additional applications that need to know about data from a new authoritative source. For this reason, integration testing is a critical part of the process. It is also the part of the testing process that gives management a tangible level of confidence that it is appropriate to proceed with a production implementation.

Update Enterprise Documentation Artifacts

Practicing the OpenEAI methodology produces a number of documentation artifacts, including an analysis template for each application that interfaces with other applications, enterprise data object definitions, message definitions, and javadoc for commands that implement support for each message object. These artifacts should be posted in a web-accessible format. For example, message definitions and enterprise object documents

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

should be posted on a web server so that messages can be validated when necessary and so that field-level business rules and translations can be applied by Java Message Objects at runtime. This practice allows you to build a web page to nicely document each messaging application or gateway, linking to and leveraging each of these artifacts that must be created anyway.

The OpenEAI Methodology Document recommends a format for these web pages. Just as your organization may customize the analysis template or the OpenEAI methodology itself, you will likely choose to customize the web documentation template as well. This web page can be posted on your organizational web site or intranet. Posting this documentation helps managers, functional analysts, and technical analysts plan and prepare for new integrations. Additionally, many organizations have auditing or best-practice requirements that mandate the preparation of some type of formal documentation for each integration. Posting this documentation as a web page makes this information available to those who need it. In some cases, it can even be helpful to complete and post most of this template prior to or during the development phase.

Deploy in Production

There's not much to say about this step from an overview perspective, since if you get to this point, most of the work has already been done. If you follow the recommended OpenEAI practices for testing in pre-production environments, deploying in production should be anticlimactic. The OpenEAI Deployment Patterns Document provides details on the minimum number of recommended environments you should set up for a messaging enterprise and how and when to promote messaging application and gateways from one environment to the next.

OpenEAI Foundation Components

JMS Foundation

This includes four types of messaging components specifically designed for JMS messaging. They include:

- PointToPoint producers for producing requests to JMS queues and handling a reply
- PubSub producers for publishing messages to JMS topics
- PointToPoint consumers for consuming requests from JMS queues and returning a reply
- PubSub consumers for consuming messages from JMS topics

These components will be used by applications and gateways to produce or publish to other applications in a messaging enterprise and to consume enterprise messages from other applications in a messaging enterprise. These components simplify the process of making an application JMS-aware. They can be used in Java and non-Java applications

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

alike. By using these objects, consistency and simplicity is ensured in the development of JMS-aware applications without requiring all developers become JMS experts.

Enterprise Messages

Enterprise messages are the definitions of enterprise business objects that will be used in integrations as well as the actions that will be performed on those objects. These are defined during integration analysis and are implemented as constrained XML. They constitute the "contract" with any application or gateway involved in an organization's messaging enterprise. Refer to the OpenEAI Message Definitions Document and the OpenEAI Message Protocol Document for more details regarding the definition of enterprise messages and the protocol.

Java Message Objects

These are Java objects that "wrap" the enterprise message objects defined using XML. This exposes an API (the Message Object API, or "MOA") to developers of messaging applications and gateways. The MOA simplifies the implementation of these applications. With an MOA, application developers can function effectively even without a great deal of knowledge of JMS and XML. Instead, they just need to be familiar with the Java API. This also opens the door for development languages like ColdFusion, PERL, and any other language that can instantiate and call methods on Java objects to use this same API without have to use a specialized set of XML libraries and more rudimentary communications protocols like TCP, HTTP, HTTPS, etc. In essence, Java message objects summarize enterprise messages into a common, re-usable set of objects that can be used consistently in many different application development environments.

Other Foundation Components

The following are peripheral OpenEAI foundation components:

- database connection pools (org.openeai.dbpool package)
- thread pools (org.openeai.threadpool package)
- producer pools (org.openeai.jms.producer package)
- scheduled application foundation (org.openeai.afa package)
- XML utilities (org.openeai.xml)

Additional information regarding these APIs may be included in this document at some point in the future.

Java Message Object Details

All examples listed in this section are for a fictitious system called the AnyERP system. Its domain is any-erp-vendor.com. The artifacts mentioned in the examples can be found in the OpenEAI CVS repository for additional reference information. The official API documentation (javadoc) is also available for download and online browsing.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Important note: all of these objects can be automatically generated using the OpenEAI MoaGenerationApplication reference implementation application. This application also takes care of generating the appropriate `build…Message` methods in the case of an object requiring a different `buildQueryMessage` or `buildGenerateMessage` implementation. Additionally, if the object doesn't support a particular action, it will add the code necessary to throw the appropriate exception. It can do this because all of the information required to make these programming decisions is available in the message definitions and sample messages that an organization prepares during the analysis process. If your organization follows the directory structure recommendations for organizing these artifacts included in the OpenEAI Message Definitions Document, the MoaGenerationApplication can read that structure and generate your entire MOA code for you. ***The detailed descriptions provided in this section are solely provided to clarify what is going on under the covers.*** They can be particularly helpful for developers and architects who are looking to build their own implementation of this functionality. Users of the OpenEAI API can remain largely unaware of these mechanics if they desire.

As mentioned above, message objects can be of two types: XML-aware or both XML- and JMS-aware.

XML Enterprise Objects

Objects that are only XML-aware implement an interface called `XmlEnterpriseObject` and extend a class called `XmlEnterpriseObjectImpl`. `XmlEnterpriseObjectImpl` is the class that provides most of the implementation that makes the Java Message Objects XML-aware. Examples of these objects include: Address, Phone, and Name. These are complex child elements within enterprise message objects like BasicPerson, EmergencyContact, and InstitutionalIdentity. The definition of these enterprise message objects can be found in the appropriate segments file for the organization that defined them. For more information on the segments file and the practice of defining enterprise message objects and maintaining enterprise message definitions, see the OpenEAI Message Definitions Document.

The following hierarchy shows the relationship between these objects:

```
- org.openeai.moa.EnterpriseObject
   - org.openeai.moa.XmlEnterpriseObjectImpl
      - com.any_erp_vendor.moa.objects.resources.v1_0.Address
```

JMS Enterprise Objects

Objects that are both JMS- and XML-aware implement an interface called `JmsEnterpriseObject` and extend an abstract class called `JmsEnterpriseObjectBase`, which extends `XmlEnterpriseObjectImpl` and provides most of the implementation for JMS objects. `JmsEnterpriseObjectBase` adds JMS awareness to an object in addition to the XML awareness it inherits from `XmlEnterpriseObjectImpl`. Examples of these objects are BasicPerson, EmergencyContact, and InstitutionalIdentity. These are objects that have enterprise

messages defined for them and can perform actions such as com.any-erp-vendor.Person.BasicPerson.Query-Request.

The following hierarchy shows the relationship between these objects:

```
- org.openeai.moa.EnterpriseObject
   - org.openeai.moa.XmlEnterpriseObjectImpl
      - org.openeai.moa.jmsobjects.JmsEnterpriseObjectBase
         - com.any_erp_vendor.moa.jmsobjects.person.v1_0.BasicPerson
```

In their simplest form, these objects contain getter and setter methods that correspond to elements and attributes contained in the XML definition of the enterprise object. For example, the `Address` Java object has a private instance variable called `m_type` that corresponds to the XML "type" attribute in the Address element, which is defined in the Segments.dtd file of the AnyERP Vendor. Data is retrieved from this variable via the public `getType()` method and is set via the `setType(String type)` method.

Simple fields (elements that do not have child elements and attributes) are always specified in the Java object as variables of type `String`. If the field is complex (an element with children) the variable specified in the Java object has a type corresponding to that object. For example, the BasicPerson element includes a child element called Name, which is a complex element consisting of simple fields like FirstName, LastName and MiddleInitial. Therefore, the `BasicPerson` Java object has a private instance variable called `m_name` that is an object of type `Name`, which is another Java object that implements `XmlEnterpriseObject` and extends `XmlEnterpriseObjectImpl`. The `BasicPerson` object then contains a `getName()` method which returns a `Name` object associated to the `BasicPerson` object and a `setName(Name aName)` method that accepts a `Name` object in its signature, allowing a developer to set the `Name` object of the `BasicPerson` object. This is true for all "exactly one" or "zero or one" complex fields that are children of other fields.

If the definition of a field specifies that it is "zero or more" or "one or more" (that is, it's a repeating field) then the parent object contains a `java.util.List` of those types of fields. If the repeating field is a simple field then this list will include a list of strings; otherwise, it will include a list of objects of the appropriate type.

For example, the BasicPerson message object in the org.any-erp-vendor.Person category is defined as having "zero or more" Address objects in it. Therefore, the `BasicPerson` Java object includes a private instance variable called `m_address` which is of type `java.util.List`. The contents of this `List` will be a list of `Address` Java objects when populated via the methods of the `BasicPerson` object. Another example is the AdmissionsApplication message object in the com.sct.Student category. This object has a repeating field called "Cohort". However, the definition of the Cohort element in the segments file says that the Cohort element is just a simple field (PCDATA). Therefore, the private `m_cohort` instance variable in the `AdmissionsApplication` Java object is a `List` of `String` objects. Each `String` object represents a Cohort associated with the AdmissionsApplication.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

If the field is a simple field (PCDATA), the getter method returns a `String` and the setter method accepts a `String` as a parameter. For example, the following six methods must be implemented for a repeating, simple field (using Cohort as an example):

```
public int getCohortLength()
public String getCohort(int index)
public java.util.List getCohort()
public void setCohort(int index, String cohort) throws EnterpriseFieldException
public void setCohort(Vector cohort)
public void addCohort(String cohort) throws EnterpriseFieldException
```

If the field is a complex field, the getter method returns an object and the setter method accepts an object as a parameter. For example, the following seven methods must be implemented for a "repeating, complex" field (using Address as an example):

```
public int getAddressLength()
public Address getAddress(int index)
public java.util.List getAddress()
public void setAddress(int index, Address anAddressObject) throws
  EnterpriseFieldException
public void setAddress(Vector addresses)
public void addAddress(Address anAddressObject) throws EnterpriseFieldException
public Address newAddress()
```

The reason for many of these methods is simply convenience. However, they can become very important to an organization trying to integrate non-Java applications. Many non-Java technologies can instantiate and call methods on Java objects (for example, ColdFusion and PERL). Developers of applications using these technologies may not understand Java well enough to know exactly how a typical Java developer might use a `java.util.List`. Additionally, depending on the language, it may be very cumbersome for them to do things that Java developers may take for granted, because of the style imposed upon them by the language they are using.

For all non-repeating fields, only the getter and setter methods need to be implemented. If the field is a simple field (PCDATA), the getter method returns a `String` and the setter method accepts a `String` as a parameter. Using Gender as an example:

```
public String getGender()
public void setGender(String theGenderValue) throws
  EnterpriseFieldException
```

Notice that `the setGender` method throws an `EnterpriseFieldException`. When the setter method is called, the data being passed has to be validated against the Enterprise Object document, which defines the business rules associated with the Gender field such as length, translations, scrubbers, etc.

To accomplish this enterprise object validation, all setter methods for simple fields execute the `getEnterpriseValue` method, which is inherited by all Java message objects and is implemented in the `XmlEnterpriseObjectImpl` class. This method takes the data being passed to the setter method and makes sure it is valid enterprise-quality data. If it is not valid enterprise-quality data and if it cannot be scrubbed, translated, or otherwise converted into valid enterprise data by executing the business rules specified in the enterprise object document for that object, it throws the `EnterpriseFieldException` to indicate that the field value that has been passed to

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

the setter method is invalid enterprise data. The following is an example of how the `setGender` method is actually implemented in a `com.any_erp_vendor.moa.jmsobjects.person.v1_0.BasicPerson` object:

```
m_gender = getEnterpriseValue("Gender", theGenderValue);
```

This call uses the information found in the org/any-erp-vendor/Person/1.0/BasicPersonEO.xml document (BasicPersonEO stands for BasicPerson Enterprise Object) to validate the data for the BasicPerson/Gender element. This is how consistent business rules can be applied to all enterprise objects at the field level.

If the field is a complex field, the getter method returns an object and the setter method accepts an object as a parameter. Additionally, there is a convenience method that returns a newly initialized object of that type. This is so the child objects don't have to be listed in the Deployment Descriptor for the application. Instead, the parent object has the information and ability to create an object of the desired type and initialize it with the appropriate configuration information so developers can use that child object and the rules that have been specified for that object in its EO document. The OpenEAI Deployment Descriptor section of this document discusses how the parent object is made aware of the child object's configuration information. For example:

```
public Name getName()
public void setName(Name aNameObject)
public Name newName()
```

Performing Request Message Actions

When a message action, such as `create`, is called on a `JmsEnterpriseObject`, such as `BasicPerson`, the data contained in the `BasicPerson` object is used to build a com.an-erp-vendor.Person.BasicPerson.Create-Request XML message. That message is sent to a destination (a JMS queue) by the `PointToPointProducer` passed to the `BasicPerson` object's `create` method. A gateway then consumes that message and executes the business logic associated with the create action specified in the com.any-erp-vendor.Person.BasicPerson.Create/ControlAreaRequest/@messageAction attribute of the message. The com.any-erp-vendor.Person.BasicPerson.Create /DataArea/BasicPerson element supplied in the Create-Request message is the data with which to perform the action.

All of the logic needed to build the com.any-erp-vendor.Person.BasicPerson.Create-Request XML message, as well as sending that message to the destination and consuming the reply from the gateway, is implemented in either the `JmsEnterpriseObjectBase` class or in the Java message object itself, if necessary, which extends `JmsEnterpriseObjectBase` and implements `JmsEnterpriseObject`. In this example, it is implemented specifically in the `create` method of the `JmsEnterpriseObjectBase` class. If any errors occur during this process, the create method throws an exception indicating the nature of the problem. Additionally, if the gateway that consumed the create message has problems actually creating a record in its repository with the data sent to it, the `create` method in the `JmsEnterpriseObject` class will inspect the result returned from the gateway, and if

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

errors exist it will throw an exception containing the error that occurred in the gateway. The calling application will catch this exception and either use the information provided in the exception or call the `getLastErrors` method of the `JmsEnterpriseObjectBase` class to get the actual `Error` objects that were created when the error occurred. With these objects, the calling application will have more information at its disposal regarding the error(s) that occurred. The error information will be retrieved from the reply document returned from the gateway and `Error` Java objects will be built from the contents of the reply document. The message actions `update` and `delete` work the same way as the `create` action described here.

There are two other request actions that work a little differently, `query` and `generate`. These actions differ in that they accept an additional parameter in their signature. That parameter is the "key object" with which to query or generate. For example, the com.any-erp-vendor.Person.BasicPerson.Query-Request.dtd specifies that a LightweightPerson element should be included in the DataArea of the query message. Therefore, to perform a BasicPerson query action using the Message Object API, you call the `query` method of a `BasicPerson` object passing a `LightwieghtPerson` object that has been populated with values corresponding to the person you are querying for. The `query` method will return a `List` of objects returned from the gateway that processed the request. These will be Java objects of the type specified in the query; in this case they will be `BasicPerson` objects. These objects will have been built from the contents of the reply document sent back from the gateway and are ready to be used by the developer as Java objects.

The `generate` method works much the same way except the key object being sent is generally some sort of seed or source data that the gateway's generation algorithm uses to generate a composite object. For example, the org.any-openeai-enterprise.CoreApplication.InstitutionalIdentity.Generate-Request.dtd specifies that an UnknownPerson object is required to generate an InstitutionalIdentity. To implement this case, the application developer retrieves an `UnknownPerson` object from the application's `AppConfig` or creates a new object via the `newUnknownPerson()` method in the `InstitutionalIdentity` object, populates it with the appropriate data, and passes it to the `InstitutionalIdentity.generate` method, which builds the org.any-openeai-enterprise.CoreApplication.Generate-Request message and sends it to the destination that will perform the generation and return the generated `InstitutionalIdentity`. Again, if any errors occur in the process of building the outbound message or if errors are sent back from the gateway that performed the generation, it will throw an exception indicating the nature of the problem.

