White paper
December 2004

Enterprise Metadata Discovery.

A joint white paper with IBM and BEA

By Jim Gish, BEA Systems; John Green,
IBM Corporation; Amy Mollin, IBM Corporation;
Piotr Przybylski, IBM Corporation;
and Mitch Upton, BEA Systems

Contents

Introduction

Businesses need automation to manage changes in business processes. Rapid development tools for business processes allow a company to meet time-to-market requirements. Similarly, organizations often rely heavily on Web portals to collect and disseminate information to customers and employees. These portals often need to change quickly to respond to changing customer and employee needs. Rapid development tools for Web portals allow a business to quickly alter its Web portal to meet those needs. Business processes and Web portals often use data stored in proprietary enterprise information systems (EIS), such as customer data, purchase orders and inventory, to make decisions, modify the presentation of data, and to do work to accomplish their goals.

Today, standards defining adapters are immature and cannot adequately address the needs of application developers. A new class of rapid development tool, called an *integration framework*, is available from several large software vendors. An integration framework attempts to unlock enterprise data through the use of modular software components called adapters. These adapters act as the keys the integration framework uses to unlock data in a particular EIS and make it available for use by an integration developer.

This white paper introduces a new metadata discovery and import model for adapters that can dramatically improve the usability of adapters within an integration framework. Using this model, developers would be able to gather information about the data and functions of an EIS (using metadata discovery). Having gathered this information, the developer would use the metadata import facilities to define a new custom interface to the required data and functions in the EIS.

The current state of affairs

The use of an integration framework and adapters is intended to allow an integration developer to focus on business logic and data moving through the integration solution, rather than on the details of the integration interface for any given EIS. Unfortunately, as it stands today, integration frameworks do not completely relieve the integration developer from EIS interface details, because there is no standard that defines how an adapter should plug into the

framework in a way that enables integration developers to use it easily. This can mean either that significant hand coding is involved, or that detailed knowledge of the EIS interface is required. Thus, the data contained in the EIS is really only unlocked for those integration developers who are well versed in the interface details for that EIS.

What is needed?

To relieve the integration developer from EIS interface details, to allow them to define a tightly focused and relevant interface to EIS data and functions, and to enable them to integrate EIS data seamlessly into the integration framework, adapters must allow the dynamic discovery, description, and use of the following EIS attributes:

- *System capabilities – what types of data are managed, and what can be done to that data*
- *Data structures – the form of the data passed between the client and the EIS resource.*
- *Communication protocols – the choreography that defines the flow of communication between a client and the EIS resource.*

Dynamic discovery and description of these attributes requires the adapter to provide descriptive data (called *metadata*) that a general-purpose tool can interpret. The tool then passes this metadata on to the integration developer as information displays and dialogs. The developer uses these displays and dialogs to discover system capabilities, data structures and communication protocols (through the metadata representing them). The developer can then bring the metadata to bear at the appropriate times in the development cycle and appropriate places in the integration solution.

Once the ability to plug adapters into the integration framework is addressed (through the use of metadata) the last remaining problem is that of limited adapter availability. To solve this problem, the discovery of metadata must be codified in strong and widely accepted standards. Then, all integration frameworks can use any compliant adapter, thus increasing the number of adapters available to that framework.

Current standards for adapters

Current standards, such as Java™ 2 Platform, Enterprise Edition (J2EE), have begun to define some of the facilities needed to provide adapter metadata, but still fall short. The J2EE Connector Architecture (JCA), Version 1.5 specification [1] defines basic communication protocols, but does not define how a client can discover or use the system capabilities or the data structures of the EIS resource.

The JCA, Version 1.5 specification defines a set of system contracts and a client API that help to standardize the run-time interaction with an adapter. However, the client is still required to have a significant amount of information that is specific to the particular adapter. The *JCA, Version 1.5 limitations* section provides descriptions of specific limitations related to integration development.

Conclusion

Widely accepted standards for metadata discovery would allow the use of a single adapter across integration frameworks and would provide the integration developer with the information required to easily access EIS data, without having to understand EIS interface details.

JCA, Version 1.5 limitations

JCA, Version 1.5 outbound connection management, security management, transaction management, message inflow contracts and Common Client Interface (CCI) are a good start to normalizing communications with any EIS. The deployment descriptor for a resource adapter depict the different types of request-response outbound connections that can be configured, and the different types of inbound connection to which the adapter can deliver inbound communication. However, neither CCI, nor the message inflow contracts, provides the detail required to fully standardize the description of the resource adapter's behavior.

