$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

Open EAI Implementation Strategies
Version 1.0

January, 2003

by

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

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

Contents

Introduction

Enterprise Application Integration (EAI) using the OpenEAI Message Protocol and its supporting APIs can be achieved in a variety of ways. This document describes several different implementation strategies that have been used in real-world integration scenarios.

Here, we present descriptions of eleven different scenarios and the details regarding how each scenario has been implemented utilizing the OpenEAI Message Protocol and supporting APIs.

We describe only strategies which have been implemented using this foundation. This is not abstract theory. However, it is also not a definitive description of everything that might be accomplished using OpenEAI concepts, techniques, and supporting foundation components. New techniques and strategies are uncovered all the time; typically it's just a matter of understanding the concepts core to the OpenEAI Protocol and having requirements that drive the need for new techniques. This document's purpose is to provide a high-level overview of some of these strategies to help get thinking about how OpenEAI has been used to perform common integration tasks and how you might use OpenEAI.

For more details on EAI concepts, the OpenEAI protocol, the OpenEAI methodology, or the OpenEAI API, refer to the corresponding documents in the OpenEAI Core Documentation Suite.

Scenarios

An important thing to consider when determining how two applications will be integrated is how invasive the strategy can be. Depending on resources available to perform the work, the lifecycle of the applications being integrated and the complexity involved, it may not be appropriate in all cases to actually change a system in order to integrate it into an enterprise. However, that doesn't mean it can't be integrated. Here are some key items to consider:

- Is the application being integrated developed on a platform (or in a language) that is a strategic direction of your organization? If it is, it may be worth the time and effort

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

to figure out how to make that application message-aware, and use the same technique for all other applications developed with that technology. If it isn't, you might not want to invest the time to make it message-aware. Instead, you might look at some less-invasive ways to integrate it into the enterprise.

- What types of resources (staff) are available to perform the work?
- Does the application have the ability to use Java? This could be natively or through third-party tools that make the application "Java-aware". If it has the ability to use Java, then you should be in a good position to make the application message-aware using the OpenEAI foundation APIs.
- What is the cost associated with making an application "message-aware" vs. using another less-invasive OpenEAI technique to integrate it into the enterprise? Depending on the answers to the questions above, this cost may be too much to support actually changing the application to make it message-aware.

All of these questions should be answered when determining how an application will be integrated into an enterprise. Using the OpenEAI methodology and supporting APIs, they can all be addressed. Based on the answers to the questions, the most useful integration strategy will become increasingly obvious. This document covers several strategies used to answer to these questions.

These are the high-level integration scenarios we will discuss:

Java Application

If an existing application that needs to be integrated into an enterprise or an application being developed is a Java application, there is very little cost associated to making the application message-aware. Utilizing the OpenEAI APIs and an organization's message object API (MOA), making a Java application message-aware is very simple. Refer to the OpenEAI API Introduction Document for more information regarding the Java application development process.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

Java-Aware Application

Any application that can instantiate Java objects can use the OpenEAI APIs and an organization's MOA. This has been successfully demonstrated by message-enabling ColdFusion, PERL, and Oracle applications. Since these applications can instantiate Java objects, they really become very similar to any normal Java application. Of course, the syntax specific to one of these applications may not look exactly like a Java application. However, the API that is exposed to these applications is the same as the API exposed to typical Java applications. That is, they use the same JAR files that the Java application does. Therefore, configuration of these applications is identical to configuration for a Java application.

The benefit of this approach where it is possible is that even those unfamiliar with these applications will be able to understand what they are doing. This not only includes how the applications are configured, but how they are developed. When a person that has never looked at a ColdFusion application, but knows how to use the OpenEAI API in a Java application, looks at the ColdFusion application code, they are immediately able to discern what the application is doing because its code looks very similar to that of the Java applications they're used to developing. They can see references to the same objects and method calls they use in the Java applications they develop. Additionally, they are able to look at the deployment descriptor for the ColdFusion application and determine exactly what types of OpenEAI foundation it uses and what enterprise message objects (MOA objects) are used by the application. Based on these two pieces of information alone, they are able to understand what the application does very quickly.

This concept is one of the core concepts related to application architecture. The goal is to develop foundation that is ubiquitous and can be used by as many people as possible. By doing this, you're guiding your organization to a place where everyone has the knowledge and tools they need to do their jobs. Then, they can make choices on how to actually get the job done. They, however, are not choosing their own, possibly unique, application integration strategy, they are simply given the tools that can be used in whatever development environment they are most familiar with. This is a very important enterprise architecture goal in large organizations that have heterogeneous environments.

Gating Strategies

As discussed in the OpenEAI API Introduction Document, MessageGateways are Java applications that are intended to run for extended periods of time as a daemon process. These gateways are configured with an OpenEAI deployment descriptor that includes PubSubConsumer and/or PointToPointConsumer foundation objects. These consumer objects are Java components that serve as JMS MessageListeners. Additionally, these consumers employ the Command Pattern to execute appropriate business logic associated with a message delivered to the consumer (SyncCommands and RequestCommands). Finally, they provide many features intended to make the gateways robust and reliable. For more information on the OpenEAI PubSub and/or PointToPoint consumer foundation objects, or any other OpenEAI foundation component, you may wish to review the official API documentation (Javadoc). These gateways are intended to either provide functionality and/or services on behalf of an authoritative

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

application, or they are intended to keep a non-authoritative system up-to-date with activities performed in an authoritative system.