Performing Synchronization Message Actions

For every request action that causes data to be created, generated, updated, or deleted at an authoritative application, there is a corresponding synchronization message that must be published by that authoritative application, which indicates that the action has occurred. Gateways that "gate" authoritative applications or message-aware applications that are authoritative sources can also use the Message Object API to publish these sync messages when these actions occur. These sync methods are also specified in the `JmsEnterpriseObject` interface and implemented in `JmsEnterpriseObjectBase`. They are `createSync`, `updateSync`, `deleteSync` and `generateSync`. They behave

similarly to the request action methods, except they accept a `PubSubProducer` object instead of a `PointToPointProducer` in their signatures, because a sync message does not expect a reply.

Building Messages

When an action is performed on a Java Message Object, the contents of that object and key object (in the case of query and generate actions) are used to build an XML document corresponding to the action being performed. This is a two-part process. The first part involves building the XML message corresponding to the action. The second part of the process actually sends that message to a destination via a `PointToPointProducer` or a `PubSubProducer`.

This process is broken into two parts to provide flexible support for various actions of the protocol for each message object. For example, many objects related to person information may use a LightweightPerson object as a key object for queries. In other words, the DataArea element of their Query-Request message is defined to contain a LightweightPerson element. However, other objects, such as InstitutionalIdentity, require a different key object, such as UnknownPerson, in the DataArea of their query document. Still other message objects are totally unrelated to person data and require other objects in Query-Request messages. One common case (for example, the building of Query-Requests with a LightweightPerson) can be implemented the `JmsEnterpriseObjectBase` class; this common case is implemented there. However, if an individual Java Message Object requires different processing than this common case (and many will), it must override the method and build the appropriate document for the action being performed.

These methods all correspond to the action being performed. For example, one of the first things the `query` method of a Java Message Object does is call the `buildQueryMessage` method to build the Query-Request document for the object. If an object such as `InstitutionalIdentity` requires a different type of key object it must override the `buildQueryMessage` method it inherits from `JmsEnterpriseObjectBase` and build the document appropriately for itself, casting the `XmlEnterpriseObject` object to an `UnknownPerson` (or whatever is appropriate according to the message definition) instead of a `LightweightPerson`, which is what the `buildQueryMessage` method in `JmsEnterpriseObjectBase` does because it is a fairly common case.

For every action, there is a corresponding `build…Message` method. Five actions specified by the OpenEAI Message Protocol can actually be performed on a message object: query, create, update, delete, and generate. So the following `build…Message` methods are defined in `JmsEnterpriseObjectBase`: `buildQueryMessage`, `buildCreateMessage`, `buildUpdateMessage`, `buildDeleteMessage` and `buildGenerateMessage`. Since the DataArea portion of the sync messages look exactly the same as their corresponding request messages, they use the same `build…Message` methods to build the XML document for the corresponding sync actions.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Additionally, if an object doesn't support a certain type of message action, it should throw an exception in the `build…Message` method that it implements to inform users that they have attempted to use the object incorrectly—that it doesn't support that method. For example, relatively few objects will need to support the `generate` action. Therefore, all objects that don't support that action must implement a `buildGenerateMessage` method that simply throws an exception saying the action is not supported by that object. In fact, the `buildGenerateMessage` method is defined as an abstract method in `JmsEnterpriseObjectBase`, meaning that all objects that extend `JmsEnterpriseObjectBase` must implement that method. The reason for this design is that generation of an object is usually very specific to that object. For example, the seed or key object is very specific to the object being generated. Most objects, however, will simply throw an exception stating that they don't support the `buildGenerateMessage` method.

Important note: As mentioned above, all of these objects can be automatically generated using the OpenEAI MoaGenerationApplication reference implementation application. This application also takes care of generating the appropriate `build…Message` methods in the case of an object requiring a different `buildQueryMessage` or `buildGenerateMessage` implementation. Additionally, if the object doesn't support a particular action it will add the code necessary to throw the appropriate exception. It can do this, because all of the information required to make these programming decisions is available in the message definitions and sample messages that an organization prepares. If your organization follows the directory structure recommendations for organizing these artifacts included in the OpenEAI Message Definitions Document, the MoaGenrationApplication can read that structure and generate your entire MOA code for you. ***The detailed descriptions provided in this section are solely provided to clarify what is going on under the covers.*** They can be particularly helpful for developers and architects who are looking to build their own implementation of this functionality. Users of the OpenEAI API can remain largely unaware of these mechanics if they desire.

Developing Messaging Applications

When a message-aware application is developed using the OpenEAI foundation components, everything starts with a specialized object called an `AppConfig` object. This object is an XML-aware object that knows how to configure itself from an XML file stored in a directory server, on a web server, or on the file system. This object works in conjunction with an XML configuration document called the OpenEAI Deployment Descriptor. These documents describe, in technical terms, the integrations that will be performed among applications in an organization by specifying all OpenEAI components that will be used by the messaging application or gateway.

AppConfig (org.openeai.config.AppConfig)

Simply put, the `AppConfig` object reads the deployment descriptor and loads itself with all the objects that will be needed for this application based on what it finds in that file. The types of objects that it may load include: message objects, producers, consumers, logging objects, thread pools, database connection pools and general application properties. So, in essence, `AppConfig` is a container that holds pre-configured and, in some cases, started objects that can be retrieved by an application developer when the objects are needed.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

It should be noted that all applications that use OpenEAI foundation components do this same thing one way or another. If a developer chooses to develop a standalone application with a "main" method, it will configure an `AppConfig`. Likewise, the `org.openeai.afa.GenericAppRunner` and the `org.openeai.jms.consumer.MessageConsumerClient` classes both instantiate and initialize an `AppConfig` in the same manner.

For example, let's assume we've been given a completed analysis template, enterprise object documents, and a generated MOA for exposing a legacy mainframe system that runs in batch and is authoritative for person and employee information. Let's refer to this system affectionately as the BarneySystem, meaning the big purple dinosaur. We prepare our technical stories, which basically outline what we need to develop a scheduled application. Let's call it the BarneyScheduledApplication. The BarneyScheduledApplication will read the mainframe extract files that are created at the end of each batch run of the BarneySystem and build BasicPerson and BasicEmployee objects from the mainframe extract files. The building of these message objects from the extract lines will be handled for us by the OpenEAI layout manager foundation and extract layout implementation. It will use the extract file-to-object mappings and the BarneySystem values-to-enterprise-values translations provided by the analysis team in the enterprise object documents for BasicPerson and BasicEmployee. The scheduled application will then publish messages to inform other applications in the messaging enterprise about new people and employees and changes in person and employee state.

There's just one interesting twist, though. The BarneySystem does not know about ID numbers. Years ago when the BarneySystem was implemented, the organization did not issue ID numbers. The BarneyApplication only knows about social security number. So we see in the analysis that our scheduled application is going to have to perform some Query-Requests and Generate-Requests for InstitutionalIdentity with the organization's ID card system to get ID numbers for known people and generate them for new people. So our application needs to read extract file lines, build message objects from the extract lines, query for or generate ID numbers through messages, set the ID number (the InsitutionalId) on the BasicPerson and BasicEmployee objects that were either returned from the query or generated to complete these objects, and then publish sync messages for the appropriate actions.

So, this technical story helps us determine that the BarneyScheduledApplication is going to need the following configurations in its deployment descriptor:

1. Message Objects
   - BasicPerson
   - BasicEmployee

2. A PointToPointProducer to perform request/reply messaging with the ID card systems to get those ID numbers

3. A PubSubProducer to publish the sync messages for BasicPerson and BasicEmployee

Once these objects are loaded into the `AppConfig` object and initialized with information obtained from the deployment descriptor, the application can retrieve the objects out of the `AppConfig` object whenever they are needed. This gives the developer the ability to

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

configure these objects once, and then the application has them at its disposal for use while it's running. This is important because some of these objects require valuable resources that may be expensive to initialize. It also allows us to specify in one place all the configuration information for an application in its entirety.

Very little information has to be stored in a deployment on the target hosts where the application will run. The only information needed with the application jar files and a start script is information used to locate and read the deployment descriptor. This will include some sort of URI specifying the location of the document and may include security credentials necessary to read the document from a directory server or secured web server. Finally, it contains information that tells the `AppConfig` which messaging component it should configure (an application/gateway name). This also gives us the opportunity to develop highly customizable applications using consistent methods. Consistent methods improve an organization's ability to quickly develop and deploy sophisticated applications and administer them efficiently.

There are three places in which deployment descriptors are typically stored: on a file system with the deployed application, on a web server, or in a directory server. For security and deployment management reasons, many organizations are starting to experiment with storing configuration files such as these deployment descriptors in a directory server. Access to all configurations can then be restricted to application administrators, these application administrators do not necessarily have to have access to the servers on which these applications are deployed to administer them, and configuration files do not have to be synchronized between servers when many instances of the same application are run on multiple servers in a clustered deployment. The OpenEAI deployment patterns recommended that deployment descriptors be stored in a directory server. However, that is an organizational choice. Even if your organization does use a directory server for this purpose, during development phases you may find it convenient to use the local file system or a web server at times.

To switch between these storage locations, simply change the `providerUrl` property shown below to be the desired URI. The `AppConfig` object takes care of the rest.

All applications, gateways, and servlets can be started with a simple properties file that contains the following properties (obviously, the values assigned to the properties will differ depending on the application, gateway, or servlet). The following are examples of properties files that point the `AppConfig` object of a starting application to its deployment descriptor. Note that in some cases, lines have been broken to fit them on this page in this document.

A properties file pointing to the deployment descriptor in a directory server:

```
providerUrl=ldaps://ldap.any-openeai-enterprise.org:636/ou=Development,
  ou=Configurations,ou=Messaging,dc=openerp,dc=org
initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
securityPrincipal=uid=SampleApplication,ou=Development,ou=Users,
  ou=Messaging,dc=openerp,dc=org
securityCredentials=secretpassword
configDocName=configxmlname=SampleApplication
messageComponentName=SampleApplication
```

A properties file pointing to the deployment descriptor on a web server:

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
providerUrl= http://xml.openeai.org/xml/message/uillinois
  /xml/1.0/Configuration/SampleApplication.xml
messageComponentName=SampleApplication
```

A properties file pointing to the deployment descriptor on a local file system:

```
providerUrl= file://configs/SampleApplication.xml
messageComponentName=SampleApplication
```

The following is an example of how a standalone application can load the properties object:

```
String propFileName = args[0];
Properties props = new Properties();
InputStream in = new FileInputStream(propFileName);
props.load(in);
in.close();

aConfig = new AppConfig(props);
```

After the properties file is loaded it is passed to the `AppConfig` constructor, which will bind to the directory server (in this case), search for and read the deployment descriptor, and initialize itself with contents found in that file. This is the same processing that occurs when the document is stored on a web server or file system, except with a directory server, stricter access controls can be put in place regarding who has access to the deployment descriptor. So, as it turns out, there is a very small set of external properties that must be stored with the application. Once the application has this information, it can obtain the contents of the configuration document and initialize itself.

Scheduled applications and message gateways all use a common component to start themselves, called `GenericAppRunner` or `MessageConsumerClient` respectively, that use this same approach. Therefore, it is unlikely that very many developers even need to code this `AppConfig` initialization. The only time it would be necessary is if they are developing a standalone application that is not a scheduled application or a message gateway. Decisions to do this should be weighed carefully, because foundation components are already in place to do this initialization and ensure that these applications and gateways can be started and stopped in a uniform fashion.

Once `AppConfig` has been loaded with pre-configured, instantiated, and initialized objects, developers can use a couple different methods of the `AppConfig` object to retrieve those objects from `AppConfig` and use them in the normal course of development. These methods are:

```
getObject(String name)
getObjectByType(String fullyQualifiedClassName)
```

For example, the following example demonstrates getting a `PubSubProducer` from `AppConfig` named “SyncProducer,” which it instantiated and initialized based on information found in the application section of the deployment descriptor.

```
PubSubProducer pubSub1 = (PubSubProducer)aConfig  .getObject("Sync
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

The next example asks the `AppConfig` object to locate and return the first object it finds that has a class name equal to the class name of the PubSubProducer which it instantiated and initialized based on information found in the deployment descriptor. Note that this will return the first object of this type. If multiple PubSubProducers are listed in the deployment descriptor, and the developer wishes to retrieve a specific one, s/he should use the `getObject` method instead.

```
PubSubProducer pubSub2 = null;
PubSub2 = (PubSubProducer)aConfig
  .getObjectByType(pubSub2.getClass().getName());
```

Scheduled Applications

A scheduled application is an application that executes certain business logic at a configurable interval. That interval can be immediate or it can be based on a flexible, built-in scheduling facility that allows developers to specify certain business logic be executed at a given interval or on specified days at specified times.

Scheduled Applications can be of four types:

1. Application. The application starts, the business logic executes immediately, and the application ends. This type of scheduled application is referred to as simply an "application". That is, it executes the specified business logic once then it exits. Therefore, it's a "scheduled" application that just runs once and then ends.

2. Triggered application. The application starts, the business logic executes immediately and the application waits to be manually stopped. This type of scheduled application is referred to as a "triggered application". That is, it executes the specified business logic once then it waits for manual intervention before it ends.

3. Daemon with immediate execution. The application starts, it sleeps for a given period of time, it wakes up and executes the business logic, then goes back to sleep. This process is repeated indefinitely. This type of scheduled application is referred to as a "daemon" scheduled application with "immediate" execution.

4. Daemon with scheduled execution. The application starts, it sleeps for a given period of time, it wakes up, and if it is supposed to run based on its configuration, it executes the business logic, then goes back to sleep. This process is also repeated indefinitely. This type of scheduled application is referred to as a "daemon" scheduled application with "scheduled" execution.

All scheduled applications are instances of the `org.openeai.afa.GenericAppRunner` class. This is the only runnable class that needs to exist for these types of applications. Scheduled applications are an implementation of the command pattern. The business logic executed according to the application's schedule is implemented in commands (Java classes) that perform the desired business logic.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Really, the only difference between a scheduled application and a gateway is what triggers the execution of the business logic. Where a gateway executes commands when it one of its consumers consumes a message, a scheduled application executes commands when a schedule is met.

Refer to the OpenEAI API javadoc in the org.openeai.afa package for more details information on scheduled application foundation.

Message Gateways

All message gateways are an instance of the `org.openeai.jms.consumer.MessageConsumerClient` class. This is the only runnable class that exists for OpenEAI based message gateways. `MessageConsumerClient` instantiates an `AppConfig` object with the appropriate deployment descriptor, and the consumers associated with the gateway are started. Message gateways developed with the OpenEAI foundation may use `PointToPointConsumers` to handle and reply to incoming request messages and `PubSubConsumers` to consume and process incoming sync messages. The configurations for these objects are included in the deployment descriptor for the message gateway. When a message gateway is started, it builds its `AppConfig` object, which instantiates and configures the appropriate consumer(s) for the message gateway. Once the consumers are started, the application is ready to consume messages.

Consumers behave differently when they consume different messages. This behavior is implemented with the command pattern, a common object-oriented design pattern. The JMS message that is consumed by the consumer has in it information that tells the consumer what business function to perform, in other words, which command to execute. Commands are Java classes that implement at least one method, the `execute` method. When the consumer receives a message, it looks at the message and calls the `execute` method on the appropriate command associated with that message. All of these commands are loaded and initialized when the consumer is started up. If a message doesn't have a command associated with it, a default command is executed, if one has been specified. These commands are all specified in the consumer configuration in the deployment descriptor.

`ConsumerCommand` objects themselves contain an `AppConfig` object that they use to pre-configure objects needed by the command when it executes. This is specified in the command configuration portion of the deployment descriptor. The configuration of a command is identical to the configuration of an application. Basically, commands can be thought of as "mini-applications" that get executed when a consumer consumes a message. Note that this same principal is used for scheduled applications. The only difference is that gateways execute commands as a result of consuming a message and scheduled applications execute commands as a result of a schedule being met. The instantiation and initialization of the command's `AppConfig` object is automatically performed when the consumer is configured and the commands the consumer might execute are instantiated.