CCI

CCI normalizes request-response communication to some degree, because all requests are initiated through a call to Interaction.execute(), and this method has clearly defined request-response semantics. (Unless otherwise specified, particular interfaces are part of J2EE CA javax.resource.cci package) However, the metadata provided by the deployment descriptor is basic and does not allow for more descriptive metadata, like:

- *Whether properties of the outbound connection are required or optional*
- *Grouping of properties by function or other common attribute*
- *Enumerated values*

CCI defines only a simple application programming interface (API) for function invocations on the adapter. It is appropriate for outbound operations, but does not fully support EIS-initiated inbound operations. CCI does not define any facility for browsing EIS capabilities or data structures. It does provide a rudimentary request-response object factory, but defines only a few standard request-response object types, which are insufficient to represent a large breadth of data structures.

The key outbound method in the CCI interface is the Interaction.execute() method. This method allows a client to invoke a function on the adapter, and in turn, in the EIS. The actual name of the function to invoke and any other configuration data needed to describe the invocation is contained in an object called an *InteractionSpec*. The InteractionSpec is specific to the adapter, so constructing one requires detailed knowledge of the adapter and custom code. While CCI defines a facility to discover the InteractionSpecs supported by the adapter, the developer coding to the CCI interface must know which InteractionSpec implementation the adapter requires for specific EIS functions.

Further, the Java data structure the developer passes to or receives from the Interaction.execute() call is specific to the adapter or EIS itself because CCI does not define a truly general-purpose data structure. Nor does it specify an interface to have the adapter generate the data structure for the developer. CCI defines an inbound method interface MessageListener.onMessage() which supports a request-response style invocation.

Message inflow contracts

The message inflow contracts normalize EIS notification and EIS-initiated operation communication to some degree because they define the basic protocol needed to enable these communication primitives. However, the message inflow contracts do not clearly specify how to configure the inbound connection for an adapter. The inbound connection is responsible for accepting incoming messages and dispatching them to interested consumers. JCA defines the javax.resource.spi.ResourceAdapter interface to include a set of properties, but does not indicate what these properties should be used for. Specifically, properties can be set on the ResourceAdapter, or on the javax.resource.spi.ActivationSpec specific to an inbound connection type. Nowhere does it specify which of these should contain connection properties designating the EIS instance for inbound connections, or any other property types. According to the JCA specification, getting inbound communication is predicated on the client registering a MessageEndpoint and ActivationSpec instance with the ResourceAdapter. However, it does not clearly define the role of ActivationSpec in getting this inbound communication.

Goals of this white paper

The problems of limited adapter availability, compatibility and inconsistent features are all directly related to the lack of mature and widely adopted standards defining adapters and their ability to expose metadata for discovery from integration tools. The JCA, Version 1.5 specification defines basic communication protocols, but does not define how a client can discover or use the system capabilities or the data structures of the EIS.

The concepts put forward in this white paper are intended to augment, not replace, existing J2EE standards such as the JCA, Version 1.5. It defines how clients discover and use system capabilities, data structures and communication protocols for an EIS. An explicit goal of this whitepaper is to describe how existing adapters (compliant with JCA, Version 1.5 and implementing CCI as their client interface) may be extended in a straight forward way to allow them to provide for metadata discovery and import. This can be done with no changes to the adapter's CCI interface or any of the adapter's implementation of the system programming interfaces (SPIs).

IBM and BEA envision that the only changes required would be to add implementation classes for the components of the metadata discovery and import service described later in this white paper, and to add additional packaging information (e.g., descriptors) needed to register the new implementations.

Using adapters that allow for metadata discovery and import, integration developers would be able to create services with the following (where the operation style is supported by the underlying adapter) capabilities:

- *Integration-framework-initiated operations to retrieve data from or do work within the EIS.*
- *EIS-initiated operations, where the request originates within the EIS and the response is generated by the integration framework. This type of operation is used for retrieving data from, or doing work within, the integration framework.*

Conceptual overview

Figure 1 and the accompanying description provide an overview of the solution model that is being proposed to address the limitations of adapters as previously described.