The two types of gating strategies using the OpenEAI foundation APIs and methodology that will be discussed here are: gating an authoritative application, and gating a non-authoritative application.

One of the root concepts surrounding the OpenEAI Methodology is that of an "authoritative source". Systems that are not authoritative will request that data be changed in the authoritative application and will keep themselves up-to-date when data changes in that authoritative application.

If a system is considered the authoritative source for a message object or qualified message object, there are a few rules that it must follow according to the OpenEAI Message Protocol. The authoritative system must provide an interface that supports all actions of request/reply messaging for a particular enterprise message object for which it is authoritative. According to the OpenEAI Message Protocol, these actions are query, create, update, delete, and, if appropriate for the object, generate. Non-authoritative applications can then send request messages with these actions to request that these actions be performed on that message object in the authoritative application. Additionally, when an application requests that an action be performed on an enterprise message object in an authoritative application, it is the authoritative application's responsibility to publish a synchronization message to the rest of the enterprise, informing all interested parties of that action. Finally, that authoritative application must publish synchronization messages for any data changed within the system itself through its own native user interfaces that involves any enterprise message object for which it is authoritative.

For example, if a system is considered authoritative for all com.any-erp-vendor.Person.BasicPerson message objects within a particular organization, it must expose an interface (i.e., implement a gateway) that handles the following requests appropriately according to the OpenEAI Message Protocol...

com.any-erp-vendor.Person.BasicPerson.Query-Request
com.any-erp-vendor.Person.BasicPerson.Create-Request
com.any-erp-vendor.Person.BasicPerson.Update-Request
com.any-erp-vendor.Person.BasicPerson.Delete-Request

...to allow non-authoritative applications to request a BasicPerson message object be created in the authoritative system and to maintain it and delete it in the authoritative application. Additionally, the authoritative application must publish the following sync messages to keep non-authoritative applications that store data about these BasicPerson objects in their own data stores up to date:

com.any-erp-vendor.Person.BasicPerson.Create-Sync
com.any-erp-vendor.Person.BasicPerson.Update-Sync
com.any-erp-vendor.Person.BasicPerson.Delete-Sync

For general a general discussion of the message support that must be implemented for authoritative and non-authoritative applications, see the OpenEAI Message Protocol Document.

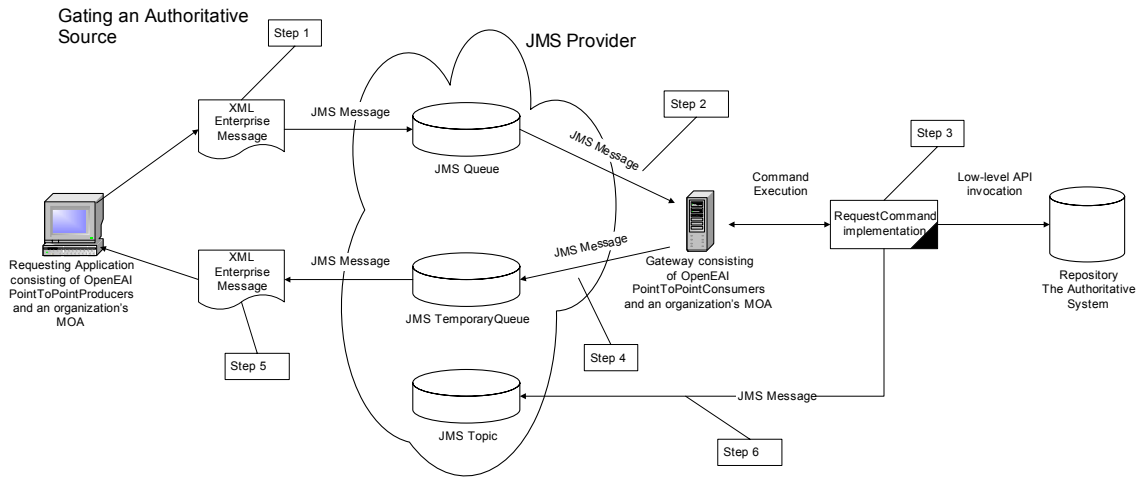
$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

To gate an authoritative system, typically a gateway is deployed that consists of at least one PointToPointConsumer which consumes messages from a JMS queue and executes specific RequestCommand implementations based on the type of message it consumes. There are many choices that can be made regarding how to deploy gateways and develop the commands that make up a gateway: this discussion will assume that this one gateway is responsible for all request/reply message support for this authoritative system. See the OpenEAI Deployment Patterns Document for more information regarding this and other deployment considerations.

When a gateway for an authoritative system consumes a request to perform an action on an enterprise message object, it will execute the appropriate RequestCommand implementation, which will typically use whatever lower-level mechanism it has available to it to perform that action against whatever repository it is gating, and then return the appropriate response to the requesting application. This may mean calling stored procedures in a database, invoking inline SQL statements against a database, performing simple file system I/O operations, binding to a directory server and updating or retrieving information from it, or any number of other forms of lower-level business functions necessary to perform the action. The key point is: it doesn't matter to the requesting application. All that application has to do is send a message to the authoritative source requesting that some action be performed. The gateway is the only thing responsible for knowing what that lower-level business logic really looks like or for having access to call the lower-level business logic. Since most organizations are heterogeneous, this allows many flexible integrations to occur among a variety of systems using the same Message Protocol and invoking many different forms of business logic (without even knowing it).