The `AppConfig` object for a message gateway will always include at least one consumer, and the consumer(s) will include at least one command to execute when the consumer receives a message. Additionally, the command configuration section of the

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

deployment descriptor may include message objects, producers, thread pools, database connection pools, and general commad properties, just like an application's configuration.

From an OpenEAI foundation component perspective, a message gateway can be summed up as follows: ***A Gateway contains one or more Consumers that execute one or more Commands.*** These commands are what developers write to make a gateway perform the business logic specified in the analysis templates.

For more information regarding the OpenEAI JMS consumer foundation, please refer to the OpenEAI API javadoc in the org.openeai.jms.consumer and org.openeai.jms.consumer.commands packages.

OpenEAI Deployment Descriptor

The OpenEAI deployment descriptor is an XML document used to configure applications developed using the OpenEAI foundation components. This section includes the document type definition, which constrains the descriptor. The definition includes detailed descriptions of each section of the definition. For additional information regarding the OpenEAI configuration foundation, please refer to the OpenEAI API javadoc in the org.openeai.config package.

Begin Deployment.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!--    $Revision: 1.14 $
        $Date: 2003/01/29 00:33:07 $
        $Source: /cvs/repositories/openeai/project/xml/dtd/1.0/Deployment.dtd,v $
-->
<!-- This file is part of the OpenEAI Application Foundation or
        OpenEAI Message Object API created by Tod Jackson
        (tod@openeai.org) and Steve Wheat (steve@openeai.org) at
        the University of Illinois Urbana-Champaign.

        Copyright (C) 2003 The OpenEAI Software Foundation

        This library is free software; you can redistribute it and/or
        modify it under the terms of the GNU Lesser General Public
        License as published by the Free Software Foundation; either
        version 2.1 of the License, or (at your option) any later version.

        This library is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
        Lesser General Public License for more details.

        You should have received a copy of the GNU Lesser General Public
        License along with this library; if not, write to the Free Software
        Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

        For specific licensing details and examples of how this software
        can be used to build commercial integration software or to implement
        integrations for your enterprise, visit http://www.OpenEai.org/licensing.
 -->
<!--Release History
        1/26/2003       tod@openeai.org         Original release.
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

steve@openeai.org
-->
<!--
A Deployment represents one or more applications, message gateways, or servlets. The document that conforms to this definition most frequently serves as the deployment descriptor for one application, message gateway, or servlet. However, the deployment descriptor is constrained to describe multiple applications, message gateways, or servlets, so many configurations could be rolled into one deployment descriptor, if that is beneficial to an organization's deployment practices. In fact, all configurations for a messaging enterprise could be rolled into a single deployment descriptor for documentation purposes only. For example, in the future the OpenEAI Methodology department may provide an application to take all deployment documents, roll them into large Deployment document that is representative of your entire messaging enterprise, and then generate messaging enterprise schematics and skeleton enterprise messaging documentation for your enterprise from the these combined deployment descriptors.

The name of the deployment should be descriptive of the applications in the deployment. The type and status should indicate whether this deployment is for a production, demo, test, or development environment. The status should indicate whether the deployment is planned or whether it has already been deployed. Planned indicates that the configuration has never been used to run an application. Deployed means that an application or applications have been run with the configuration contained in the descriptor.-->
<!ELEMENT Deployment (Application*, MessageGateway*, Servlet*)>
<!ATTLIST Deployment
name CDATA #REQUIRED
type (production | demo | test | development) #REQUIRED
status (planned | deployed) #REQUIRED
baseURI CDATA #REQUIRED
>
<!ELEMENT MessagingComponents (Applications?, MessageGateways?, Servlets?)>
<!ELEMENT Applications (Application*)>
<!--
An application consists of a full name, an optional short name, a description, and a configuration. It also has an "id" attribute that should uniquely identify the application in the messaging enterprise. This id attribute contains the value that corresponds to the "messageComponentName" property in the application's properties file. These elements identify and document that application. The names should be clear, and the description should be comprehensive enough to provide some background about the purpose and function of the application. This information may be used to generate enterprise schematics in the future, so it is recommended that this information be filled out accurately with meaningful data.-->
<!ELEMENT Application (FullName, ShortName?, Description, Configuration)>
<!ATTLIST Application
id CDATA #REQUIRED
>
<!--
A Configuration element is comprised of container elements for different types of specific and tightly constrained configuration elements. For example, the LoggerConfigs element will contain a LoggerConfig element. ScheduledAppConfigs will contain a ScheduledAppConfig element. ProducerConfigs will contain one or more ProducerConfig elements.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

This layer of container elements helps organize the deployment descriptor, particularly in the case of large and complex applications. This layer of container elements ensures that developers and administrators will always place and find configurations for objects of a certain type in the same section of the descriptor.

Additionally, most container-level elements allow the developers to specify default values for some relevant child elements of the configuration elements they contain. For example, at the ProducerConfigs level you may specify ConnectionFactoryName, InitialContextFactory, ProviderURL, SecurityPrincipal, SecurityCredentials, ProducerIdURL, and ConfigClass element values and numberOfProducers, tempQueuePoolSize, startOnInitialization, acknowledgementMode, transacted, and deliveryMode attribute values. These elements and attributes are also children of each ProducerConfig element that the ProducerConfigs element may contain. If the deployer specifies values for these elements at the container (ProducerConfigs) level, then these values will be applied to all ProducerConfig child elements of that container by the OpenEAI configuration foundation at runtime, unless the individual ProducerConfig elements contained in ProducerConfigs override these default values by specifying different values. This ability to specify default values at the container level helps eliminate redundancy in the deployment descriptor. It is common for different configuration elements within a container to be very similar, differing only in one or two element or attribute values.

Each of the configuration elements are wrapped as Java objects in the OpenEAI configuration foundation, and these OpenEAI Java configuration objects are used to configure the rest of the OpenEAI foundation components throughout the OpenEAI API. The XML configuration elements are wrapped as Java configuration objects, so the foundation components that we want to configure, initialize, and use do not have to be XML aware. The constructors of foundation components expect an appropriate Java configuration object from the OpenEAI configuration foundation. In this way, we isolate the more complex foundation components from the mechanics of reading and parsing configuration parameters. This allows us to more quickly and efficiently respond to change and enhance the configuration of foundation components. If we want to configure foundation components from simple properties files, from some completely different text file format, from a database structure, or from a directory server schema, the reading and parsing of these configuration parameters must be implemented in the configuration objects or in their ancestors and not in the functional foundation components (such as producers, consumers, and connection pools) themselves.

As you will see these Java configuration objects are named for the object being configured which allows us to use different Java configuration objects in the future if desired.

So when an application starts, it instantiates an AppConfig object, which reads the deployment descriptor that conforms to this definition. AppConfig then instantiates Java configuration objects that wrap the configuration elements it has found in the deployment descriptor and then uses these Java configuration objects to instantiate the foundation components that these Java configuration objects are intended to configure. Finally, depending upon the properties of the configurations themselves, AppConfig will initialize and may start or otherwise prepare foundation components for use.-->

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
<!ELEMENT Configuration (LoggerConfigs?, ScheduledAppConfigs?, ProducerConfigs?,
ConsumerConfigs?, MessageObjectConfigs?, ThreadPoolConfigs?, DbConnectionPoolConfigs?,
PropertyConfigs?)>
<!ATTLIST Configuration
    initializeUsingThreads (true | false) #IMPLIED
>
<!--
        The ScheduledAppConfigs element contains one ScheduledAppConfig
        element.  Like all container elements ScheduledAppConfigs allows
        the deployer to specify default values at the container level.  However,
        in this case, these default values are only applied to the one
        ScheduledAppConfig that may appear in this container.  At this time it is
        only appropriate for there to be one ScheduledAppConfig element in this
        container.-->
<!ELEMENT ScheduledAppConfigs (ConfigClass, ObjectClass, ThreadPoolConfig, ScheduledAppConfig)>
<!ATTLIST ScheduledAppConfigs
    type (daemon | application | triggered) #REQUIRED
    scheduleCheckInterval CDATA #REQUIRED
>
<!--
        The ScheduledAppConfig element contains all of the information needed to
        configure a scheduled application using OpenEAI foundation components.
        The following is a description of each of the elements and attributes of
        ScheduledAppConfig:

        name

        This is the unique name by which this Scheduled Application is known to
        the AppConfig object.

        type

        The type of scheduled application being started.  The choices are "daemon",
        "application" and "triggered".  Type 'daemon' means the application will
        start and continue running indefinitely, checking the schedules that it
        supports on a configurable interval.  Type 'application' means the
        application will start, execute the commands associated with all schedules
        that it supports and then exit.  Applications of type 'triggered' start,
        execute the commands associated with all schedules that it supports and
        then wait for a signal (like a kill) to exit.

        scheduleCheckInterval

        The interval in milliseconds at which the scheduled application should check
        whether it is time to execute any of its schedules.

        ConfigClass

        The ConfigClass is the fully-qualified class name of the enterprise
        configuration object that will wrap all this configuration information
        and which will be passed to the constructor of the ScheduledApp class
        to initialize the ScheduledApp.  For scheduled applications, this class
        name will always be org.openeai.config.ScheduledAppConfig.

        ObjectClass

        The ObjectClass element should contain the fully-qualified class name of
        the object to instantiate and start. For scheduled applications this should
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

always be org.openeai.afa.ScheduledApp.

ThreadPoolConfig

The ThreadPoolConfig element configures the main thread pool used by the scheduled application to add transactions to for command execution when a schedule is met. This is where the commands are executed. Since the scheduled application uses a thread pool, it doesn't have to wait for the command to finish execution before checking the next schedule. Instead, it just checks the schedule listed, and if the schedule is supposed to run, it executes the command(s) associated with the schedule by adding them as jobs in the thread pool. Thread pools used as main thread pools in a scheduled application should be configured to check their status prior to adding a command to execute to the pool by setting the checkBeforeProcessing attribute to 'true'. This ensures that if the thread pool is totally busy (that is, if the maximum number of threads allocated to this pool are already busy processing transaction) the scheduled application will stop checking schedules and wait until some threads in the pool are idle and the pool can accept more transactions. This setting reduces the risk of business logic not being executed if the scheduled application should terminate abnormally. A detailed description of the elements and attributes of the ThreadPoolConfig can be found with the ThreadPoolConfig element in this document definition.

The following is an example of a ScheduledAppConfig element from a sample deployment descriptor:

```
<ScheduledAppConfigs scheduleCheckInterval="50000" type="daemon">
    <ConfigClass>org.openeai.config.ScheduledAppConfig</ConfigClass>
    <ObjectClass>org.openeai.afa.ScheduledApp</ObjectClass>
    <ThreadPoolConfig name="Standard" maxThreads="150" minThreads="0" maxIdleTime="1">
        <ConfigClass>org.openeai.config.ThreadPoolConfig</ConfigClass>
        <ObjectClass>org.openeai.threadpool.ThreadPoolImpl</ObjectClass>
    </ThreadPoolConfig>
    <ScheduledAppConfig name="SampleScheduledApp">
        <Schedules isImmediate="false" runMechanism="Thread">
            <ObjectClass>org.openeai.afa.Schedule</ObjectClass>
                <Schedule name="CheckForPendingTransactions">
                    <RunTime>
                        <Day name="Daily"/>
                        <Times>
                            <Time>
                                <Hour>06</Hour>
                                <Minute>05</Minute>
                            </Time>
                            <Time>
                                <Hour>08</Hour>
                                <Minute>05</Minute>
                            </Time>
                        </Times>
                    </RunTime>

            *** see ConsumerConfig for descriptions of Commands because these are the same ***
                    <Commands inboundXmlValidation="false" outboundXmlValidation="false"
writeToFile="false">
                        <Command type="scheduled" isDefault="false">
                        </Command>
                    </Commands>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                        </Schedule>
                    </Schedules>
                </ScheduledAppConfig>
            </ScheduledAppConfigs>

            Simply put, this configuration says...wake up every 5 minutes and check to see
            if the "CheckForPendingTransactions" schedule should run.  If so, execute the
            command associated with that schedule in a thread. Therefore, in this case,
            whichever command(s) were specified in the Commands element would be
            executed by the ScheduledApp foundation every day at 6:05 AM and 8:05 AM.-->
<!ELEMENT ScheduledAppConfig (ConfigClass?, ObjectClass?, ThreadPoolConfig?, Schedules)>
<!ATTLIST ScheduledAppConfig
      name CDATA #REQUIRED
      type (daemon | application | triggered) #IMPLIED
      scheduleCheckInterval CDATA #IMPLIED
>
<!ELEMENT Schedules (ObjectClass, RunTime*, Schedule+)>
<!ATTLIST Schedules
      isImmediate (false | true) #REQUIRED
      runMechanism (Thread | SubProcess) #REQUIRED
>
<!ELEMENT Schedule (ObjectClass?, RunTime*, Commands)>
<!ATTLIST Schedule
      name CDATA #REQUIRED
      isImmediate (false | true) #IMPLIED
      runMechanism (Thread | SubProcess) #IMPLIED
>
<!ELEMENT RunTime (Day, Times)>
<!ELEMENT Day EMPTY>
<!ATTLIST Day
      name (Daily | Weekdays | Weekends | Sunday | Monday | Tuesday | Wednesday | Thursday | Friday |
Saturday) #REQUIRED
>
<!ELEMENT Times (Time+)>
<!ELEMENT Time (Hour, Minute)>
<!ELEMENT Hour (#PCDATA)>
<!ELEMENT Minute (#PCDATA)>
<!--
            The ProducerConfigs element contains one or more ProducerConfig
            elements. Like all container elements ProducerConfigs allows the
            deployer to specify default values at the container level that are
            applied to the elements and attributes of each ProducerConfig contained
            within it unless these default values are overridden by specifying different
            values within an individual ProducerConfig.-->
<!ELEMENT ProducerConfigs (ConnectionFactoryName, InitialContextFactory, ProviderURL,
SecurityPrincipal, SecurityCredentials, ProducerIdURL, ConfigClass, ProducerConfig+)>
<!ATTLIST ProducerConfigs
      numberOfProducers CDATA #IMPLIED
      tempQueuePoolSize CDATA #IMPLIED
      startOnInitialization (true | false) #REQUIRED
      acknowledgementMode (AUTO_ACKNOWLEDGE | CLIENT_ACKNOWLEDGE |
DUPS_OK_ACKNOWLEDGE) #IMPLIED
      transacted (true | false) #IMPLIED
      deliveryMode (NON-PERSISTENT | PERSISTENT) #IMPLIED
>
<!--
            The ProducerConfig element contains all of the information needed to
            configure and start (if specified) OpenEAI JMS producer foundation objects
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

(PointToPointProducer or PubSubProducer). The following is a description of each of the elements and attributes of ProducerConfig:

LoggingProducer

This is a PubSubProducer that is associated with a PubSub producer, which can be used to log every sync message that is published by a PubSubProducer producer to a JMS topic. These logged sync messages can then be consumed and processed by a logging service. See the OpenEAI Implementation Strategies Document for details on enterprise synchronization message logging strategies. Since the LoggingProducer is just another producer, it has the same definition as ProducerConfig itself. In order to avoid a less clear, recursive definition, this LoggingProducer element is named differently than its parent element ProducerConfig. Presently, only PubSubProducers will use a LoggingProducer. When the PubSubProducer is asked to publish a message via the publishMessage method, in addition to publishing a sync message to the primary destination associated with the PubSunProducer, it will check to see if the LoggingProducer associated with the PubSubProducer is started. If it is, it will use the LoggingProducer to publish a copy of the sync message to the destination specified in the LoggingProducer configuration. If the LoggingProducer is not started, it will not be used. Therefore, to turn the sync message logging production off for a PubSubProducer, you set its LoggingProducer's 'startOnInitialization' attribute to 'false'.

ConnectionFactoryName

This specifies the lookup name for the JMS ConnectionFactory that will be used by this producer. This can be either a TopicConnectionFactory or a QueueConnectionFactory, depending on the type of producer you are configuring. A JMS connection factory contains JMS provider-specific information about how to connect to the JMS provider. Connection factories are 'administered objects' and are stored in a directory server or other JNDI store whose location is specified in the ProviderURL element.

DestinationName

This is the name of the JMS queue or topic to which this producer will produce or publish messages. This is also an 'administered object' stored in the directory server or other JNDI store whose location is specified in the ProviderURL element.

InitialContextFactory

This is the fully-qualified class name of the InitialContectFactory the producer will use to obtain an initial context with the directory server or other JNDI store where the administered objects reside. When an organization uses an LDAP repository for JMS administered objects, this will almost always be com.sun.jndi.ldap.LdapCtxFactory. This value will vary based on the repository being used to store JMS administered objects.

ProviderURL

This is the location in the directory server or other JNDI store where the connection factories and destinations (administered objects) reside.

SecurityPrincipal

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

This element specifies the distinguished name of the directory user allowed to access the administered objects in the directory server or other JNDI store at the location specified by ProviderURL.

SecurityCredentials

This element specifies the password associated with the SecurityPrincipal.

Username

This is the JMS Provider username allowed to publish or produce messages to the desired destination. If you are following JMS and OpenEAI preferred deployment patterns, you will leave this element blank, because this username should be specified in the ConnectionFactory maintained in the JNDI administered object store.

Password

This is the password corresponding to the JMS Provider username allowed to publish or produce messages to the desired destination. If you are following JMS and OpenEAI preferred deployment patterns, you will leave this element blank, because the username and password should be specified in the ConnectionFactory maintained in the JNDI administered object store.

ProducerIdUrl

This is the URI to the UuidGen servlet or other web service used by your organization to generate producer ID numbers. If your organization's central producer ID number generations service cannot be reached at runtime, the OpenEAI producer foundation with generate an ID number locally using a UUID generation algorithm. For more information on the significance and requirements for producer ID services, see the comments for the definition of the MessageId element in the OpenEAI segments file (http://xml.openeai.org/message/releases/org/openeai/Resources/1.0/Segments.dtd).

ConfigClass

The ConfigClass is the fully-qualified class name of the enterprise configuration object that will wrap all this configuration information and which will be passed to the constructor of the Producer class to initialize the Producer. For producers, this class name will always be org.openeai.config.ProducerConfig.

ObjectClass

The ObjectClass element should contain the fully-qualified class name of the object to instantiate and start. This should only be one of the following classes: org.openeai.jms.producer.PointToPointProducer for point-to-point message production or org.openeai.jms.producer.PubSubProducer for publish/subscribe message production.

DefaultCommandName

This is the default command name that should be associated with each message produced by this producer in its JMS properties, if that message does not already have a COMMAND_NAME property with a value. This is the mechanism by which the OpenEAI message consumer foundation determines, which business

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

logic command to execute when a message is consumed.

name

The name attribute contains the unique name by which this producer is known to the AppConfig object. When the developer of application wants to retrieve the producer from the AppConfig object, it may do so by specifying this name in the getObject(String name) method of the AppConfig object.

numberOfProducers

This attribute specifies how many producers with this configuration should be initialized and started. Specifying a number greater than one effectively creates a pool of identically configured producers. This technique of producer pooling can improve the performance of message production in high-volume messaging applications. There will be this number of producers in AppConfig that all start with the same name but have a sequence number appended to their name. Additionally, AppConfig will contain a ProducerPool object with this producer's base name that can be retrieved by the application developer just like any other object. For example, if a producer configuration contained a name of 'P2PProducer' and a numberOfProducers of '3', then a developer could get a reference to a producer pool and get a producer from the pool as follows:

```
/* retrieve the ProducerPool from AppConfig */
ProducerPool p2pProducerPool = (ProducerPool)m_appConfig.getObject("P2PProducer"));

/* retrieve a PointToPointProducer from the producer pool */
PointToPointProducer p2pn = PointToPointProducer)p2pProducerPool.getProducer();
```

The preferred programming practice is for developers to code to use producer pools even if they are only using one producer for each purpose initially. Doing so gives deployers an additional lever to push when tuning a messaging application. If during testing or in production bottlenecks are detected and attributed to a heavy need for concurrent message production, the developer can simply increase the number of producers in each producer pool, which will increase the application's capacity for concurrent message production.

tempQueuePoolSize

This number specifies how many TemporaryQueue and QueueReceiver objects should be created upon initialization. This is another performance enhancement that allows multithreaded applications to produce messages using pre-established TemporaryQueues instead of creating them at message production time.

startOnInitialization

This attribute tells the producer whether or not to start itself when it is initialized. If this is 'false', it will be the responsibility of the application to start the producer. In most cases, this should always be set to 'true' so the expense of starting the producer is paid only once at the time the application is initialized.

acknowledgementMode

This attribute specifies the JMS acknowledgement mode to use at the JMS session level. Because of the lack of value this JMS property provides at

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

this time, it is defaulted to AUTO_ACKNOWLEDGE for all producers.

transacted

This specifies whether the JMS session sending these messages is to be transacted or not. This JMS property also has limited value at this point, because when you commit a transaction at the consuming end, you're committing everything that has been sent on that session and not on a message-by-message basis.

deliveryMode

This is defaulted to PERSISTENT for PointToPointProducers and is not used for PubSubProducers, because all OpenEAI PubSubConsumers subscribe durably.

The following is an example specifying configuration information for a set of producers (one PubSubProducer and one PointToPointProducer) from a sample deployment descriptor:

```
<ProducerConfigs startOnInitialization="true">
<ConnectionFactoryName>cn=TCF.PaymasterBatchApplication</ConnectionFactoryName>
 <InitialContextFactory>com.sun.jndi.ldap.LdapCtxFactory</InitialContextFactory>
<ProviderURL>ldaps://ldap.any-openeai-
enterprise.org:636/ou=PubSub,ou=Development,ou=AdministeredObjects,ou=Version3.5,ou=SonicMQ,ou=
Providers,ou=Messaging,dc=any-openeai-enterprise,dc=org</ProviderURL>
<SecurityPrincipal>uid=PaymasterBatchApplication,ou=Development,ou=Users,ou=Messaging,dc=any-
openeai-enterprise,dc=org</SecurityPrincipal>
        <SecurityCredentials>secretpassword</SecurityCredentials>
        <ProducerIdURL>http://www.any-openeai-
enterprise.org/uuidgen/servlet/UuidGen</ProducerIdURL>
        <ConfigClass>org.openeai.config.ProducerConfig</ConfigClass>
        <ProducerConfig name="BasicEmployeeSyncProducer">
            <LoggingProducer name="PaymasterSyncLoggingProducer" startOnInitialization="false">
                <ConnectionFactoryName>cn=TCF.EnterpriseSyncLogger</ConnectionFactoryName>
                <DestinationName>cn=org.any-
openeai.enterprise.topic.EnterpriseSyncLogger</DestinationName>
                <InitialContextFactory>com.sun.jndi.ldap.LdapCtxFactory</InitialContextFactory>
<ProviderURL>ldaps://ldap.any-openeai-
enterprise.org:636/ou=PubSub,ou=Development,ou=AdministeredObjects,ou=Version3.5,ou=SonicMQ,ou=
Providers,ou=Messaging,dc=any-openeai-enterprise,dc=org</ProviderURL>
<SecurityPrincipal>uid=PaymasterBatchApplication,ou=Development,ou=Users,ou=Messaging,dc=any-
openeai-enterprise,dc=org</SecurityPrincipal>
                <SecurityCredentials>secretpassword</SecurityCredentials>
                <ProducerIdURL>http://www.any-openeai-
enterprise.org/uuidgen/servlet/UuidGen</ProducerIdURL>
                <ConfigClass>org.openeai.config.ProducerConfig</ConfigClass>
                <ObjectClass>org.openeai.jms.producer.PubSubProducer</ObjectClass>
                <DefaultCommandName>EnterpriseSyncLogger</DefaultCommandName>
            </LoggingProducer>
            <ConnectionFactoryName>cn=TCF.PaymasterBatchApplication</ConnectionFactoryName>
            <DestinationName>cn=org.any-openeai-
enterprise.topic.EnterpriseTransRouter</DestinationName>
            <ObjectClass>org.openeai.jms.producer.PubSubProducer</ObjectClass>
        </ProducerConfig>
        <ProducerConfig name="P2PPaymasterGateway" tempQueuePoolSize="12">
            <ConnectionFactoryName>cn=QCF.PaymasterBatchApplication.producer
</ConnectionFactoryName>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                    <DestinationName>cn=org.any-openeai-
enterprise.queue.PaymasterGateway</DestinationName>
          <ProviderURL>ldaps://ldap.any-openeai-
enterprise.org:636/ou=PointToPoint,ou=Development,ou=AdministeredObjects,ou=Version3.5,ou=SonicMQ
,ou=Providers,ou=Messaging,dc=any-openeai-enterprise,dc=org</ProviderURL>
                    <ObjectClass>org.openeai.jms.producer.PointToPointProducer</ObjectClass>
                    <ThreadPoolConfig name="P2P1 ThreadPool" maxThreads="10" minThreads="0"
maxIdleTime="1"/>
               </ProducerConfig>
               <ProducerConfig name="P2PICardGateway" tempQueuePoolSize="3"
numberOfProducers="1">
                    <ConnectionFactoryName>cn=QCF.PaymasterBatchApplication.producer
</ConnectionFactoryName>
                    <DestinationName>cn=org.any-openeai-
enterprise.icard.queue.IcardGateway</DestinationName>
          <ProviderURL>ldaps://ldap.any-openeai-
enterprise.org:636/ou=PointToPoint,ou=Development,ou=AdministeredObjects,ou=Version3.5,ou=SonicMQ
,ou=Providers,ou=Messaging,dc=any-openeai-enterprise,dc=org</ProviderURL>
                    <ObjectClass>org.openeai.jms.producer.PointToPointProducer</ObjectClass>
                    <ThreadPoolConfig name="P2P2 ThreadPool" maxThreads="10" minThreads="0"
maxIdleTime="1"/>
               </ProducerConfig>
          </ProducerConfigs>


          The following example demonstrates the retrieval of a Producer from the
          AppConfig object:

          PointToPointProducer p2p1 =
(PointToPointProducer)aConfig.getObject("PaymasterP2PProducer");
          PointToPointProducer p2p1 =
(PointToPointProducer)aConfig.getObjectByType("org.openeai.jms.producer.PointToPointProducer");

          Note that the second example would only make sense if there were only one
          PointToPointProducer stored in the AppConfig object.-->
<!ELEMENT ProducerConfig (LoggingProducer?, ConnectionFactoryName?, DestinationName,
InitialContextFactory?, ProviderURL?, SecurityPrincipal?, SecurityCredentials?, Username?, Password?,
ProducerIdURL?, ConfigClass?, ObjectClass, DefaultCommandName?)>
<!ELEMENT ObjectClass (#PCDATA)>
<!ATTLIST ProducerConfig
     name CDATA #REQUIRED
     numberOfProducers CDATA #IMPLIED
     tempQueuePoolSize CDATA #IMPLIED
     startOnInitialization (true | false) #IMPLIED
     acknowledgementMode (AUTO_ACKNOWLEDGE | CLIENT_ACKNOWLEDGE |
DUPS_OK_ACKNOWLEDGE) #IMPLIED
     transacted (true | false) #IMPLIED
     deliveryMode (NON-PERSISTENT | PERSISTENT) #IMPLIED
>
<!ELEMENT LoggingProducer (ConnectionFactoryName, DestinationName, InitialContextFactory,
ProviderURL, SecurityPrincipal, SecurityCredentials, Username?, Password?, ProducerIdURL, ConfigClass,
ObjectClass, DefaultCommandName?)>
<!ATTLIST LoggingProducer
     name CDATA #REQUIRED
     startOnInitialization (true | false) #REQUIRED
>
<!ELEMENT ConnectionFactoryName (#PCDATA)>
<!ELEMENT DestinationName (#PCDATA)>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
<!ELEMENT DefaultCommandName (#PCDATA)>
<!ELEMENT InitialContextFactory (#PCDATA)>
<!ELEMENT ProviderURL (#PCDATA)>
<!ELEMENT SecurityPrincipal (#PCDATA)>
<!ELEMENT SecurityCredentials (#PCDATA)>
<!ELEMENT Username (#PCDATA)>
<!ELEMENT Password (#PCDATA)>
<!ELEMENT ProducerIdURL (#PCDATA)>
<!--
        The ConsumerConfigs element contains one or more ConsumerConfig
        elements. Like all container elements ConsumerConfigs allows the
        deployer to specify default values at the container level that are
        applied to the elements and attributes of each ConsumerConfig contained
        within it unless these default values are overridden by specifying different
        values within an individual ConsumerConfig.-->
<!ELEMENT ConsumerConfigs (ConnectionFactoryName, InitialContextFactory, ProviderURL,
SecurityPrincipal, SecurityCredentials, ConfigClass, GenericResponse, ThreadPoolConfig,
ConsumerConfig+)>
<!ATTLIST ConsumerConfigs
    startOnInitialization (true | false) #REQUIRED
    transacted (true | false) #IMPLIED
>
<!--
        The ConsumerConfig element contains all of the information needed to
        configure and start (if specified) OpenEAI JMS consumer foundation objects
        (PointToPointConsumer or PubSubConsumer). The following is a description
        of each of the elements and attributes of ConsumerConfig:

        ConnectionFactoryName

        This specifies the lookup name for the JMS ConnectionFactory that will be
        used by this consumer. This can be either a TopicConnectionFactory or a
        QueueConnectionFactory, depending on the type of consumer you are
        configuring. A JMS connection factory contains JMS provider-specific
        information about how to connect to the JMS provider. Connection factories
        are 'administered objects' and are stored in a directory server or other
        JNDI store whose location is specified in the ProviderURL element.

        DestinationName

        This is the name of the JMS queue or topic from which this consumer will
        consumer messages. This is also an 'administered object' stored in the
        directory server or other JNDI store whose location is specified in
        the ProviderURL element.

        InitialContextFactory

        This is the fully-qualified class name of the InitialContectFactory the
        consumer will use to obtain an initial context with the directory server or
        other JNDI store where the administered objects reside.  When an
        organization uses an LDAP repository for JMS administered objects, this
        will almost always be com.sun.jndi.ldap.LdapCtxFactory.  This value will
        vary based on the repository being used to store JMS administered
        objects.

        ProviderURL

        This is the location in the directory server or other JNDI store where the
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

connection factories and destinations (administered objects) reside.

SecurityPrincipal

This element specifies the distinguished name of the directory user allowed to access the administered objects in the directory server or other JNDI store at the location specified by ProviderURL.

SecurityCredentials

This element specifies the password associated with the SecurityPrincipal.

Username

This is the JMS Provider username allowed to consume messages from the desired destination. If you are following JMS and OpenEAI preferred deployment patterns, you will leave this element blank, because this username should be specified in the ConnectionFactory maintained in the JNDI administered object store.

Password

This is the password corresponding to the JMS Provider username allowed to consume messages from the desired destination. If you are following JMS and OpenEAI preferred deployment patterns, you will leave this element blank, because the username and password should be specified in the ConnectionFactory maintained in the JNDI administered object store.

ConfigClass

The ConfigClass is the fully-qualified class name of the enterprise configuration object that will wrap all this configuration information and which will be passed to the constructor of the Consumer class to initialize the consumer. For consumers, this class name will always be org.openeai.config.ConsumerConfig.

ObjectClass

The ObjectClass element should contain the fully-qualified class name of the object to instantiate and start. This should only be one of the following classes: org.openeai.jms.consumer.PointToPointConsumer for point-to-point message consumption or org.openeai.jms.consumer.PubSubConsumer for publish/subscribe message consumption.