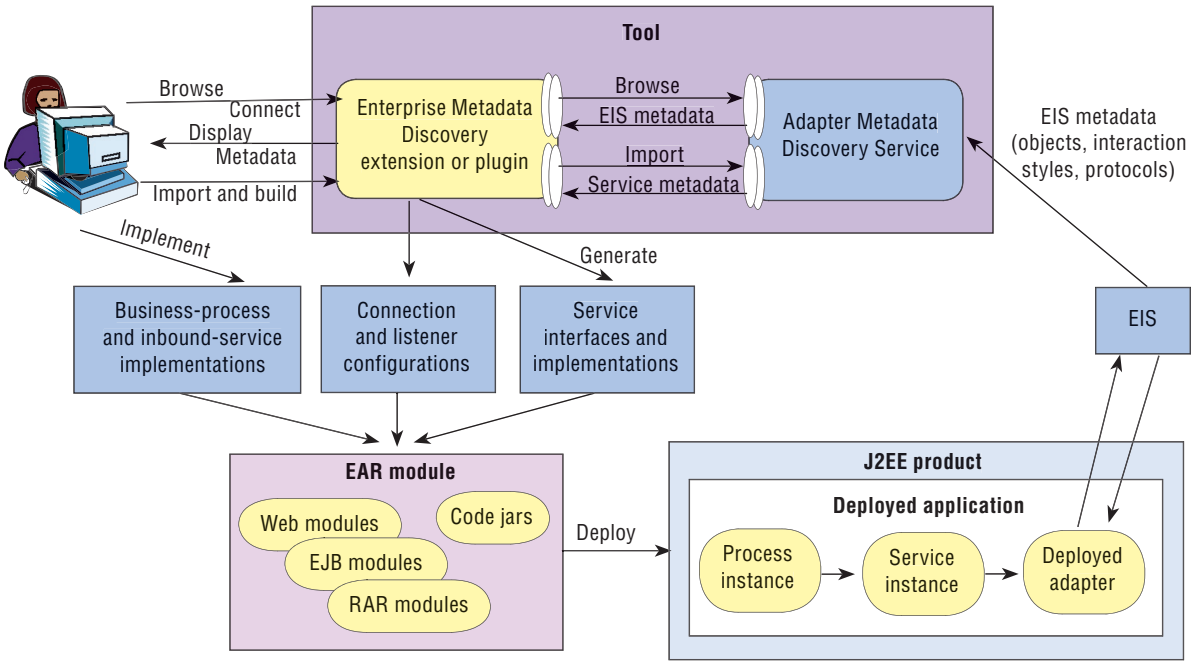

Figure 1. Solution model overview

Figure 1 shows a tool user as she performs the following steps (in order) with the goal of developing an integration solution:

1. Define service description
2. Build service interface and implementation
3. Test the service (possibly from a separate tool)
4. Package and deploy the application (possibly from a separate tool)

Defining an application using services for an EIS

Define service description

The tool user synthesizes EIS metadata into service metadata representing an abstract business service called a *service description*. The service description cannot be directly invoked from Java code, but instead contains all the information needed to generate a Java interface for the service (called the *service interface*) that can be invoked from Java code. The tool user defines the service description by picking objects to interact with, the style of the interaction (event from EIS, request to EIS and so on) and configuration information that controls the interaction with the EIS.

Build service interface and implementation

The tool user directs the tool to generate a service interface and implementation for the service description. The tool generates a Java interface for the service and any classes or J2EE modules that are needed to implement the service. The exact form of the service interface and the generated service implementation are tool-specific. One tool may choose to generate the service interface and implementation as a simple Java interface and class, respectively. Another tool may choose to generate the service interface and implementation as an Enterprise JavaBeans (EJB) remote interface and bean implementation, respectively. Put simply, each tool would generate a service interface and service implementation that best matches the requirements of the tool user, taking into account the tool's approach to developing the overarching integration solution.

Package and deploy the application

The tool user directs the tool to package and deploy the application. This may be done in a managed or nonmanaged environment. In a managed environment the tool could package and then deploy the integration solution as a J2EE enterprise archive (EAR). Testing the assembled application as a whole would be a step to delivering the finished application to the deployer for down-stream environments such as acceptance testing and production. Tool vendors would choose whether to provide integrated packaging and deployment support from within the same tool used to develop the service.

The application archive would contain the generated service interface and implementation and any other classes such as user-written business processes, web services and so on. The resource adapter archive (RAR) modules needed to support inbound or outbound communication may be contained in the application archive or deployed separately. Note that the tooling framework can package the final application archive so it can run on a standard J2EE, Version 1.4 server.

Metadata categories

Adapters that support metadata discovery and import can provide two different categories of metadata. These are *EIS metadata* and *service metadata*.

EIS metadata

EIS metadata describes the system capabilities, data structures and communication configuration supported by the EIS.

System capabilities

At the system level, metadata describes the types of information managed by the EIS system. This can be represented as functions on the EIS client interface or as a set of business objects that are accessed from the EIS client interface. In addition to function and object descriptions, EIS metadata includes a description of the styles of communication that can occur with functions and objects in the EIS metadata model. Communication styles include:

- *Inbound versus outbound – Inbound is EIS-initiated; outbound is client-initiated.*
- *One-way versus request-response – One-way communication takes a request, but does not return a response. Request-response communication takes a request and returns a response.*

From the point of view of the discovery service, this white paper is concerned only with whether a communication style is inbound versus outbound. The distinction between one-way and request-response can be made after import through service metadata.