This business logic is contained within the commands executed by the PointToPointConsumer(s) that make up the gateway. These commands are executed according to the message consumed by the consumer, and it is in these commands that these lower-level mechanisms are invoked. OpenEAI provides this framework for gating these authoritative systems by way of PointToPointConsumers, RequestCommands and many other foundation components. Refer to the OpenEAI API Introduction Document for more details regarding these components.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$



Step 1 - Requesting application requests that an action be performed on an Enterprise Message Object in an authoritative system through the use of OpenEAI APIs.
Step 2 - Gateway for authoritative source consumes request
Step 3 - Gateway executes RequestCommand implementation which invokes lower-level business APIs to perform the action.
Step 4 - Gateway returns a response to the requesting application.
Step 5 - Requesting application consumes response (via action method called on Enterprise Message Object through use of OpenEAI APIs)
Step 6 - If the action was create, update, delete or generate, the authoritative system publishes a synchronization message informing the rest of the enterprise that the action occurred.

Figure 1 – Gating an Authoritative Source (note, this figure does not depict the Router or Request Proxy components, which are discussed later).

Some non-authoritative applications need to persist data for which they are not authoritative in their own data stores. For example, a directory service that is used to present your organization's phone book information on your intranet and some key contact information on your web site may need to persist some person and job data, so that it can be retrieved from the directory to display name, department affiliation, and contact information. However, the authoritative source for this information may be your organization's ERP system, where HR staff and the people themselves (through self-service applications) maintain this information.

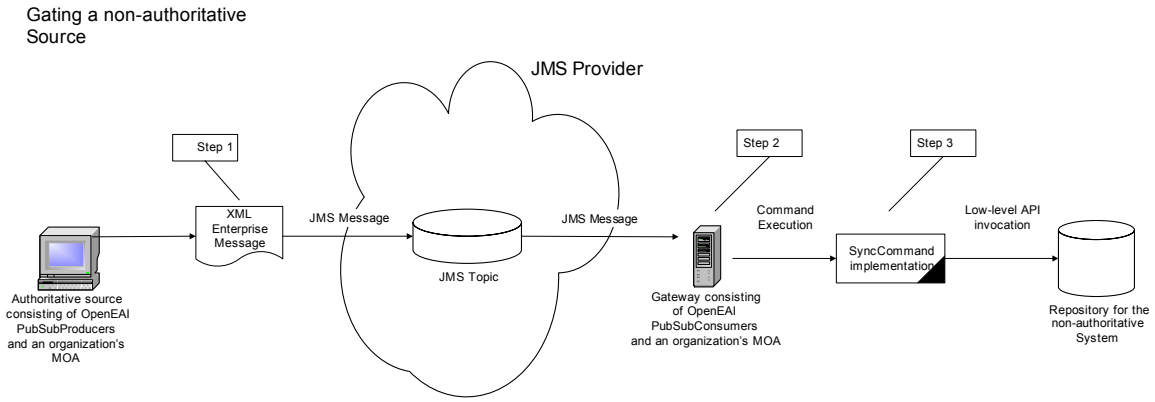
Like a gateway for an authoritative source, a gateway for a non-authoritative application that must actually persist data consumes messages. However, the messages consumed by a gateway for non-authoritative systems are called synchronization messages, not request messages. When actions are performed at an authoritative source, the authoritative application must publish synchronization messages to a JMS topic indicative of the actions performed. All non-authoritative systems that are interested in these business events must consume these synchronization messages to keep themselves up-to-date with the authoritative system.

To gate a non-authoritative system, one typically deploys a gateway that consists of at least one PubSubConsumer that consumes messages from a JMS Topic. This consumer is responsible for consuming synchronization messages from authoritative applications and invoking the appropriate lower-level APIs for that system to take the appropriate actions indicated in the messages.

The process of implementing this gateway is almost identical to the process of implementing a gateway for an authoritative system. The only differences are the fact that instead of consuming messages from a JMS queue with an OpenEAI PointToPointConsumer, the gateway consumes messages from a JMS topic with an

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

OpenEAI PubSubConsumer. Additionally, the business logic is implemented in SyncCommands as opposed to RequestCommands. Finally, there are several differences in the rules that must be followed when consuming a synchronization message as opposed to when consuming a request message. See the OpenEAI Protocol Document for more information regarding the precise messaging behavior for consuming request messages and synchronization messages.



Step 1 - A business event occurs in an authoritative system causing a synchronization message to be published.
Step 2 - Gateway for non-authoritative system consumes synchronization message
Step 3 - Gateway for non-authoritative system executes the appropriate SyncCommand implementation based on the message consumed which invokes lower-level APIs to update the non-authoritative system to keep it up-to-date with the authoritative source.

Figure 2 – Gating a Non-Authoritative Application (note, this figure does not depict the Router component, which is discussed later in this document).