Commands

See the comments included with the Commands container and command element in this definition for details.

GenericResponse

This element is used to specify the 'primed' xml document that will be used in case the consumer encounters errors consuming a message and determining which command to execute for the message. This is only relevant if errors occur in the consumer prior to successfully executing the appropriate command for the message. If errors occur during the execution of a command, the command is responsible for either logging those errors (via

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

org.openeai.CoreMessaging.Sync.Error-Sync messages) or returning an error.

ThreadPoolConfig

This is the main thread pool to which the consumer adds command execution transactions. Since we use a thread pool, the onMessage method in the consumer's MessageListener does not have to wait for the command to complete execution before consuming the next message. Instead the consumer consumes a message and adds the incoming message to the thread pool where command execution takes place. These thread pools should be configured to check their status prior to adding the command execution as a job to the pool by setting the checkBeforeProcessing attribute of the thread pool to 'true'. In this way, the consumer will attempt to add the command execution to the thread pool, and if the thread pool is busy, it will stop consuming messages and wait until the thread pool can accept more command execution transaction. This reduces the risk of command executions being missed if the gateway should terminate abnormally. See the comments with the ThreadPoolConfig element in this definition for details on thread pool configuration.

DbConnectionPoolConfig

This element contains the configuration for the database connection pool used by the MessageBalancer in PubSubConsumers to ensure that only one consumer per publish/subscribe destination actually processes a message. The MessageBalancer is an infrastructure component that PubSubConsumers use to determine if another consumer is already processing the message. If so, the consumer will not process the message. Currently, the message balancer uses a database to store the MessageId of the message that is consumed to make this determination. This database connection pool is a pool of connections to that repository.

If this database connection pool is not specified, the consumer will log a warning message and all messages consumed by this consumer will be processed. If an organization runs multiple instances of gateways that consume sync messages (which should be the case to ensure high availability) this will lead to issues, because multiple consumers will be consuming and processing the same message. In essence, the same message will be processed more than once. Obviously, this would not be desirable. For example, in the case of Create-Sync messages, multiple instance of the same gateway would be attempting to create same record. This would lead to unnecessary errors on account of duplicate key violations or, even worse, duplicate data actually being stored in the application consuming the sync messages, depending on the underlying data structure into which the data is being inserted.

name

The name attribute contains the unique name by which this consumer is known to the AppConfig object. When a developer or an application wants to retrieve the consumer from the AppConfig object, it may do so by specifying this name in the getObject(String name) method of the AppConfig object.

startOnInitialization

This attribute tells the consumer whether or not to start itself when it is initialized. If this is 'false', it will be the responsibility of the application to start the consumer. In most cases, this should always be set to 'true' so the expense of starting the producer is paid only once at the

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

time the application is initialized.

The following is an example from a sample deployment descriptor:

```
<ConsumerConfig name="LoggingConsumer1">
        <DestinationName>cn=org.any-openeai-
enterprise.topic.EnterpriseSyncLogger</DestinationName>
        <Commands inboundXmlValidation="false" outboundXmlValidation="false"
writeToFile="false">
            <Command type="syncMessage" isAbsolute="true">
                <CommandName>EnterpriseSyncLogger</CommandName>
                <CommandClass>
org.openeai.implementations.gateways.enterprisesynclogger.EnterpriseSyncLoggerCommand</CommandC
lass>
                <Configuration>
                    <ProducerConfigs startOnInitialization="true">
                        <ConnectionFactoryName>cn=TCF.EnterpriseSyncLogger
</ConnectionFactoryName>
                        <InitialContextFactory>com.sun.jndi.ldap.LdapCtxFactory
</InitialContextFactory>
                        <ProviderURL>
ldaps://ldap.openerp.org:636/ou=PubSub,ou=Development,ou=AdministeredObjects,ou=Version3.5,ou=Son
icMQ,ou=Providers,ou=Messaging,dc=any-openeai-enterprise,dc=org</ProviderURL>
                        <SecurityPrincipal>
uid=EnterpriseSyncLogger,ou=Development,ou=users,ou=messaging,dc=any-openeai-
enterprise,dc=org</SecurityPrincipal>
                        <SecurityCredentials>EnterpriseSyncLogger22</SecurityCredentials>
                        <ProducerIdURL>localhost</ProducerIdURL>
                        <ConfigClass>org.openeai.config.ProducerConfig</ConfigClass>
                        <ProducerConfig name="SyncErrorPublisher">
                            <DestinationName>cn=org.any-openeai-
enterprise.topic.EnterpriseSyncErrorLogger</DestinationName>
                            <ObjectClass>org.openeai.jms.producer.PubSubProducer</ObjectClass>
                            <DefaultCommandName>EnterpriseSyncErrorLogger
</DefaultCommandName>
                        </ProducerConfig>
                    </ProducerConfigs>
                    <DbConnectionPoolConfigs>
                        <ConfigClass>org.openeai.config.DbConnectionPoolConfig</ConfigClass>
                        <ObjectClass>org.openeai.dbpool.EnterpriseConnectionPool</ObjectClass>
                        <DbConnectionPoolConfig name="SyncLoggerDbPool"
dbDriverName="oracle.jdbc.driver.OracleDriver"
dbConnectString="jdbc:oracle:thin:@oradev.openerp.org:1521:MSGDEV" dbConnectUserId="synclog"
dbConnectPassword="synclog22" dbPoolSize="5"/>
                    </DbConnectionPoolConfigs>
                    <PropertyConfigs>
                        <ConfigClass>org.openeai.config.PropertyConfig</ConfigClass>
                        <PropertyConfig name="SyncErrorSyncProperties">
                            <ConfigClass>org.openeai.config.PropertyConfig</ConfigClass>
                            <Property>
                                <PropertyName>SyncErrorSyncPrimedDocumentUri</PropertyName>
                                <PropertyValue>http://xml.openeai.org/message/releases/com/any-
erp-vendor/CoreMessaging/Sync/1.0/xml/Error-Sync.xml</PropertyValue>
                            </Property>
                        </PropertyConfig>
                        <PropertyConfig name="SyncLoggerProperties">
                            <ConfigClass>org.openeai.config.PropertyConfig</ConfigClass>
                            <Property>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                                                    <PropertyName>SyncErrorSyncPrimedDocumentUri</PropertyName>
                                                    <PropertyValue>http://xml.openeai.org/message/releases/com/any-
erp-vendor/CoreMessaging/Sync/1.0/xml/Error-Sync.xml</PropertyValue>
                                                </Property>
                                                <Property>
                                                    <PropertyName>xmlFileSpec</PropertyName>
                                                    <PropertyValue>./message-store</PropertyValue>
                                                </Property>
                                            </PropertyConfig>
                                        </PropertyConfigs>
                                    </Configuration>
                                </Command>
                            </Commands>
                            <ObjectClass>org.openeai.jms.consumer.PubSubConsumer</ObjectClass>
                            <DbConnectionPoolConfig name="PubSubMessageBalancer1"
dbDriverName="oracle.jdbc.driver.OracleDriver"
dbConnectString="jdbc:oracle:thin:@oradev.openerp.org:1521:MSGDEV" dbConnectUserId="synclog"
dbConnectPassword="synclog22" dbPoolSize="3"/>
        </ConsumerConfig>

        Since consumers all use the OpenEAI Consumer Pattern (an implementation of
        the command pattern) there is really only one Java client ever needed to
        implement any consumer. These consumers are simply configured, and possibly
        started based on the AppConfig object that gets built. The commands that a
        consumer is configured to invoke distinguishes the functionality of the
        consumer.-->
<!ELEMENT ConsumerConfig (ConnectionFactoryName?, DestinationName, InitialContextFactory?,
ProviderURL?, SecurityPrincipal?, SecurityCredentials?, Username?, Password?, Commands,
ConfigClass?, ObjectClass, GenericResponse?, ThreadPoolConfig?, DbConnectionPoolConfig?)>
<!ATTLIST ConsumerConfig
    name CDATA #REQUIRED
    startOnInitialization (true | false) #IMPLIED
    transacted (true | false) #IMPLIED
>
<!ELEMENT GenericResponse (DocumentURI)>
<!ATTLIST GenericResponse
    validate (true | false) #REQUIRED
>
<!ELEMENT DocumentURI (#PCDATA)>
<!--
        The Commands container specifies the functionality that this Consumer
        supports. Like all container elements Commands allows the deployer to
        specify default values at the container level that are applied to the
        elements and attributes of each Command contained within it unless
        these default values are overridden by specifying different values
        within an individual Command.

        Each Command listed in this collection maps to Java objects that
        inherit from the ConsumerCommand super class. If a developer wishes to add
        common functionality to all commands supported by a consumer, they can also
        extend ConsumerCommand and inherit from that new level as well.

        When the Consumer receives a message, it looks at that message and calls the
        execute method on the Command (Java object) associated with it in the JMS
        property called COMMAND_NAME. These Java objects should all implement either
        the SyncCommand or the RequestCommand interfaces. If command produces a reply
        to a request, it should implement the RequestCommand interface. If no reply
        is expected, the command should implement the SyncCommand interface. Each of
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
            these interfaces define one method that must be implemented. RequestCommand
            defines an execute method that must return a JMS Message while SyncCommand
            defines an execute method that returns nothing.

            The consumer knows which command to execute by looking at the user-defined
            JMS property called COMMAND_NAME. It knows if the message coming in expects
            a reply by looking at the 'reply-to' queue specified in the message. If there
            is no 'reply-to' queue in the JMS message, the consumer knows no reply is
            needed. If it detects a 'reply-to' queue, then it knows it must execute the
            command and return the reply. If there is no COMMAND_NAME present, the
            consumer will execute a 'Default' command associated with the consumer.
            This allows you to have one consumer that supports multiple business functions.

            The COMMAND_NAME is set by the OpenEAI message-aware business objects as
            specified in the CommandName element of MessageObjectConfig when a
            message action is performed on the object (query, create, update, delete,
            or generate).
-->
<!ELEMENT Commands (Command+)>
<!ATTLIST Commands
      inboundXmlValidation (false | true) #REQUIRED
      outboundXmlValidation (false | true) #REQUIRED
      writeToFile (false | true) #REQUIRED
      messageDumpDirectory CDATA #IMPLIED
>
<!--
            The Command element contains all of the information needed to configure
            OpenEAI consumer command foundation objects. The following is a description
            of each of the elements and attributes of a Command:

            CommandName

            This element is the name of the command.  If the consumer finds a COMMAND_NAME
            JMS property on the message it consumed that matches this, it will execute this
            command.

            CommandClass

            This is the fully-qualified class name of the command class that implements
            SyncCommand or RequestCommand. This is the class on which the execute method
            will be called when a message with a COMMAND_NAME of CommandName is consumed
            by the consumer.

            Configuration

            Since commands are considered "mini" applications, they also have an AppConfig
            object associated to them.  This Configuration element provides that AppConfig object
            with the appropriate runtime objects that the command will need.  The Command can then
            retrieve this AppConfig to access the objects it needs.  This approach is used for both
            ConsumerCommands and ScheduledCommands.

            MessagingComponents

            A command can take many different forms. They can be very simple or very
            complex. For this reason, they are given the ability to perform any of the
            functions that any of our other message components can perform. Generally,
            the MessagingComponents element won't be used except when developing commands
            to implement messaging infrastructure processing such as when developing
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

message routers or proxies. This is an advanced topic.

type

This attribute specifies the type of the command: either 'scheduled', 'requestMessage' or 'syncMessage'.

All commands of type 'scheduled' implement the org.openeai.afa.ScheduledCommand interface and must implement the execute method specified by that interface.

All commands of type 'requestMessage' implement the org.openeai.jms.consumer.commands.RequestCommand interface and must implement an execute method that returns a JMS Message.

All commands of type 'syncMessage' implement the org.openeai.jms.consumer.commands.RequestCommand interface and must implement an execute method. Additionally, the configuration for a command of type 'syncMessage' must include a SyncErrorPublisher in its configuration that will be used by the command to publish org.openeai.CoreMessaging.Sync.Error-Sync messages if errors occur during the processing of a sync message. The consumer will not initialize if this SyncErrorPublisher is not specified.

isDefault

This attribute specifies whether the command is the default command the consumer should execute if there is no COMMAND_NAME JMS property associated with a message received.

isAbsolute

This attribute specifies whether the command is the ONLY command this consumer will ever execute. That is, no matter what COMMAND_NAME property is associated with the message consumed, it will execute this command always.

inboundXmlValidation

This attribute tells the ConsumerCommand whether or not it should validate the contents of the message being passed to the command. This will be executed in the initializeInput method found in the Command super class or it can also be performed by the command itself if the initializeInput method is not used.

outboundXmlValidation

This attribute tells the ConsumerCommand whether or not is should validate the contents of a reply message document (or a message document it is routing as in the case of messaging infrastructure applications like TransRoute. For example, a reply message can be validated before sending the reply in response to a request.

writeToFile

This attribute tells the ConsumerCommands whether or not it should write the incoming message to a file. This parameter is checked and acted upon by the initializeInput method in the ConsumerCommand

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
            super class.

            messageDumpDirectory

            This specifies the directory to which messages should be written if the value
            of the writeToFile attribute is 'true'.-->
<!ELEMENT Command (CommandName, CommandClass, Configuration?, MessagingComponents?)>
<!ATTLIST Command
      type (requestMessage | syncMessage | scheduled) #REQUIRED
      isDefault (true | false) "false"
      isAbsolute (true | false) "false"
      inboundXmlValidation (true | false) "false"
      outboundXmlValidation (true | false) "false"
      writeToFile (true | false) "false"
      messageDumpDirectory CDATA #IMPLIED
>
<!ELEMENT CommandName (#PCDATA)>
<!ELEMENT CommandClass (#PCDATA)>
<!--

            The LoggerConfigs element consists of one LoggerConfig element. It is
            a container in which you can configure a Logger for use within an
            application.  Once this Logger is instantiated and initialized by AppConfig,
            all OpenEAI foundation components and your objects that extend them
            inherit the same logging configuration, because the Logger object is a
            static variable in OpenEaiObject, the ancestor from which all OpenEAI
            objects inherit.-->
<!ELEMENT LoggerConfigs (LoggerConfig)>
<!--
            The LoggerConfig Element

            When the AppConfig object sees a LoggerConfig element, it instantiates the
            object specified by the ConfigClass element and then calls the initialization
            method for the Logger object inherited by all objects running in the current
            JVM. Therefore, there is never any need for an application to attempt to
            retrieve the LoggerConfig from the AppConfig object. That is all handled
            automatically because the Logger object that all objects running in the JVM
            will use is configured once, by the AppConfig object.

            The ConfigClass element tells the AppConfig object which Java object to use
            to store the configuration properties defined in the Property elements.  An
            example in this case would be:

            <ConfigClass>org.openeai.config.LoggerConfig</ConfigClass>

            The LoggerConfig object is an OpenEAI foundation component that takes the
            information found in this XML element and configures the Logger object.

            The Property elements are property values that get used to configure the
            Logger. They are specified in a name/value pair and are ultimately turned
            into a Java Properties object. These properties happen to be specific to
            Log4J because that's the logging API that the OpenEAI project has been using
            currently. Log4J is excellent. If you use a different logging API, these
            values would be different. Here's a complete example of a LoggerConfig
            entry from the sample deployment descriptor:

            <LoggerConfig name="XmlBasicPersonLogger">
                  <ConfigClass>org.openeai.config.LoggerConfig</ConfigClass>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                <Property>
                    <PropertyName>log4j.rootCategory</PropertyName>
                    <PropertyValue>INFO, stdout</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>log4j.appender.stdout</PropertyName>
                    <PropertyValue>org.apache.log4j.FileAppender</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>log4j.appender.stdout.File</PropertyName>
                    <PropertyValue>System.out</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>log4j.appender.stdout.layout</PropertyName>
                    <PropertyValue>org.apache.log4j.PatternLayout</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>log4j.appender.stdout.layout.ConversionPattern</PropertyName>
                    <PropertyValue>%-4d{ISO8601} %-5p [%t] %37c %3x - %m%n</PropertyValue>
                </Property>
            </LoggerConfig>-->
<!ELEMENT LoggerConfig (ConfigClass, Property+)>
<!ATTLIST LoggerConfig
      name CDATA #REQUIRED
>
<!--
            This is a container for specifying application specific properties for an application,
            RequestCommand, SyncCommand or ScheduledCommand.-->
<!ELEMENT PropertyConfigs (ConfigClass, PropertyConfig+)>
<!--
            The AppConfig object builds the PropertyConfig Java object from the
            PropertyConfig element. The PropertyConfig object stores the
            PropertyName/PropertyValue elements in a standard Java Properties object.
            This Properties object is available to the application via AppConfig by name.
            Therefore, an application can have many "properties" distinguished by name.
            This allows us to group properties and document the application further
            using the deployment descriptor.

            The following is an example PropertyConfig from a sample deployment
            descriptor:

            <PropertyConfig name="PaymasterGateway">
                <ConfigClass>org.openeai.config.PropertyConfig</ConfigClass>
                <Property>
                    <PropertyName>extractFileName</PropertyName>
                    <PropertyValue>/export/home/EnterpriseConsumer/extract/Big-
PaymasterExtract.test.txt</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>maxLinesToRead</PropertyName>
                    <PropertyValue>2000</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>maxProducers</PropertyName>
                    <PropertyValue>5</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>useThreads</PropertyName>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                    <PropertyValue>true</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>startingId</PropertyName>
                    <PropertyValue>400000107</PropertyValue>
                </Property>
                <Property>
                    <PropertyName>action</PropertyName>
                    <PropertyValue>create</PropertyValue>
                </Property>
            </PropertyConfig>

            The following example demonstrates the retrieval of a property from the
            AppConfig object:

            PropertyConfig propConfig = (PropertyConfig)aConfig.getObject("PaymasterGateway");
            Properties props = propConfig.getProperties();
            m_extractFileName = props.getProperty("extractFileName","");

            PropertyConfig pConfig2 = new PropertyConfig();
            pConfig2 = (PropertyConfig)m_appConfig.getObject("OtherProperties");
            logger.info("Environment: " + pConfig2.getProperties().getProperty("environment","Not found"));-->
<!ELEMENT PropertyConfig (ConfigClass?, Property+)>
<!ATTLIST PropertyConfig
    name CDATA #REQUIRED
>
<!ELEMENT Property (PropertyName, PropertyValue)>
<!ELEMENT PropertyName (#PCDATA)>
<!ELEMENT PropertyValue (#PCDATA)>
<!--
            The MessageObjectConfigs container specifies MessageObjectConfig elements
            for the message-aware Java objects that the application or gateway will
            use within its runtime. Like all container elements, MessageObjectConfigs
            allows the deployer to specify default values at the container level that
            are applied to the elements and attributes of each MessageObjectConfig
            contained within it unless these default values are overridden by specifying
            different values within an individual MessageObjectConfig.-->
<!ELEMENT MessageObjectConfigs (SenderAppId, Authentication, ConfigClass,
EnterpriseObjectDocument?, LayoutManager?, MessageObjectConfig+)>
<!ATTLIST MessageObjectConfigs
    deferInitialization (true | false) #IMPLIED
    xmlDocumentValidation (true | false) #REQUIRED
    translationType (all | application) #REQUIRED
>
<!--
            The MessageObjectConfig element contains all of the information needed to
            configure a message-aware Java object. The following is a description of
            each of the elements and attributes of a MessageObjectConfig:

            SenderAppId

            This value will be used in any messages produced by the message object to
            indicate the name of the application sending the message. Therefore, this
            should typically be the same value as the application name that uses these
            message objects. It is a part of the ControlArea in the message. This will
            be automatically applied to the message when it is produced using this
            message object. See the description of the SenderAppId element in the OpenEAI
            segments file for more details.
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Authentication

This is the value for the Authentication element that is placed into the messages that this Java message object is used to produce. It is part of the ControlArea of the message. This will be automatically applied to the message when it is produced using this message object.

CommandName

This tells a consuming application what command to execute when it receives a message that was produced using this message object. See the description of the consumer and the command configurations in this definition. This value corresponds a Command's "CommandName" element and is the value placed on the COMMAND_NAME JMS property when an action is performed with the message object (query, create, delete, update, generate etc.).

PrimedXmlDocuments

These documents are used to specify static information about the messages that will be produced. This way, we don't have to rebuild the message we're producing from scratch. These documents contain information in their ControlArea elements that will exist for all messages that get produced of a Certain action type. Any dynamic information (like SenderAppId, Authentication, Datetime, etc.) will be automatically changed in the message when it gets produced. This information is retrieved from the object as it builds the document it is producing/publishing.

ConfigClass

The ConfigClass is the fully-qualified class name of the enterprise configuration object that will wrap all this configuration information. For message objects, this class name will always be org.openeai.config.MessageObjectConfig.

ObjectClass

The ObjectClass element should contain the fully-qualified class name of the object to instantiate. For example, edu.uillinois.aits.eai.moa.jmsobjects.person.v1_0.BasicPerson.

EnterpriseObjectDocument

This element includes the attributes docUri, ignoreMissingFields, and ignoreValidation a URI to the enterprise object (EO) document that will be used to by the message object to set its field formatting, translation, scrubbing and other rules.

docUri

This specifies the location of the enterprise objects (EO) document to be used by this Java message object. These are typically stored on a web server, a local file system, or a remote file system that is exported and mounted locally.

ignoreMissingFields

This attribute specifies whether the Java message object should ignore fields

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

that do not appear in the enterprise object (EO) document or not. This attribute will be removed in the future since EO documents are automatically generated and there will most likely never be a reason for an EO document to be incomplete.

ignoreValidation

This attribute toggles enterprise object (EO) validation on or off. When this is turned on (value is 'true'), no formatting, translations or scrubbing will be performed on the data as it is put into the Java message object via the setter methods. Note that this validation is completely different from XML document validation. This validation occurs when the setter methods are called on the Java message objects.

LayoutManager

In some cases other methods of building and serializing objects will be necessary. This is accomplished with a layout manager. The LayoutManager element specifies the specialized classes that perform the real work of the build and serialization methods like building the object from records in an extract file or serializing the object as line(s) in an extract file. As more requirements are identified, the OpenEAI Project will add more generalized layout managers...like the XML, extract file, and stored procedure call layout managers that exist today. However, layout managers do not have to be as universally applicable and general as these. You may need to build a really strange thing with your message object data, and you can develop your own layout manager to do exactly what you need as well.

When the application configures the Java message objects via the AppConfig object, it tells those objects which LayoutManager to use by default. Since all objects must be able to serialize themselves from and to XML, they all must specify at least an XML layout manager. See the comments with the LayoutManager element in this definition for details on configuring a layout manager.

name

This is the name of the object as it will be known to the AppConfig object. Generally, this should be just be a short version of the Java object name such as "BasicPerson" or "EmergencyContact", but this is not a requirement. The name can be anything the developer wishes it to be. For example, if you needed to have two BasicPerson objects pre-configured differently by AppConfig, you would want to name them differently because you are going to retrieve them from AppConfig by name. If you named them the same, the second one in the list would overwrite the first!

xmlDocumentValidation

This element specifies whether or not to perform XML document validation with a validating XML parser as messages are produced with this message object. Generally, this should only be true during development and testing, because of the performance implications of XML validation.

translationType

This indicates the type of translations that will be performed on values being set on the object. The value "all" indicates that we wish to allow any input values found in the enterprise object (EO) document to be mapped to the enterprise value for a given field within this object. The value

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

"application" says we only want to accept input values associated with a specific application and translate them to enterprise values. In this case any other input values will result in an exception being thrown during the building of an object. See the detailed comments within enterprise object (EO) document time definition for more details on translations.

The following is an example of the configuration for a Java message object within its MessageObjectConfigs container element from a sample deployment descriptor:

```
<MessageObjectConfigs xmlDocumentValidation="false" translationType="application">
    <SenderAppId>org.openeai.TestSuiteApplication</SenderAppId>
    <Authentication>
        <UserId>TestSuiteApplication</UserId>
    </Authentication>
    <ConfigClass>org.openeai.config.MessageObjectConfig</ConfigClass>
    <EnterpriseObjectDocument
docUri="http://xml.openeai.org/xml/configs/messaging/environments/development/enterpriseobjects/1.0/Ent
erpriseObjects.xml" ignoreMissingFields="true" ignoreValidation="false"/>
    <LayoutManager>
        <InputLayout type="xml">
            <ObjectClass>org.openeai.layouts.XmlLayout</ObjectClass>
        </InputLayout>
        <OutputLayout type="xml">
            <ObjectClass>org.openeai.layouts.XmlLayout</ObjectClass>
        </OutputLayout>
    </LayoutManager>
    <MessageObjectConfig name="BasicPerson.v1_0">
        <CommandName>BasicPerson</CommandName>
        <PrimedXmlDocuments>
            <PrimedXmlDocument
type="query">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Query-Request.xml</PrimedXmlDocument>
            <PrimedXmlDocument
type="create">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Create-Request.xml</PrimedXmlDocument>
            <PrimedXmlDocument
type="delete">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Delete-Request.xml</PrimedXmlDocument>
            <PrimedXmlDocument
type="update">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Update-Request.xml</PrimedXmlDocument>
            <PrimedXmlDocument
type="createSync">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Create-Sync.xml</PrimedXmlDocument>
            <PrimedXmlDocument
type="updateSync">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Update-Sync.xml</PrimedXmlDocument>
            <PrimedXmlDocument
type="deleteSync">file://localhost/checkout/openeai/project/message/releases/com/any-erp-
vendor/Person/BasicPerson/1.0/xml/Delete-Sync.xml</PrimedXmlDocument>
        </PrimedXmlDocuments>
        <ObjectClass>com.any_erp_vendor.moa.jmsobjects.person.v1_0.BasicPerson</ObjectClass>
        <EnterpriseObjectDocument
docUri="file://localhost/checkout/openeai/project/configs/messaging/environments/examples/EnterpriseObje
cts/com/any-erp-vendor/Person/1.0/BasicPersonEO.xml" ignoreMissingFields="true"
ignoreValidation="false"/>
    </MessageObjectConfig>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
            </MessageObjectConfigs>


            The following is an example of retrieving a Java message object by type from
            AppConfig:

            BasicPerson bPerson = null;
            bPerson = (BasicPerson)aConfig.getObject("BasicPerson");

            BasicPerson bPerson = (BasicPerson)aConfig
                  .getObjectByType("com.any_erp_vendor.moa.jmsobjects.person.v1_0.BasicPerson");-->
<!ELEMENT MessageObjectConfig (SenderAppId?, Authentication?, CommandName,
PrimedXmlDocuments*, ConfigClass?, ObjectClass, EnterpriseObjectDocument?, LayoutManager?)>
<!ATTLIST MessageObjectConfig
      name CDATA #REQUIRED
      deferInitialization (true | false) #IMPLIED
      xmlDocumentValidation (true | false) #IMPLIED
      translationType (all | application) #IMPLIED
>
<!ELEMENT EnterpriseObjectDocument EMPTY>
<!ATTLIST EnterpriseObjectDocument
      docUri CDATA #REQUIRED
      ignoreMissingFields (false | true) #REQUIRED
      ignoreValidation (false | true) #REQUIRED
>
<!ELEMENT ConfigClass (#PCDATA)>
<!ELEMENT SenderAppId (#PCDATA)>
<!ELEMENT Authentication (UserId, Signature?)>
<!ELEMENT UserId (#PCDATA)>
<!ELEMENT Signature (#PCDATA)>
<!ELEMENT PrimedXmlDocuments (PrimedXmlDocument+)>
<!ELEMENT PrimedXmlDocument (#PCDATA)>
<!ATTLIST PrimedXmlDocument
      type (query | generate | generateSync | create | createSync | update | updateSync | delete | deleteSync |
provide | response) #REQUIRED
>
<!--
            The LayoutManager element is comprised of one or more InputLayout and
            one or more OutputLayout elements. These are specified so the object knows
            how to build itself via the buildObjectFromInput(Object) method and how to
            serialize itself via the buildOutputFromObject() method. When the AppConfig
            object is initialized, it looks at information in this location of the document
            and stores all LayoutManagers associated with the object. The LayoutManager
            that's listed first in the list is the "Default" LayoutManager to be used.
            All objects must have an XML LayoutManager.  During the runtime of the
            application, it can switch between layout managers as needed.
            They both contain the following elements and attributes:

            ObjectClass

            This element contains the fully-qualified class name of the class that
            implements the LayoutManager functionality.

            type

            This attribute specifies the type of layout that is being used. This will be
            stored in the Java object along with the layout manager implementation so
            developers can specify the layout manager to use on the object when building
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
            the object from an input (buildObjectFromInput method) or serializing the
            object to an output (buildOutputFromObject method).

            name

            The name of the layout manager that will be searched for during
            initialization. If this value is left blank, the name of the message object
            (such as BasicPerson) will be used as the layout manager name. The name
            attribute is only needed if an object will need layout managers in addition
            to the XML layout managers.-->
<!ELEMENT LayoutManager (InputLayout+, OutputLayout+)>
<!ELEMENT InputLayout (ObjectClass)>
<!ATTLIST InputLayout
      type (xml | other | extract) #REQUIRED
      name CDATA #IMPLIED
>
<!ELEMENT OutputLayout (ObjectClass)>
<!ATTLIST OutputLayout
      type (xml | other | extract | spcalls) #REQUIRED
      name CDATA #IMPLIED
>
<!--
            The ThreadPoolConfigs container specifies ThreadPoolConfig elements
            for use in configuring and initializing individual thread pools at
            runtime. Applications and commands may use several different ThreadPools
            during their execution. This allows the deployer to pre-configure these
            ThreadPools and retrieve them from AppConfig just like all other foundation
            objects. Like all container elements, ThreadPoolConfigs allows the deployer
            to specify some default values at the container level that are applied to the
            elements and attributes of each ThreadPoolConfig contained within it unless
            these default values are overridden by specifying different values within an
            individual ThreadPoolConfig.-->
<!ELEMENT ThreadPoolConfigs (ConfigClass, ObjectClass, ThreadPoolConfig+)>
<!--
            The ThreadPoolConfig element contains all of the information needed to
            configure a ThreadPool object. The following is a description of each
            of the elements and attributes of a ThreadPoolConfig:

            ConfigClass

            The ConfigClass is the fully-qualified class name of the enterprise
            configuration object that will wrap all this configuration information.
            For thread pools, this class name will always be
            org.openeai.config.ThreadPoolConfig.

            ObjectClass

            The ObjectClass element should contain the fully-qualified class name of
            the object to instantiate. For thread pools, this class name will always
            be org.openeai.threadpool.ThreadPoolImpl.

            name

            The name of the ThreadPool as it will be known to AppConfig. This is what the
            developer uses to retrieve this specific thread pool from AppConfig.

            maxThreads
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

The maximum number of threads to allocate to the thread pool. This is the maximum number of threads that will ever be in progress at the same time.

minThreads

The minimum number of threads that can be allocated to the thread pool.

maxIdleTime

The maximum amount of time the thread pool will wait for a thread to complete before allocating another thread to the pool.

checkBeforeProcessing

This tells the thread pool whether or not it should check the number of threads currently in use before attempting to add any more jobs to the pool. If the number of threads in use (jobs in progress) is equal to the maxThreads, it will throw a ThreadPoolException indicating that no more threads can be allocated until some of the jobs currently in progress have finished. If this is false, the thread pool will continue to accept incoming jobs, but will not be able to process them until some have finished. In some cases, this can lead to a risky situation in which there are pending jobs waiting for a thread to become available in which to execute. In this case, if the process dies for some reason, those pending jobs are lost. For example, OpenEAI consumers should initialize their thread pools with this flag set to 'true' and only add jobs to the thread pool if the pool is ready to accept and process those jobs. Otherwise, the consumer stops consuming messages and waits for some threads to free up in the pool.