Interaction style

An interaction style defines whether the interaction will be inbound or outbound, and includes a display name for the interaction.

Data structures

A data structure describes the structure and content of arguments to functions on the EIS client interface or the business objects accessed through the EIS client interface.

Communication configuration

A communication configuration describes properties required for communication with the EIS. At a high level, this describes how data is moved between the adapter and the resource. These properties include those required for both inbound and outbound communication. In some cases, more than one set of properties may be required to describe different types of outbound connections necessary for design time and run time, or outbound connection and inbound connection[1] configurations.

Note that this white paper allows for the discovery service to access metadata from the EIS or other sources. If the metadata is stored in a separate repository, the discovery service can mine the metadata from there. This allows the discovery service to combine all metadata at its disposal to provide a meaningful user experience to the tool user.

Service metadata

Service metadata is generated by the adapter as functions are imported into a service description. This includes service-level metadata such as the service name, and configuration for an outbound connection to the EIS (outbound case) or configuration for an inbound connection to the EIS (inbound case). Service metadata also includes function-level metadata such as the function name, data description for the input and output, and either InteractionSpec properties (outbound case) or EIS function name (inbound case).

Metadata Discovery Service

The metadata discovery and import capabilities of an adapter are exposed through a facility called the *Metadata Discovery Service* (or *Discovery Service*, for short) for the adapter. The Metadata Discovery Service is the centerpiece of an adapter's metadata discovery and import capabilities. It allows the adapter to provide a view of the metadata model of the EIS to the tool. The adapter can present any metadata it wishes through its discovery service. This makes the discovery service the best single place to innovate and provide a powerful user experience to users of the adapter.

Each of the categories of metadata described in the previous sections is made available to the tool through the adapter's metadata discovery service. The Metadata Discovery Service is based on a discovery SPI implemented by the adapter that allows tooling to import information required to build a service interface. This SPI provides two essential functions: the discovery of EIS metadata as described previously and the import of that metadata (and subsequent generation of service metadata) that is required to build a service interface.

The following examples and diagrams involve hypothetical adapters running in hypothetical tool environments. These examples and diagrams are intended to illustrate the concepts in this white paper and are not indicative of how adapters and tools choose to represent or visualize metadata.

Discover of communication configuration

Metadata about communication protocols is retrieved and used separately from the metadata about system capabilities because tools need to establish an outbound connection to the EIS before browsing metadata about system capabilities. The Metadata Discovery Service defines methods for discovering the communication protocols supported by the adapter.

For outbound communication (from client to EIS), this white paper uses the term *outbound connection* to represent the data channel that carries data using a given protocol. For inbound communication (EIS to client), this white paper uses the term *inbound connection* to represent an active component that listens for any communication initiated by the EIS.

The Metadata Discovery Service allows an adapter to describe the types of connections it supports by returning metadata about them. This metadata takes the form of properties that describe the configuration required to establish a connection of a given type. Properties have a name, Java type, default value and description. Properties may define enumerated values that represent the valid values for the property. Properties may also be collected into property groups to allow related properties to be displayed together in the tool.

Tables 1-3 and Figure 2 relate to a fictitious database adapter. The Metadata Discovery Service for this adapter might return several outbound connection types (Java Database Connectivity [JDBC] for running queries against different types of databases) and a single inbound connection type (for listening to database table changes through a trigger). These types and the properties that define them are provided in Table 1.

Property Name	Type	Default	Enumerated?	Description
URL	String			JDBC URL
UserName	String			User name for login
Password	String			Password for user given by user name

Table 1. JDBC outbound connection type

Property Name	Type	Default	Enumerated?	Description
ID	String	acme		Unique ID for database instance
Host	String			Name of host machine
Port	Integer	1501		Port to connect to on host machine
UserName	String			User name for login
Password	String			Password for user given by user name

Table 2. ACME outbound connection type

Property Name	Type	Default	Enumerated?	Description
URL	String	acme		JDBC URL
UserName	String			User name for login
Password	Integer	1501		Password for user given by user name
EventCatalog	String			Name of the catalog holding the staging tables for the inbound connection
EventSchema	String			Name of the schema holding the staging tables
PollingInterval	Long	2000		How long to wait (in millisec-onds) between polling sessions
ExceptionHandling	String	'Log and continue'	'Log and continue', 'Shutdown'	What the inbound connection should do if an error is encoun-tered while polling

Table 3. JDBC inbound connection type (polls staging tables in a generic database)

The Discovery Service might also choose to group the URL, user name and password properties in a property group named *outbound connection*, to signify that these properties represent the information needed to establish the polling outbound connection for the inbound connection. The discovery service might also define a property group called *inbound connection* containing the EventCatalog, EventSchema, PollingInterval, and ExceptionHandling properties.