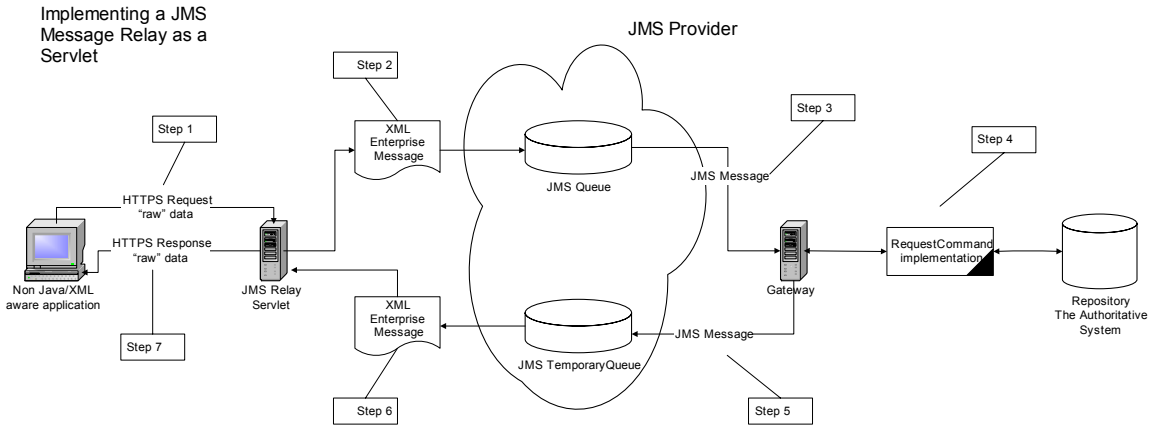
Proprietary Products that are not Java- or XML-Aware

Many products used within an organization might not have the ability to use Java and/or XML natively. Additionally, it might be too costly to make them Java- and/or XML-aware. There are other approaches. One involves developing an API for the proprietary environment and implementing a MessageRelay that allows it to message with the rest of the enterprise.

This technique has been used successfully at the University of Illinois to integrate administrative applications developed in PowerBuilder. PowerBuilder is no longer a strategic development tool for the enterprise. For this reason, it was determined that it would be inappropriate to spend the time and effort to make the PowerBuilder applications Java- and XML-aware. Instead, a PowerBuilder API was developed that allows PowerBuilder developers to work in that environment in a familiar way. A foreign message format was developed (a name/value delimited format which is not the native OpenEAI message format) that would be sent via HTTPS (which is not the native OpenEAI transport) to a MessageRelay. This MessageRelay (a servlet) takes the foreign message format and builds an enterprise message object (a Java business object, for instance, BasicPerson) from the data provided in the foreign message format. It then uses that enterprise message object to perform the action specified in the data passed in (query, create, delete, update) via the OpenEAI foundation APIs, XML, and the Java™ Message Service (JMS). Finally, it receives a response from the target application to

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

which it sent the request, converts the reply back into the foreign format and returns it to the PowerBuilder application as a reply to the original HTTPS request from the PowerBuilder application. PowerBuilder doesn't have to know anything about Java, JMS, or XML. With the PowerBuilder API and the MessageRelay, we are able to successfully integrate any PowerBuilder application into the enterprise. See figure 3.



Step 1 - Non Java/XML aware application sends a foreign message format to the JMS Relay Servlet via its own proprietary mechnisms (HTTPS in this case).
Step 2 - JMS Relay Servlet takes foreign message format and builds an Enterprise Message Object using the OpenEAI APIs and an organization's MOA.
Step 3 - JMS Relay Servlet produces the request on behalf of the application to the authoritative source.
Step 4 - Gateway for authoritative source consumes the request and executes the appropriate RequestCommand implementation based on the message consumed.
Step 5 - Gateway for authoritative source returns a response to the JMS Relay Servlet.
Step 6 - JMS Relay Servlet consumes response from authoritative source and converts it back into the foreign message format appropriate for the requesting application.
Step 7 - Non Java/XML aware application receives the HTTP Response from the JMS relay and populates its user interfaces with that data using its native APIs.

Figure 3 – Implementing a JMS Message Relay as a Servlet.

If PowerBuilder were a strategic direction for the University of Illinois, the OpenEAI API would have been "wrapped" in such a way that the API could have been used directly from the PowerBuilder applications. This would have involved turning the Java objects into COM/ActiveX components for use by the PowerBuilder applications.

OpenEAI provides a MessageRelay reference implementation called the JmsRelayServlet. To learn more about the JmsRelayServlet see the Javadoc.

Routing strategies

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.

When data changes at an authoritative source, the enterprise must be notified of that change according to the OpenEAI Message Protocol. In technical terms, this means a synchronization message must be published by the authoritative source indicating what enterprise message object has changed and what those changes were. Then any non-authoritative system that wants to be informed of those changes can consume that sync message and keep themselves up-to-date with the authoritative source. With OpenEAI

this means that sync messages will be published to a JMS topic and everyone that subscribes to that topic will get the message.

While JMS does provide mechanisms to filter message delivery, it is up to the client application to set that filter. When data is being published from an authoritative source to the entire enterprise, some of that data may not be relevant, or even worse, may not be appropriate for one of those target systems. Therefore, OpenEAI recommends that these situations be handled centrally via special infrastructure gateways called Routers. This component consumes all sync messages published by all authoritative sources and routes them to the appropriate target application based on what target systems are interested in, the content of the message, or several other factors. Those factors are completely determined by the implementation of the router. This way, an organization doesn't have to give all gateways access to all topics to which authoritative sources publish messages. Those targets are simply given access to their own delivery topic and the router delivers messages to them after it's done its work. The centrally-managed router is the only component that has subscription access to the topics to which authoritative applications publish messages.