The following is an example ThreadPoolConfig from a sample deployment descriptor:

```
<ThreadPoolConfigs>
    <ThreadPoolConfig name="ProcessLine" maxThreads="150" minThreads="0"
maxIdleTime="1">
        <ConfigClass>org.openeai.config.ThreadPoolConfig</ConfigClass>
        <ObjectClass>org.openeai.threadpool.ThreadPoolImpl</ObjectClass>
    </ThreadPoolConfig>
    <ThreadPoolConfig name="ProcessEmployee" maxThreads="30" minThreads="0"
maxIdleTime="1">
        <ConfigClass>org.openeai.config.ThreadPoolConfig</ConfigClass>
        <ObjectClass>org.openeai.threadpool.ThreadPoolImpl</ObjectClass>
    </ThreadPoolConfig>
    <ThreadPoolConfig name="ProcessPerson" maxThreads="15" minThreads="0"
maxIdleTime="1">
        <ConfigClass>org.openeai.config.ThreadPoolConfig</ConfigClass>
        <ObjectClass>org.openeai.threadpool.ThreadPoolImpl</ObjectClass>
    </ThreadPoolConfig>
</ThreadPoolConfigs>
```

The following is an example of retrieving a ThreadPool from AppConfig:

```
    ThreadPool tPool = (ThreadPool)m_appConfig.getObject("ProcessLine");-->
<!ELEMENT ThreadPoolConfig (ConfigClass?, ObjectClass?)>
<!ATTLIST ThreadPoolConfig
    name CDATA #REQUIRED
    maxThreads CDATA #REQUIRED
    minThreads CDATA #REQUIRED
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
    maxIdleTime CDATA #REQUIRED
    checkBeforeProcessing (true | false) "false"
>
<!--
        The DbConnectionPoolConfigs container specifies DbConnectionPoolConfig
        elements for use in configuring and initializing individual database
        connection pools at runtime. Applications and commands may use several
        different database connection pools during their execution. This allows
        the deployer to pre-configure these connection pools and retrieve them
        from AppConfig just like all other foundation objects. When AppConfig
        initializes these pools, they are "ready-to-use" database connections.
        Like all container elements, DbConnectionPoolConfigs allows the deployer
        to specify some default values at the container level that are applied to the
        elements and attributes of each DbConnectionPoolConfig contained within it
        unless these default values are overridden by specifying different values
        within an individual DbConnectionPoolConfig.
-->
<!ELEMENT DbConnectionPoolConfigs (ConfigClass, ObjectClass, DbConnectionPoolConfig+)>
<!--
        The DbConnectionPoolConfig element contains all of the information needed
        to configure an EnterpriseConnectionPool object. The following is a
        description of each of the elements and attributes of a
        DbConnectionPoolConfig:

        ConfigClass

        The ConfigClass is the fully-qualified class name of the enterprise
        configuration object that will wrap all this configuration information.
        For database connection pools, this class name will always be
        org.openeai.config.DbConnectionPoolConfig.

        ObjectClass

        The ObjectClass element should contain the fully-qualified class name of
        the object to instantiate. For database connection pools, this class name
        will always be org.openeai.enterprise.dbpool.EnterpriseConnectionPool.

        name

        The name of the database connection pool. This is what the developer uses
        to retrieve this specific connection pool from AppConfig.

        dbDriverName

        This is the name of the JDBC implementation to use for the connections in
        this pool; for example, oracle.jdbc.driver.OracleDriver.

        dbConnectString

        This is the name of the target database to which the connections in this pool
        should connect; for example, jdbc:oracle:thin:@dbserver.openeai.org:0989:MSGDEV.
        This will be a JDBC provider-specific connect string.

        dbConnectUserId

        This is the name of the user as which to connect to the database.

        dbConnectPassword
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

This is the password to use for the user.

dbPoolSize

This is the initial number of connections to allocate to the pool.

dbPoolMaxSize

If specified, this is the maximum number of connections to allocate to the pool. If dbPoolMaxSize is not specified and additional available connections are needed, the pool will create additional connections dynamically indefinitely. If specified, the pool will only create additional connections dynamically until this maximum number of connections is reached.

dbVerificationString

This is a string used to externalize the verification of database connections as they are retrieved from the pool. Not all vendors who provide a JDBC implementation support the isClosed() method that JDBC specifies. There are cases where additional verification is required. For example, for Microsoft SQLServer the string that is typically used is "select getDate()". If this value is not specified, and the dbDriver is an Oracle implementation, the connection pool will attempt to use a default verification string. Currently, this value is set to "declare x number; begin x:=1; end;" for all Oracle drivers. This will attempt to declare a stored procedure which will fail if the db connection has been lost. You may be wondering why we didn't use the old Oracle standby like "select * from DUAL". Well, it turns out that those operations are pretty resource intensive for Oracle; it is not something that you want to do regularly every time you want to verify that status of a connection. If the value of dbVerificationString is not specified and the dbDriver is not Oracle, the connection pool will simply use the isClosed() method that JDBC specifies, which as mentioned above is not implemented by some JDBC providers. You will want to consult your JDBC provider's documentation to determine if your provider implements the isClosed() method adequately or to figure out what a good connection verification string would be to use for your database server.

The following is an example of two DbConnectionPoolConfig elements within their DbConnectionPoolConfigs container element from a aample deployment descriptor:

```
<DbConnectionPoolConfigs>
    <ConfigClass>org.openeai.config.DbConnectionPoolConfig</ConfigClass>
    <ObjectClass>org.openeai.enterprise.dbpool.EnterpriseConnectionPool</ObjectClass>

    <DbConnectionPoolConfig name="InstIdServiceDbPool"
dbDriverName="oracle.jdbc.driver.OracleDriver"
dbConnectString="jdbc:oracle:thin:@dbdev.openeai.org:3987:MSGDEVP1" dbConnectUserId="idservice"
dbConnectPassword="secretpw" dbPoolSize="2"/>

    <DbConnectionPoolConfig name="IcardDbPool"
dbDriverName="com.microsoft.jdbc.sqlserver.SQLServerDriver"
dbConnectString="jdbc:microsoft:sqlserver://dbdev.openeai.org:1111" dbConnectUserId="uid"
dbConnectPassword="secretpw" dbPoolSize="2" dbVerificationString="select getdate()"/>
</DbConnectionPoolConfigs>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
        The following is an example of retrieving these database connection pools
        from AppConfig:

        EnterpriseConnectionPool connPool1 =
              (EnterpriseConnectionPool)m_appConfig.getObject("InstIdServiceDbPool");
        EnterpriseConnectionPool connPool2 =
              (EnterpriseConnectionPool)m_appConfig.getObject("IcardDbPool");
        try {
          java.sql.Connection conn = connPool1.getConnection();
          if (conn != null) {
            logger.info("Phase2ModAppl, Got a connection from the InstIdService DB Pool...");
          }
          conn = connPool2.getConnection();
          if (conn != null) {
            logger.info("Phase2ModAppl, Got a connection from the IcardDbPool DB Pool...");
          }
        }
         catch (SQLException e) {
          logger.fatal(e.getMessage(), e);
        }

        Note:  these examples do not show the use of the
        getExclusiveConnection method which should be used if the connection
        being retrieved is to be used for multiple database actions within a single
        transaction.  See the org.openeai.dbpool package Javadoc for more information.-->
<!ELEMENT DbConnectionPoolConfig (ConfigClass?, ObjectClass?)>
<!ATTLIST DbConnectionPoolConfig
    name CDATA #REQUIRED
    dbDriverName CDATA #REQUIRED
    dbConnectString CDATA #REQUIRED
    dbConnectUserId CDATA #REQUIRED
    dbConnectPassword CDATA #REQUIRED
    dbPoolSize CDATA #REQUIRED
    dbPoolMaxSize CDATA #IMPLIED
    dbVerificationString CDATA #IMPLIED
>
<!ELEMENT FullName (#PCDATA)>
<!ELEMENT ShortName (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT MessageGateways (MessageGateway*)>
<!ELEMENT MessageGateway (FullName, ShortName?, Description, Configuration)>
<!ATTLIST MessageGateway
    id CDATA #REQUIRED
    type (daemon | scheduled) #REQUIRED
>
<!ELEMENT Servlets (Servlet*)>
<!ELEMENT Servlet (FullName, ShortName?, Description, Configuration)>
<!ATTLIST Servlet
    id CDATA #REQUIRED
>
```

End Deployment.dtd

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

OpenEAI Enterprise Object Document

The OpenEAI Enterprise Object Document (EO documents) is an XML document that describes an organization's enterprise message objects from a business perspective. Structurally, it matches the definition of the object in the DTD. However, it goes much further than the object's definition by way of a DTD or Schema. These documents allow an organization to specify very specific business rules on each field within an enterprise message object. These rules are implemented by the EnterpriseFields OpenEAI foundation object (org.openeai.config.EnterpriseFields). Each object within an organization's MOA contains a reference to this object and the rules specified in these EO documents. Each complex object within an MOA has a corresponding EO document generated for it.

The EO documents themselves are generated when an organization's MOA is generated by way of the OpenEAI MOAGenerationApplication. However, the EO document that gets generated does not contain all rules for that object and its fields. Some of those rules are impossible to generate automatically. However, the auto-generated EO document provides a consistent, properly formatted starting place.

The EO documents provide the full definition, including business rules, for an enterprise message object within an organization. This means it includes the structure of the object as well as any business rules that should be applied to fields within that object.

Following is the document type definition for EO documents. The definition includes a detailed description of each section of an EO document.

Begin EnterpriseObjects.dtd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v3.5 NT (http://www.xmlspy.com) by Tod Jackson (Administrative Information
Technology Services) -->
<!--	$Revision: 1.6 $
	$Date: 2003/01/29 00:00:37 $
	$Source: /cvs/repositories/openeai/project/xml/dtd/1.0/EnterpriseObjects.dtd,v $
-->
<!-- This file is part of the OpenEAI Application Foundation or
	OpenEAI Message Object API created by Tod Jackson
	(tod@openeai.org) and Steve Wheat (steve@openeai.org).

	Copyright (C) 2003 The OpenEAI Software Foundation

	This library is free software; you can redistribute it and/or
	modify it under the terms of the GNU Lesser General Public
	License as published by the Free Software Foundation; either
	version 2.1 of the License, or (at your option) any later version.

	This library is distributed in the hope that it will be useful,
	but WITHOUT ANY WARRANTY; without even the implied warranty of
	MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
	Lesser General Public License for more details.

	You should have received a copy of the GNU Lesser General Public
	License along with this library; if not, write to the Free Software
	Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

```
$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

        For specific licensing details and examples of how this software
        can be used to build commercial integration software or to implement
        integrations for your enterprise, visit http://www.OpenEai.org/licensing.
 -->
<!--Release History
        1/28/2003       tod@openeai.org     Original release.
                        steve@openeai.org
-->
<!ENTITY % DEPLOYMENT SYSTEM "Deployment.dtd">
%DEPLOYMENT;
<!ELEMENT EnterpriseObjects (ExternalFieldMapping?, IncludeList?, ObjectDefinition+)>
<!--
        IncludeList includes zero or more child elements called DocumentUri. All
        complex objects must have an EO document associated with them. If an object
        includes within its definition another complex object, it must include that
        complex child object's definition in order to make itself complete at
        runtime. The IncludeList allows a parent object to include as many child
        object EO document references as it must to make its definition complete at
        runtime. This is a recursive include. This means that if an object includes
        another object and that object includes another (and so on),
        all object definitions will be available at runtime.-->
<!ELEMENT IncludeList (DocumentUri+)>
<!ELEMENT DocumentUri (#PCDATA)>
<!--
        The ExternalFieldMapping element is a future enhancement that will
        allow organizations to specify field mappings in an external source.
        This might include things like databases or other services that are
        authoritative for field mapping information.  This element is not
        used by the OpenEAI APIs at the time of this writing.-->
<!ELEMENT ExternalFieldMapping (Configuration)>
<!--
        ObjectDefinition is the main component within the EO document. It defines
        the data structure of the message object and provides the framework for
        specifying the field-level rules that can be enforced by the OpenEAI
        enterprise field and message object foundation components. The following
        are the direct child elements and attributes of an ObjectDefinition:

        name

        The name attribute should contain the name of the object that is defined in
        this object definition. This should exactly match the information contained
        in the XML constraint that defines this object's XML structure as well as match
        the Java object that implements it. For example, names of message objects may
        be BasicPerson, EmergencyContact, InstitutionalIdentity, etc.

        Description

        The description element should contain some information about the purpose and
        nature of the object from a business perspective. Clearly, good descriptive
        information is not required for the OpenEAI foundation components to
        effectively implement the functionality of the constraints and rules
        contained in the EO document. However, it is good practice to include some
        helpful, official, and comprehensive information here to help analysts and
        developers. The OpenEAI methodology recommends including links to EO
        documents in the online documentation of messaging applications and gateways,
        so providing useful information here is part of the practice of OpenEAI and
        can be generally helpful. Additionally, in the future, the OpenEAI Project
        may provide applications that generate enterprise documentation from EO
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

documents just as it may provide applications to generate enterprise documentation templates from deployment descriptors. So it is generally a good idea to get into the habit of providing this documentation.

ClassName

The ClassName element contains the fully-qualified class name of the object being described by this object definition. For example, class names of message objects (these come from the example implementations) might be:

com.any_erp_vendor.moa.jmsobjects.Person.BasicPerson.v1_0.BasicPerson
com.any_erp_vendor.moa.jmsobjects.Person.EmergencyContact.v1_0.EmergencyContact
org.any_openeai_enterprise.moa.jmsobjects.coreapplication.v1_0.InstitutionalIdentity

Field

The Field element is the structure used to specify the rules for each field in an object. See the comments with the Field element in this definition for more details.-->

```
<!ELEMENT ObjectDefinition (Description, ClassName, Field+)>
<!ATTLIST ObjectDefinition
    name CDATA #REQUIRED
>
<!ELEMENT ClassName (#PCDATA)>
<!--
```

The Field element is the structure used to specify the rules for each field in an object. There will be one Field element for each field of the object. The combination of all these fields within an object definition provides the XML structure of the object; therefore, fields that are of type 'Element' must be listed in the same order as specified in the XML constraint (DTD) for the object. However, the sequencing of fields of type 'Attribute' isn't important as long as they are listed prior to fields of type 'Element'. The OpenEAI MoaGenerationApplication generates these object definitions for you when it generates the EO document skeleton into which you can add your own desired valid values, transformations, and scrubber specifications, so you don't have to worry too much about the details of how to prepare properly structured object definitions. The following are descriptions of the child elements and attributes of the Field element:

name

The name attribute contains the name of the field as it is specified in the DTD that defines this object. All Field names should match the name of the Element or Attribute as specified in the Segments file. However, Attribute names should match the same structure as Element names. That is, they should be mixed case with the first letter of the name being upper case. This is slightly different than the way attributes are typically named in the Segments file. However, since this file is automatically generated, you shouldn't have to worry about this too much.

type

The type attribute specifies the type of the field from an XML perspective. This information is used by the generic XML layout manager to determine what type of entity it should expect as input or what type of entity to build as output when the object is built from an input or serialized to output. The valid types are 'Attribute', 'Element', and 'Object'.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

isKey

The isKey attribute indicates whether or not this field is considered a part of this object's unique key. Many objects have one key field or several fields which together comprise a composite key. The value of a key field or values of key fields (in the case of objects with composite keys) uniquely identify an object instance. Key values are typically used to determine which instance of a repeatable field has changed when an application or gateway consumes a message with an action of Update. The OpenEAI MoaGenerationApplication attempts to determine these keys based on field requiredness as specified in the Segments file. However, you may need to change this information as application-specific requirements are determined.