Using this information, the tool can render form dialogs that allow the tool user to see the entire required configuration for a connection type and easily provide values. A representation of the JDBC inbound connection type is shown in Table 3 as an example of how a tool might display a form dialog for obtaining this information.

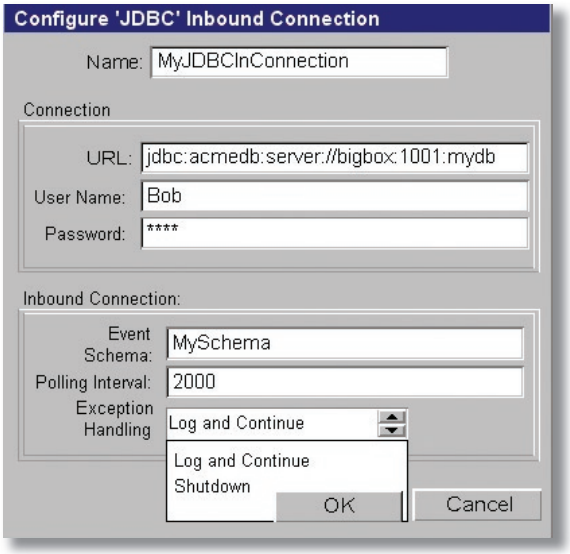

Figure 2. Inbound connection dialog for JDBC inbound connection type

Discovery of system capabilities

The discovery of system capabilities (e.g., functions, business objects, and the interaction styles they support) is provided through a set of navigation methods that allow the tool user to navigate through a hierarchy of objects (or, a *tree*) that represents the functions and business objects provided by the EIS or by the adapter. Discovery of system capabilities generally (though not always) requires the tool user to establish an outbound connection (which this white paper refers to as the *metadata discovery connection* to the EIS for purposes of browsing the metadata model. See the D*iscovery of Communication Configuration* section for details.

Figure 3 depicts the metadata tree exposed from the discovery service for a hypothetical database adapter. Note that different tools display the metadata tree in different ways and that Figure 3 is just one example of how a tool might do so.

Some objects are "active" and support import. The tool should highlight them. Other objects are just for grouping or information purposes.

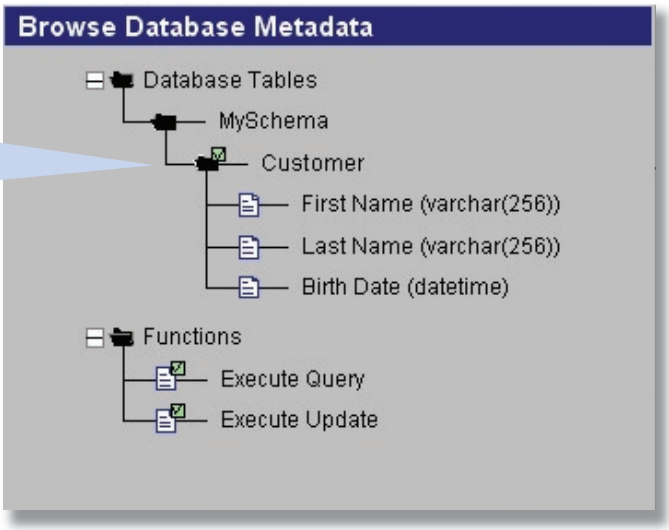

Figure 3. Metadate tree

In some cases, the number of possible functions and business objects is very large, so the Metadata Discovery Service navigation methods allow the resource adapter to return this information for only a single level of the hierarchy at a time. This facilitates navigating a large and deep hierarchy without having to retrieve the entire hierarchy. The structure of the tree is entirely determined by the adapter's discovery SPI implementation, but ideally should follow the organization in the EIS. Each node in the tree is described by a set of properties, including whether a particular node represents a function for which metadata can be imported. Some nodes in the tree cannot support import capabilities and are used to represent the structure of the tree, hold child nodes in the tree, or simply provide descriptive information.

Augmenting EIS metadata

In some instances the resource adapter may optionally augment the EIS metadata. In those cases, the discovery service may also display functions and business objects in the metadata tree that are provided by the resource adapter. Figure 4 shows how the adapter provider has added value by allowing the user to save Structure Query Language (SQL) queries for later use (most relational database management systems (RDBMS) don't allow for native storage of SQL queries, but this is a common need for database users).