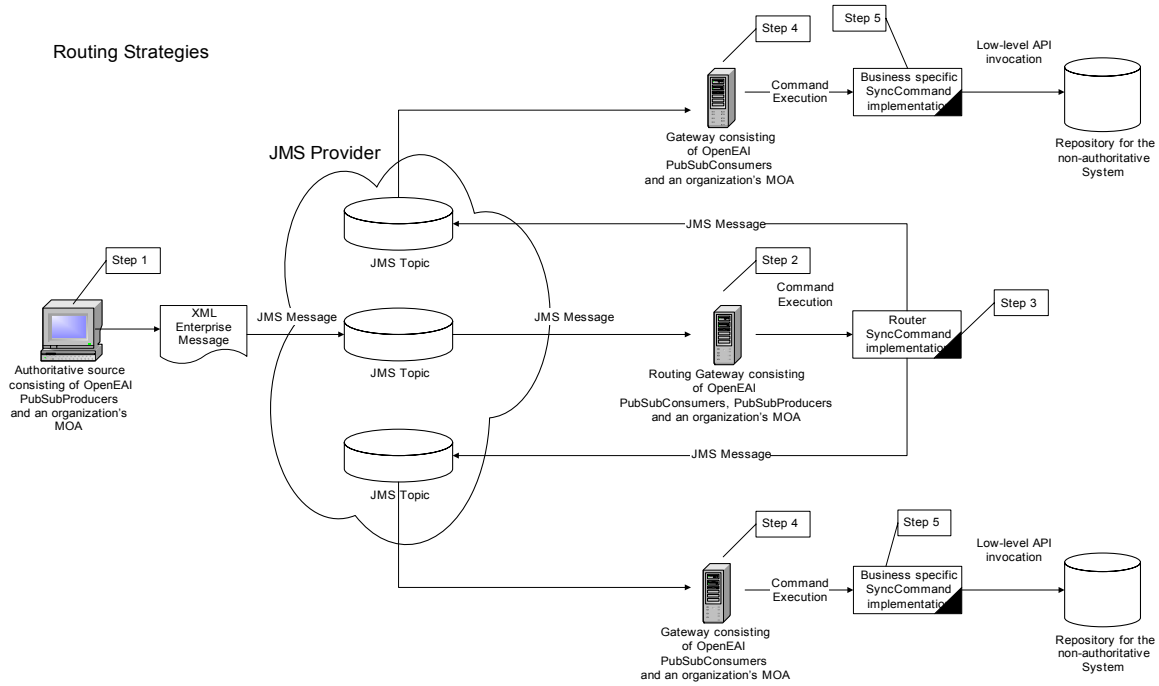
OpenEAI provides a reference implementation gateway called the EnterpriseTransRouter that performs this function based on the type of message that was published, who's interested in that message, and what content within a message a non-authoritative application is allowed to see. Additionally, it provides routing based on the content of the message. This means that a message will be routed to an endpoint if the message published is of a certain message category and message type and if it contains content that makes it appropriate for that endpoint. See the OpenEAI Message Protocol Document for a detailed description of message category, message type, and other important elements of message structure.

There are several layers to this messaging infrastructure gateway. The first layer relates to the type of message published. This includes the message category, message object, and message action that was performed at the authoritative source. For example, a non-authoritative application may only be interested in knowing when a new person is created in an enterprise and not when their data is updated. In this case, it would be interested in com.any-erp-vendor.Person.BasicPerson.Create-Sync messages but not com.any-erp-vendor.Person.BasicPerson.Update-Sync messages. Another target may be interested in all com.any-erp-vendor.Person.BasicPerson sync messages. This is all handled by the main EnterpriseTransRouter SyncCommand implementation. These routing "rules" are specified entirely within the configuration of the router. If a new target needs to be routed to, there are no programming changes necessary in the router. That target is simply added to the appropriate section of the router's configuration in the deployment descriptor of the router.

The next layer provides very specific content-based routing. The configuration of the EnterpriseTransRouter allows an organization to specify routines that should be executed prior to routing to a target, based on the content of the message consumed by the router. These routines are simply Java classes that are invoked by the main EnterpriseTransRouter SyncCommand implementation after it determines (based on the category, type and action associated to the Sync message) that the target is interested in the message. The RoutingCriteria classes look at specific message content and only allow the message to be routed if that content is appropriate for the target. For example, the gateway that was interested in all com.any-erp-vendor.Person.BasicPerson.Create-Sync messages is only interested in com.any-erp-vendor.Person.BasicPerson.Create-

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

Syncs if the BasicPerson object in that message is of a certain "type", otherwise, the message shouldn't be routed to the target application. These criteria can also be incorporated into the router's process with no code changes to the main routine itself. It's simply a matter of developing the RoutingCriteria class and referring to it in the deployment descriptor for the router.



Step 1 - A business event occurs in the authoritative system and a synchronization message is published.
Step 2 - The enterprise router consumes the message.
Step 3 - Based on the configuration of the router, the message is routed to all other interested non-authoritative systems.
Step 4 - Non-authoritative systems consume sync message.
Step 5 - Business specific logic is executed by the gateway gating the non-authoritative systems to keep themselves up-to-date.

Figure 4 – Routing from an Authoritative Source to Non-authoritative systems

It is important to note that this messaging infrastructure gateway is configured, deployed and developed just like any other OpenEAI gateway. It uses all the same OpenEAI foundation components that all other gateways use. The difference is that the command(s) for this gateway are not intended for a specific business purpose or a specific application or department in an organization. They are intended for a much more general purpose. This is why this type of gateway is referred to as an "infrastructure" gateway.