Description

The description element should contain some information about the purpose and nature of the field from a business perspective. Clearly, good descriptive information is not required for the OpenEAI foundation components to effectively implement the functionality of the constraints and rules contained in the EO document. However, it is good practice to include some helpful, official, and comprehensive information here to help analysts and developers. The OpenEAI methodology recommends including links to EO documents in the online documentation of messaging applications and gateways, so providing useful information here is part of the practice of OpenEAI and can be generally helpful. Additionally, in the future, the OpenEAI Project may provide applications that generate enterprise documentation from EO documents just as it may provide applications to generate enterprise documentation templates from deployment descriptors. So it is generally a good idea to get into the habit of providing this documentation.

The OpenEAI MoaGeneration Application provides a starting point for this description.

Format

The Format element is used to specify field-specific business rules. The information contained within the EO document is implemented by the org.openeai.config.EnterpriseFormatter Java object. These business rules include translations that should occur when data is placed into the field via the setter methods that exist in the Java message object, the allowable length of data that can be put into the field, the datatype of that data, and the requiredness of the field.

Layouts

Object Definition

The Field element can contain another ObjectDefinition element. Generally, this element will never be used. However, the purpose of this element is to override the definition for an object that may be included through the object definition import mechanism implemented by the include list. This ability to override an object definition would only be used in very specific situations such as if you want to specify additional scrubbers or layout managers for an object that are not included in the EO document for that object that you are including in the include list.-->

```
<!ELEMENT Field (Description, Format, Layouts?, ObjectDefinition?)>
<!ATTLIST Field
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
    name CDATA #REQUIRED
    type (Attribute | Element | Object) #REQUIRED
    isKey (false | true) #REQUIRED
>
<!--
        The Format element is used to specify field-specific business rules. The
        information contained within the EO document is implemented by the
        org.openeai.config.EnterpriseFormatter Java object. These business rules
        include translations that should occur when data is placed into the field via
        the setter methods that exist in the Java message object, the allowable
        length of data that can be put into the field, the datatype of that data, and
        the requiredness of the field.-->
<!ELEMENT Format (Translation?, Length?, Mask?, Scrubber*)>
<!ATTLIST Format
    datatype (char | numeric | alphanumeric | any) #IMPLIED
    required (true | false) #REQUIRED
>
<!--
        The Translation element is a container for configurations for various
        implementations of translations. Presently, the only type of
        configurable translation implemented here is a crosswalk or mapping.
        See the comments with the Mapping element in this definition for
        more details on mappings.

        Other types of configurable translations (such as crosswalks that use
        database tables or web services) may be configured here in the future.
        Presently, such externalizable translations are implemented using
        scrubbers.-->
<!ELEMENT Translation (Mapping+)>
<!--
        The Mapping element provides the ability to map application-specific
        values to enterprise values and vice-versa. These translations occur when
        the setter and getter methods for this field are called. This is typically
        useful when an application has its own version of values for a particular field,
        but needs that data translated to an enterprise value when it is made
        available to the enterprise in a message. Additionally, this translation
        facility is useful when an application that has its own specific values for a
        field wants to consume and process a message with enterprise values. Many
        mappings can be listed here and are distinguished by application name. This
        applicationName must match the "id" of the application/gateway that is using
        this enterprise message object. The following is a description of the child
        elements and attributes of the Mapping element:

        EnterpriseValue

        The OpenEAI methodology recommends that you select and maintain a set of
        enterprise values or enterprise formatting rules for each field of every
        message object that you define. Valid values are not really appropriate for
        some fields, formatting rules or masks are more appropriate. Keeping with the
        XML precepts of transparency and clarity, these enterprise values and formats
        should be as obvious in their meaning as possible. Alternately, you might
        choose to take the values native to your most important enterprise
        applications, such as your ERP system, and dub them as your enterprise
        values. These will be the values that should appear in messages as they move
        about your enterprise, as these messages are serialized to enterprise
        messaging logs, and as these messages are used to present clear and usable
        information to end users in all of your new web applications without decoding
        or translating values. In other words, your enterprise values should be the
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

clear and understandable values that human beings actually understand and use. These are values that have business meaning.

The EnterpriseValue element contains one valid enterprise value for the field to which this mapping belongs.

ApplicationValue

The ApplicationValue element or elements of a mapping contain application-specific values that map to the enterprise value for this field contained in the EnterpriseValue element. Application values are application-specific. The application to which the values belong are indicated by the value of the applicationName attribute of the ApplicationValue. Many applications, especially legacy applications, can have more than one value that translates to the same enterprise value. For example, a legacy payroll system may have two values to represent the gender concept of 'female'. These values may be '1' and 'F', because coding schemes can change over time and many organizations never clean this data up properly. Keeping with precepts of clarity and transparency, you may have declared the valid enterprise values for gender to be 'Female', 'Male', and 'Unknown'. So, in this case, there will be two application-specific values belonging to the legacy payroll system that map to the one enterprise value of 'Female'. This example can also help illustrate the usage of the 'preferred' attribute of the ApplicationValue element. In cases like this, where multiple application-specific values map to the same enterprise value, it is necessary to specify which of these multiple application-specific values is preferred when performing a reverse translation from enterprise values to this application's application-specific values. In this payroll example, '1' is and older, less preferred value meaning female and 'F' is a more current and preferred value meaning female. So the preferred attribute for the payroll systems application-specific value of 'F' will be set to 'true'. This indicates that whenever a translation is performed from the enterprise value of 'Female' to an application-specific value for the payroll system, the resulting value will be an 'F' and not a '1', because the 'F' is preferred.

applicationName

The name of the application to which this application specific value belongs.

Note, the application name of 'Default' listed in the example below is a special application that can be used if you know more than one application uses the same application value for a particular field. For example, many systems use 'F' for an application value for the enterprise value 'Female'. Therefore, 'F' is associated to the 'Default' application so any application using this definition can translate an 'F' to 'Female' in addition to any other application-specific translation it may require.

preferred

This attribute is used when converting enterprise values to application-specific values. The OpenEAI API allows you to convert values from application-specific values to enterprise values and vice versa. While there can be multiple application values that translate to the same enterprise value, there can only be one target application value for a particular enterprise value. This attribute is used to make that determination.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

The following is an example Mapping element within its Translation container element from a sample EO document:

```
<Translation>
    <Mapping>
        <EnterpriseValue>Female</EnterpriseValue>
        <ApplicationValue applicationName="Default" preferred="true">
            <Value>F</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="PaymasterBatchApplication" preferred="true">
            <Value>2</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="FileTrackerGateway" preferred="true">
            <Value>2</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="Isis3Gateway" preferred="true">
            <Value>F</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="PreCollegeGateway" preferred="true">
            <Value>2</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="TransferMasterFileGateway" preferred="true">
            <Value>2</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="UIDirectGateway" preferred="true">
            <Value>2</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="VsamGateway" preferred="true">
            <Value>2</Value>
        </ApplicationValue>
    </Mapping>
    <Mapping>
        <EnterpriseValue>Male</EnterpriseValue>
        <ApplicationValue applicationName="Default" preferred="true">
            <Value>M</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="PaymasterBatchApplication" preferred="true">
            <Value>1</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="FileTrackerGateway" preferred="true">
            <Value>1</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="Isis3Gateway" preferred="true">
            <Value>M</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="PreCollegeGateway" preferred="true">
            <Value>1</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="TransferMasterFileGateway" preferred="true">
            <Value>1</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="UIDirectGateway" preferred="true">
            <Value>1</Value>
        </ApplicationValue>
        <ApplicationValue applicationName="VsamGateway" preferred="true">
            <Value>1</Value>
        </ApplicationValue>
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                </Mapping>
                <Mapping>
                    <EnterpriseValue>Not Available</EnterpriseValue>
                    <ApplicationValue applicationName="PaymasterBatchApplication" preferred="true">
                        <Value/>
                    </ApplicationValue>
                </Mapping>
                <Mapping>
                    <EnterpriseValue>Not Given</EnterpriseValue>
                    <ApplicationValue applicationName="FileTrackerGateway" preferred="true">
                        <Value/>
                    </ApplicationValue>
                    <ApplicationValue applicationName="Isis3Gateway" preferred="true">
                        <Value/>
                    </ApplicationValue>
                    <ApplicationValue applicationName="PreCollegeGateway" preferred="true">
                        <Value/>
                    </ApplicationValue>
                    <ApplicationValue applicationName="TransferMasterFileGateway" preferred="true">
                        <Value/>
                    </ApplicationValue>
                    <ApplicationValue applicationName="UIDirectGateway" preferred="true">
                        <Value/>
                    </ApplicationValue>
                    <ApplicationValue applicationName="VsamGateway" preferred="true">
                        <Value/>
                    </ApplicationValue>
                </Mapping>
            </Translation>-->
<!ELEMENT Mapping (EnterpriseValue, ApplicationValue+)>
<!ELEMENT EnterpriseValue (#PCDATA)>
<!ELEMENT ApplicationValue (Value)>
<!ELEMENT Value (#PCDATA)>
<!ATTLIST ApplicationValue
    applicationName CDATA #REQUIRED
    preferred (true | false) #REQUIRED
>
<!--
        The Mask element has been added for future functionality.
        In the future, users will be able to specify a mask that should
        be applied to data being stored in a field.  This way,
        developers won't have to code this logic themselves, it
        would simply be applied automatically.  The Mask element
        is not currently used by the OpenEAI APIs.-->
<!ELEMENT Mask (Value)>
<!--
        The Length element provides the ability to specify a precise length for
        a particular field. The length of data being passed to a setter method is
        verified against this value if specified. The type attribute of the Length
        element provides the ability to specify either that a field's length be
        exactly a certain length or may be up to a maximum length. Examples of data
        that might use the 'exact' length type are SocialSecurityNumber fields or
        anything else that expects that data going into this field to be an exact
        length.

        The following is an example Length element from a sample EO document:

        <Format datatype="numeric" required="true">
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

```
                <Length type="maximum" value="2"/>
            </Format>-->
<!ELEMENT Length EMPTY>
<!ATTLIST Length
    value CDATA #REQUIRED
    type (exact | maximum) #REQUIRED
>
<!--
        The Scrubber element provides the ability to specify that an
        EnterpriseScrubber implementation be used to process the data being passed to
        a field's setter method. This is typically used for things like case
        conversion, special character removal, etc. However, this component can allow
        any number of special rules to be run on data being placed into an enterprise
        message object. The sequence attribute allows multiple scrubber
        implementations to be specified. They will be executed in the order specified
        by the sequence number (1, 2, 3, etc.).

        The following is an example Scrubber element from a sample EO document:

        <Scrubber sequence="1">
            <ClassName>org.openeai.scrubbers.CaseConverter</ClassName>
        </Scrubber>
        <Scrubber sequence="2">
            <ClassName>org.any_openeai_enterprise.scrubbers.NameScrubber</ClassName>
        </Scrubber>-->
<!ELEMENT Scrubber (ClassName)>
<!ATTLIST Scrubber
    sequence CDATA #REQUIRED
>
<!--
        EnterpriseLayoutManagers are one of the more powerful features of the
        OpenEAI foundation API. The EnterpriseLayoutManager interface specifies the
        methods that must be implemented by an implementation. Then, in the
        deployment descriptor for the application or gateway being configured, the
        Java message objects are told which layout manager implementation to use.
        This includes the name of the layout manager as specified in this EO document
        as well as the Java implementation that uses the information found in this
        document.

        These layout manager implementations are the components that allow developers
        to build objects from a variety of inputs and serialize them to a variety of
        outputs. Currently, there are three types of layouts specified here (in
        addition to XML which is a by-product of these EO documents). These are extract
        layouts, stored procedure call layouts, and result set layouts. Additional
        layouts may be specified in this document. Alternatively, references to other
        XML documents (or any other type of specification document) could be included
        here that point the layout manager implementation to the appropriate layout
        specification. This is one of the concepts that will continue to evolve over
        the life of the OpenEAI project.-->
<!ELEMENT Layouts (Layout*)>
<!ELEMENT Layout (Extract?, SpCall*, ResultSet*)>
<!ATTLIST Layout
    name CDATA #REQUIRED
    type (extract | xml | spcalls | resultset | other) #REQUIRED
>
<!--
    The Extract layout allows an organization to specify the format of a file from
    which an object should be built.  This basically involves specifying the position
```

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

in the file from which to populate the current field.

OpenEAI provides an Extract layout manager implementation in the org.openeai.implementations.layouts.ExtractLayout class.

The following is an example extract layout from a sample EO document. This layout is specified on the Address@type field. The application using this sample EO document will read a line from an extract and build enterprise Java objects from the contents of that line. This information provides the ExtractLayout class with what it needs to go to the appropriate location in that file and pull the information out before applying it to the type field on the Address object. In this case, there will be four addresses in the extract file that need to be built so the ExtractLayout class will be used to build four Address objects:

```
<Layouts>
    <Layout name="InputFeed" type="extract">
        <Extract>
            <Position>
                <Start>210</Start>
                <End>211</End>
            </Position>
            <Position>
                <Start>368</Start>
                <End>369</End>
            </Position>
            <Position>
                <Start>526</Start>
                <End>527</End>
            </Position>
            <Position>
                <Start>684</Start>
                <End>685</End>
            </Position>
        </Extract>
    </Layout>
</Layouts>-->
<!ELEMENT Extract (Position+)>
<!ELEMENT Position (Start, End)>
<!ELEMENT Start (#PCDATA)>
<!ELEMENT End (#PCDATA)>
<!--
```

The SpCall (stored procedure call) layout is an implementation that takes the current contents of an object and builds the appropriate Stored Procedure call(s) for that object. That is, when the gateway storing the object calls 'buildOutputFromObject' a List of stored procedure calls will be returned that the gateway can then execute. This implementation is provided in the org.openeai.implmementations.layouts.SpCallsLayout class.

For example, to create a BasicPerson object in a database, several stored procedures must be called. One to create the 'core' BasicPerson information, one to create each Address associated with the BasicPerson, one to create each Phone associated with the BasicPerson etc.

Note, this implementation has NOT been used in a production environment and is provided for example purposes only at this point.

The following is an example stored procedure call layout from a sample EO document:

```
$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

-->
<!ELEMENT SpCall (PackageName, ProcedureName, ProcedurePosition, ProcedureParameters,
FieldPosition)>
<!ATTLIST SpCall
    action (Create | Delete | DeleteOne | Update | Query) #REQUIRED
    dataType (VARCHAR2 | TIMESTAMP) "VARCHAR2"
>
<!ELEMENT PackageName (#PCDATA)>
<!ELEMENT ProcedureName (#PCDATA)>
<!ELEMENT ProcedurePosition (#PCDATA)>
<!ELEMENT ProcedureParameters (#PCDATA)>
<!ELEMENT FieldPosition (#PCDATA)>
<!--
    The ResultSet layout builds an object from a ResultSet returned from a database.
    It allows you to map columns existing in a ResultSet to a Java message object.
    This implementation is provided in the
    org.openeai.implementations.layouts.ResultSetLayout class.

    Note, this implementation has NOT been used in a production environment and
    is provided for example purposes only at this point.

        The following is an example result set layout from a sample EO
        document:
-->
<!ELEMENT ResultSet (MessageObject, FieldPosition, ColumnName?)>
<!ATTLIST ResultSet
    dataType (VARCHAR2 | TIMESTAMP) "VARCHAR2"
>
<!ELEMENT MessageObject (#PCDATA)>
<!ELEMENT ColumnName (#PCDATA)>
```

End EnterpriseObjects.dtd

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

Appendix 1: The GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$

verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

$Revision: 1.21$
$Date: 3/11/2003 2:01:40 PM $
$Source: /cvs/repositories/openeai/project/documentation/core/ApiIntroduction.doc$