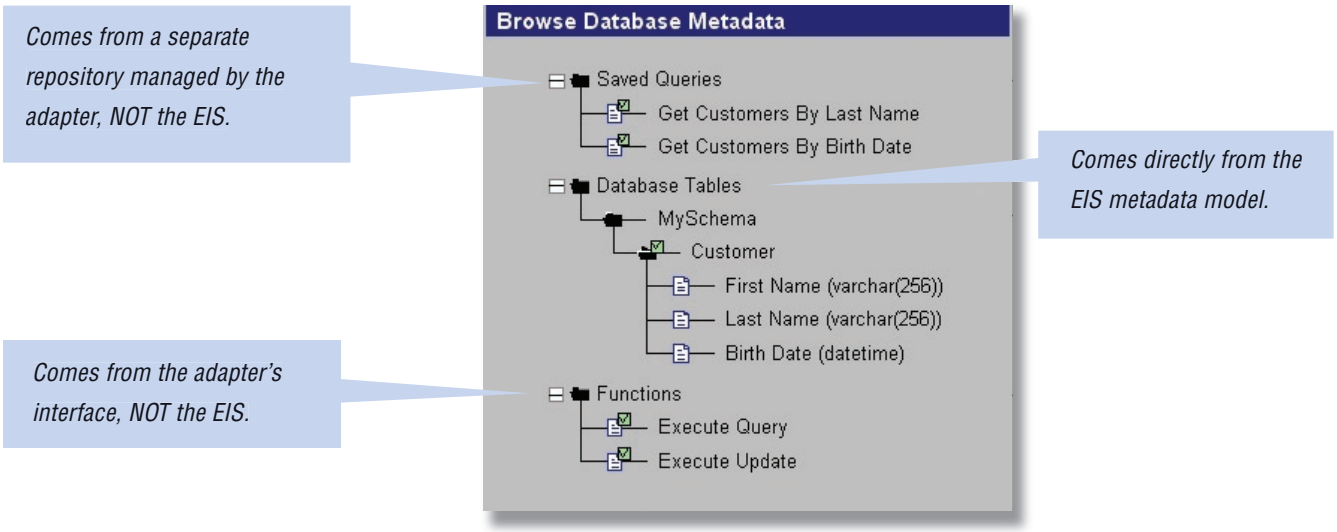

Figure 4. Metadata tree

Figure 4 assumes that the database adapter provides some tool (accessed by the user outside the integration tool that hosts the discovery service) to populate the *Saved Queries* information shown in the diagram. This white paper assumes that the saved queries are stored either in a table within the database instance, or as a file somewhere other than the database instance. The discovery service can locate the extra repository of saved queries by using standard JCA mechanisms where the location of the repository might be stored in a property on a ResourceAdapter, ManagedConnectionFactory or InteractionSpec instance.

Importing objects

The user, through the use of the tool, identifies one or more functions or business objects of interest, and selects for import the nodes representing them within the tree. For details of what the import process involves, see the *Metadata Import* section of this white paper. The discovery SPI provides the methods to generate the service metadata describing the identified functions or business objects. The service metadata is embodied in the service description. Service descriptions are described in the next section.