Refer to the OpenEAI API Introduction Document for more information regarding what gateways are and how they're developed. To learn more about the OpenEAI EnterpriseTransRouter reference implementation, see the Javadoc for that reference implementation.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

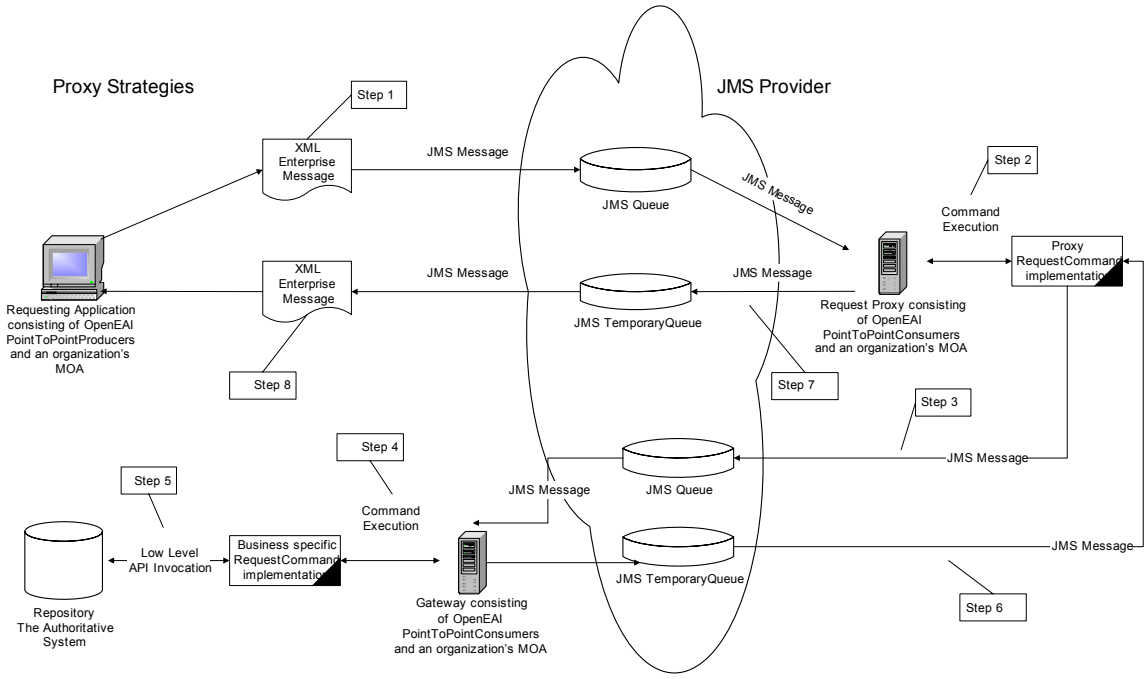
Proxy strategies

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.

Routers control which sync messages get delivered from an authoritative source to target gateways throughout an enterprise. **Request proxies** control which request messages an application may send to an authoritative source. For example, the AnyERP system is authoritative source for com.any-erp-vendor.Person.BasicPerson as it is deployed within the fictitious enterprise, Any OpenEAI Enterprise. The AnyERP system supports all the message actions for com.any-erp-vendor.Person.BasicPerson (create, update, delete and query). However, it would be very impractical and complicated to make the gateway for the AnyERP system know which message actions to allow based on which application is sending the request. That is where messaging middleware should come in…enter the Request Proxy.

A request proxy sits between a requesting application and an authoritative source (or, in reality, between many applications and many authoritative sources—one deployment of this infrastructure gateway should provide proxy services for many applications). The requesting application sends a request to the proxy and the proxy determines, based on the application making the request, the contents of the message, and potentially on other supporting data that's not included in the message, whether the requesting application is allowed to send that request to its intended destination. If it is permissible, the request proxy forwards the request to the destination on behalf of the requesting application and returns the response from the authoritative source (the destination) to the requesting application.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$



Step 1 - Requesting application sends request
Step 2 - Request proxy consumes request
Step 3 - Request proxy forwards request to the authoritative source if the requesting application is authorized.
Step 4 - Authoritative source consumes request and executes appropriate business logic
Step 5 - Lower level business API is invoked by authoritative source gateway
Step 6 - Authoritative source returns response to request proxy
Step 7 - Request proxy returns response to requesting application
Step 8 - Requesting application consumes response from authoritative source by way of the request proxy.

Figure 5 – Proxying requests from un-trusted applications.

OpenEAI provides a reference implementation gateway called the EnterpriseRequestProxy that serves this purpose. Like the EnterpriseTransRouter mentioned above, there are several layers to this component. Also, like the EnterpriseTransRouter, this gateway is an infrastructure gateway that provides general EAI services implicit in general EAI requirements.

The first layer of the EnterpriseRequestProxy determines if the application sending the request is authorized to send that request based on the sending application's name, the message category, message object, and message action of the request message. These qualifiers are specified in the deployment descriptor of the request proxy. This function is performed by the main RequestCommand implementation of the EnterpriseRequestProxy gateway.

For example, say you want a web application to be able to query for any com.any-erp-vendor.Person.BasicPerson object stored in the AnyERP system, but not be able to create any new people. If the web application sends a com.any-erp-vendor.Person.BasicPerson.Create-Request to the AnyERP system, the request proxy will determine that the application is not authorized to send such a message to the AnyERP system and it will return an appropriate error reply message (stating that the application is not authorized to make such a request) to the requesting application.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

The second layer to the Request Proxy includes a sort of content-based proxying. This means, the request proxy will use specialized proxy classes associated with a requesting application to ensure the data being sent is appropriate.

For example, a web application application can query the AnyERP system for all com.any-erp-vendor.Employee.BasicEmployee objects, but it may only create a com.any-erp-vendor.Employee.BasicEmployee if the content of a BasicEmployee indicates that the employee is in one specific class (such as 'full-time') and if the employee already has BasicPerson data in the AnyERP system. After the request proxy determines that the application can create a BasicEmployee by the first layer of the proxy, it is instructed via its deployment document to execute a special proxy class that will look at the actual content of the message being sent and determine if it's appropriate. Additionally, this class will query the AnyERP system to determine if that particular employee that the web application would like to create already has person data in the AnyERP system. If all this turns out to be copasetic, then the request proxy will actually proxy the request for the web application and return the reply from the AnyERP system to the web application. If the employee is of the wrong class or if the employee does not yet have person information in the AnyERP system, the request proxy will refuse to proxy the request and send the appropriate error reply to the web application stating that it is not authorized to perform the action it is requesting.

It is important to note that both the EnterpriseRequestProxy and the EnterpriseTransRouter use facilities already available to them to make many of these determinations. They use the same type of deployment descriptor that all OpenEAI based gateways used. They both use the standard PointToPointConsumer and PubSubConsumer foundation objects as well. The functionality of the gateways is entirely encapsulated within their RequestCommand and SyncCommand implementation. They simply use the bi-products of the deployment descriptor to make their initial decision and other properties associated to the command implementations that tell them the other actions they may need to perform.

To learn more about the OpenEAI EnterpriseRequestProxy reference implementation see the Javadoc for the EnterpriseRequestProxy.

Mainframe applications

Although it is possible to message Java-enable mainframe applications natively in organizations that keep their mainframe system software current (that is, where you can deploy Java and other current software), this appears to be rather uncommon. In the organizations we have worked in or worked with so far, most are migrating away from mainframes or, those that are continuing to operate them are running somewhat dated configurations that do not allow for the use of Java and other current technologies. The OpenEAI Project would really like to work with an organization that is interested in making mainframe applications message-aware and running Java on their mainframe. We realize that there are many enterprises that use the mainframe as a modern enterprise application server and database server. In this section, however, when we say "mainframe applications" we are generalizing to mean mostly legacy COBOL and IMS applications.

In the integration scenarios OpenEAI Project participants have worked with so far, mainframe applications are integrated through more traditional methods, such as the use

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

of flat files. Generally, a number of flat files are already produced or used by mainframe application batch processes that can be used as feed or extract points. In some cases, the capability and knowledge exists in organizations to write new programs for mainframe applications to process new file formats as input or to extract new file formats as output. However, this capability and knowledge may be *almost* as rare as the willingness of IT management to sanction modifying a legacy mainframe application.

The most common strategies we have employed so far involve developing scheduled applications that iterate through mainframe extract files and turn extract records into enterprise messages, or developing gateways that consume enterprise messages and create flat files that can be processed by the mainframe application. Both of these techniques will be discussed here.

The first scenario involves the following basic steps:

1. Create the flat files using some mainframe job. The layout of these files will vary based on the types of transactions and the data. They will typically be fixed-length record files that follow a specified record layout.

2. Iterate through the records of the file or files with a Java application that is message enabled using the OpenEAI foundation components.

3. Turn the transactions in those files into enterprise messages using a specialized input layout manager. See the OpenEAI API Introduction Document for more details on layout managers and how they are used.

4. Execute the transactions via request or synchronization messages, depending on whether or not the mainframe system is considered the authoritative system.

The mainframe job will create the flat files. The Java application will take the contents of the flat file and turn each transaction into an enterprise message using the specialized input layout manager. Then the application will use the organization's MOA and either PubSub or PointToPoint producers to either publish a sync message or send a request message.

The second scenario involves the following steps:

1. Develop a gateway to consume synchronization messages from authoritative applications. This involves developing SyncCommand implementations that will be executed by the PubSubConsumer whenever a message is delivered. These commands will take the enterprise messages and use a specialized output layout manager that will create flat files that correspond to the format that can be processed by a mainframe job.

2. Develop (or use an existing) mainframe job that will take the flat file created by the gateway and persist these transactions.

When the gateway for this integration consumes a synchronization message from an authoritative system, it will take that message and convert it into a flat file record or records using the specialized output layout manager. This process produces a flat file that can be processed by the mainframe job to reflect these transactions in the mainframe application.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

Polling Strategies

When data changes in an authoritative system, it must publish synchronization messages so that other systems can keep themselves up-to-date as data changes in the authoritative system. These synchronization messages must be published when data changes both through request/reply messaging activity and through any other online or batch activity in the system through its own user or batch interfaces. However, sometimes it is too costly to make an authoritative system natively message-aware. In these cases, OpenEAI recommends the implementation of another type of application that detects changes in an authoritative system and publishes the synchronization messages on behalf of that system without making any changes to the authoritative system itself. These applications are called polling applications.

Polling applications leverage another OpenEAI foundation component called a scheduled application. These applications are configured very similarly to a message gateway or a message-aware application. The OpenEAI Application Foundation APIs (AFA) provide a mechanism that allows specific business logic (ScheduledCommands) to be executed according to flexible schedule. See the OpenEAI API Introduction Document for details on the scheduled application foundation.

A polling application checks for business transactions that have occurred in an authoritative system on a regular interval and publishes these transactions to the rest of the enterprise when they are detected. Typically, they will use low-level APIs to retrieve information from the authoritative system and compare it to the last known state of the enterprise objects that may have been involved in business events. Based on that comparison, the polling application publishes synchronization messages on behalf of the authoritative system indicating the nature of any changes to the rest of the enterprise.