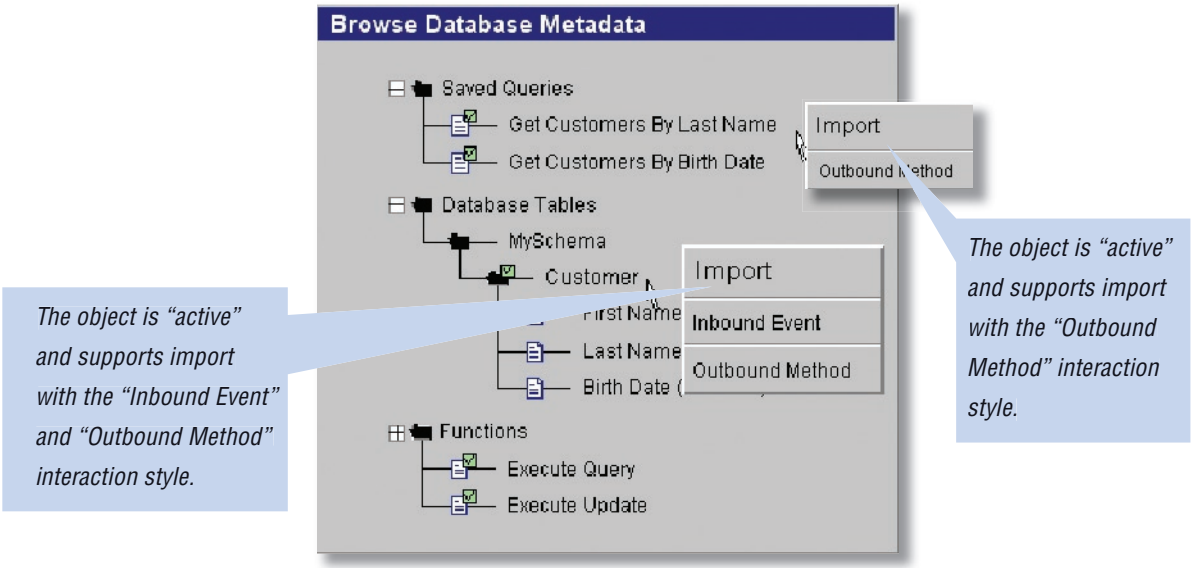

Figure 5. Selecting objects for import

Discovery of data structures

Metadata about data structures is retrieved and used by the Metadata Discovery Service (at the request of the tool) as the tool user imports functions and business objects into the service description. The data structure metadata is used to define data description instances that are contained within a function description for the function being imported. The function description and the data descriptions it contains are hosted in the service description. Service, function and data descriptions are described in later sections of this white paper.

Saving a metadata discovery connection configuration

Configuring a metadata discovery connection may require a significant amount of work on the part of the tool user. In some cases, the tool user may not even have the information needed to establish a connection instance for the EIS. In either case, the metadata discovery service defines a facility by which connection configuration can be saved for later retrieval and use.

The tool should store the discovery connection configuration. However, the discovery connection configuration may be stored directly by the discovery service of the resource adapter if the resource adapter chooses. Regardless of what actually saves the configuration, the tool should provide a list of saved connection configurations from which the tool user can choose.

Metadata import

The goal of the Metadata Discovery Service is to allow a tool user to easily define a service to accomplish a given business task. The service is represented by a service description; thus, the ultimate goal of the Metadata Discovery Service is to produce service descriptions. The creation of service descriptions is accomplished by using the metadata import facilities of the metadata discovery service.

The tool user identifies objects and interaction styles for import. The tool then directs the Metadata Discovery Service to import each object-style combination (called an *interaction*) as a function on the service description. It is the responsibility of the Metadata Discovery Service to generate a function description (see the *Service description* section later in this white paper for details) for each interaction, and include in that function description all the information necessary to carry out the interaction at run time.

For some interactions, the Discovery Service may require extra information to accomplish the import, while for others it may not. The need (or lack thereof) for extra information is expressed by the Discovery Service by returning a list of required properties (or an empty list) in response to the initial import request from the tool. If no extra information is required, the interaction can be imported and a function description can be generated for it with no further information from the tool user. However, if more information is required, the tool displays the properties indicated by the Discovery Service to the user to allow the user to provide values for those properties.

Figure 6 shows the dialog that might be shown when attempting to import the Customer table node with the *Inbound Event* interaction style using the previous hypothetical database adapter example. In this case, the adapter is being directed to deliver events any time data in the customer table changes. Databases can detect inserts, deletes, and updates on database tables, so the database adapter needs to know what type of changes the user is interested in.

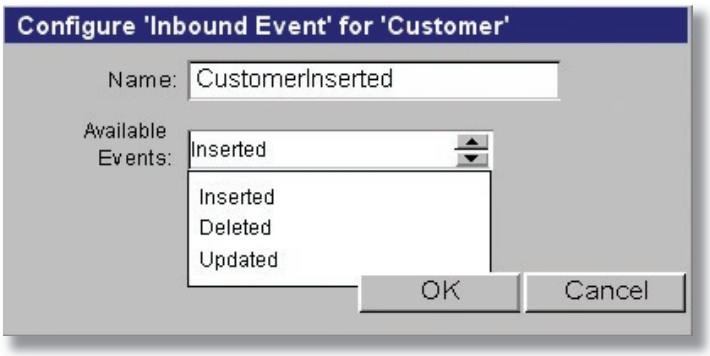

Figure 6. Configuring extra information for an interaction

Once the discovery service has all the information it needs, it proceeds with the import and obtains a function description for the interaction.

Service description

The service description embodies the service metadata generated by the Metadata Discovery Service in response to import requests made by the tool user. The service description is used either to generate the service interface and implementation, or directly by an interpretive run time to invoke the resource adapter. The metamodel for this information is comprised of the following service descriptions:

- *Service description*
- *Function description*
- *Data description*
- *Outbound connection description*
- *Inbound connection description*

Service, function, and data descriptions are present for both inbound and outbound services. The models for the service and function descriptions differ slightly for inbound and outbound connections to be consistent with the JCA, Version 1.5 specification. The data description is identical for both inbound and outbound connections.