Synchronization Message Logging

When a message-aware application requests that an action be performed at an authoritative source, there is someone or something waiting for a response to that request. This may be a person using a message-aware application (such as a servlet or a client/server application) or a batch application running during the late night hours. Regardless of the type of application making the request, with request/reply messaging, there is an opportunity to handle any errors that may occur in the target application immediately. When a request is produced, a reply must be returned.

When synchronization messages are published, they are delivered to endpoints that process these messages without any person or system waiting on a response saying that everything was processed successfully, or that there were errors. Because of the asynchronous modality of this communication, it is necessary to be able to keep a record of all these synchronization messages. Since a person isn't waiting on a response, there must be a way to review messages published after the fact if a consuming gateway encounters problems when processing the message. These logged messages can be used later when debugging problems or to send the message through again as it was originally published. Of course, this must be performed cautiously since other changes may have come through for the object in question and those changes cannot be lost.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

This type of infrastructure gateway is called a synchronization message logger. The reference implementation for this function provided by the OpenEAI Project is implemented in the EnterpriseLoggingService. Specifically, it is implemented by the EnterpriseSyncLoggerCommand. This EnterpriseLoggingService is another infrastructure gateway, which consumes all messages delivered to an organization's "synchronization logging topic" and persists these messages in their entirety. The repository is a very simple structure that allows the entire message to be stored and correlated to a message id associated to the message when it was published. Generally, one "synchronization logging topic" should be sufficient for most organizations (multiple gateways will subscribe to that one topic of course).

All OpenEAI PubSubProducer objects can be configured to publish messages not only to an intended JMS topic but also to the logging topic. The configuration item that provides this facility is the LoggingProducer associated with a ProducerConfig object in an application's OpenEAI deployment descriptor. This LoggingProducer will be used by the producer being configured to publish the message to the organization's logging topic at the same time it publishes the message to the intended topic from which business applications (or maybe a sync message router) will consume messages. This effectively makes a copy of the message. This way, an organization can keep a history of every synchronization message published. This is important for synchronization messages because typically they are published and processed when no one is looking.

To learn more about the OpenEAI sync message logging reference implementation see the Javadoc for the EnterpriseSyncLoggerCommand.

Synchronization Error Message Logging

When a request is sent to an authoritative system, errors that occur while performing the request are returned immediately by the gateway of the authoritative system. When a non-authoritative system consumes a synchronization message, any errors that occur while processing the message must be published to the "synchronization error topic." Another infrastructure gateway, called the sync error message logger, consumes these error messages and persists them for later analysis.

All SyncCommand implementations must have a PubSubProducer named "SyncErrorPublisher" associated with them. Additionally, they must have a PropertyConfig object named "SyncErrorSyncProperties" which contains a property named "SyncErrorSyncPrimedDocumentUri" which points to a "primed" org.openeai.CoreMessaging.Sync.Error-Sync message document. These objects are used to publish the error message when an error occurs processing the message consumed by the non-authoritative system. These errors can then be correlated to the message that was originally published (and logged). With these two pieces of data, integration administrators and/or support analysts can make decisions about what caused a sync message consumption error and how to rectify it.

The reference implementation for this function provided by the OpenEAI Project is implemented in the EnterpriseLoggingService. Specifically, it is implemented by the EnterpriseSyncErrorLogger command. To learn more about the OpenEAI sync message logging reference implementation see the Javadoc for the EnterpriseSyncErrorLogger command.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

Callout Strategies

When an organization determines that it is acceptable to make changes to an application to make it message-aware, there are several strategies that can be employed. Most business systems are based on some sort of data repository. This section discusses strategies that have been employed to message-enable database applications. Again, it is not intended to be a full treatment of this topic; it is only intended to provide some background on techniques that have been employed using the OpenEAI supporting APIs to date and serve as a starting point for you to start thinking more about how OpenEAI could be incorporated into your own applications. Database management systems (DBMS) have varying potential when it comes to making them message-aware. Examples include the ability to run Java within the database, invoking other callout techniques like RMI, TCP, or even calling servlets.

For example, applications based on an Oracle database have the ability to invoke Oracle's Java Stored Procedures. These are simply Java classes that are exposed as stored procedures to the database runtime system. Then, as the application is used, these stored procedures can be invoked. The stored procedures can then use an AppConfig object just like any other OpenEAI-based application to store and retrieve pre-initialized foundation components and MOA objects to make the application message-aware. When activities are performed within the database, these pre-configured objects can be used to notify the rest of the enterprise of the activities through synchronization message publication.

Another example for Oracle-based systems uses RMI (Remote Method Invocation) to invoke logic in a RMI server process that is an OpenEAI Scheduled Application (daemon application) that marshals objects created in the database into enterprise message objects and notifies the enterprise of the actions performed within the database.

Other databases may have the ability to invoke other external applications, such as servlets, or to perform simple TCP/IP communication. In these cases, the technique is similar to a message relay strategy, because the data associated with the database action must be converted into an enterprise message object and the state of that object must be made known to the rest of the enterprise.

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

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

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

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

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

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:

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

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.

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

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$

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

$Revision: 1.15$
$Date: 3/11/2003 2:05:36 PM$
$Source: /cvs/repositories/openeai/project/documentation/core/ImplementationStrategies.doc$