Inbound metamodel

The inbound metamodel is used to describe a service that conveys information from the EIS to the client. It represents a set of mappings, each of which link a function that was implemented in the EIS to a particular method on the inbound service interface that is called by the inbound connection at run time. The inbound metamodel also contains descriptions of the data types to be used, and the classes mandated by the JCA, Version 1.5 specification.

Inbound service description

An inbound service description contains the following information:

- *Service name*
- *Inbound connection description*
- *EIS function selector – class packaged with the resource adapter that takes as input the message received from the adapter and returns the name of the EIS function the message represents. This can be used by the run time to determine what function on the inbound service interface the inbound connection calls.*
- *Function list – list of the functions that can be called by the inbound connection. An inbound function description depicts the interface for each of these functions.*

Inbound connection description

An inbound connection description contains the following information:

- *ResourceAdapter – an instance of the ResourceAdapter JavaBean that is used to configure a resource adapter instance.*
- *ActivationSpec – an ActivationSpec instance that is passed to ResourceAdapter. endpointActivation() at run time to support the functions defined on the inbound service that this inbound connection belongs to.*

Inbound function description

An inbound function description contains the following information:

- *Method name.*
- *EIS function name—represents the EIS function that results in the inbound message.*
- *Input and output data types—each data type is defined by a data description.*

Outbound metamodel

The outbound metamodel describes the service interface used to invoke the adapter to run requests against the EIS. It includes a description of each function defined for the service. The inbound metamodel also contains descriptions of the data types that are to be used, and the classes mandated by the JCA, Version 1.5 specification.

Outbound service description

An outbound service description contains the following information:

- *Service name.*
- *Outbound connection description.*
- *Function list—list of the functions that can be called by the client. An outbound function description describes the interface for each of these functions.*

Outbound connection description

An outbound connection description contains the following information:

- *ResourceAdapter—an instance of the ResourceAdapter JavaBean that is to configure a resource adapter instance.*
- *ManagedConnectionFactory—an instance of the ManagedConnectionFactory JavaBean that is used to configure an instance of the ManagedConnectionFactory to create run-time outbound connections.*

Outbound function description

An outbound function description contains the following information:

- *Method name.*
- *Input and output data types—each data type is defined by a data description.*
- *InteractionSpec specification—an instance of the InteractionSpec JavaBean that can be used by the adapter at run time to control the behavior of the function.*

Data description

The data description is common to the inbound and outbound metamodels. It includes a definition of the structure and content of an EIS-neutral data object that is to be passed between the client and adapter at run time. The data description enables the client to create the proper data objects for requests and interpret the data objects returned as responses.

The data description also includes any information required by the adapter to marshal the EIS-neutral data objects to the native objects used to interact with the EIS.

The data description contains the following information:

- *XML schema list—one or more XML schema documents, each one with an associated name (similar to the file name of a .xsd file). These schemas define the structure and content of the data object represented by this data description. Generally, one schema file is sufficient to describe simple data types. However, more complex data types may require multiple schemas that have been factored out of a larger conceptual schema (possibly containing shared types and so on)*
- *Root schema name—name of the schema within the XML schema list that contains the root element definition corresponding to the data object.*
- *Root element name—name of the root element definition within the schema designated by the root schema name property.*
- *Data-binding generator—class that can generate a data-binding implementation for use with this data description. See the* Data binding *section for details.*

In general, adapters can be expected to return XML schema describing their internal data structures. Therefore, adapters are required to be able to interpret their own internal data structures and describe them in terms of XML schema in the Discovery Service.

The tool takes a data description generated by the Discovery Service and provides appropriate service data object (SDO) DataObject implementations for them. The tool may choose to provide a generic implementation or to generate a strongly-typed implementation based on the data description. At run time, adapters are expected to marshal a request DataObject to their internal data format, and to unmarshal a response from their internal data format into a DataObject. The marshal and unmarshal facility is embodied in an object called data binding.

Data binding

Enterprise Metadata Discovery supports SDO as the preferred data programming model for JCA applications. It is also important to align on one canonical model for types, so that discovery service and tool providers do not need to implement many type models or be forced due to resource constraints to choose a subset of type models to support and miss market opportunity as a result. This canonical model must support editing, because the Discovery Service may not be able to fully populate the application-specific information in the model or because the names for properties or types may not be meaningful.

These capabilities can be achieved by modeling the input and output data types as XML schema definition (XSD) (3). The tooling framework can generate SDO from XSD, and XSD editors can be used to edit the data type.

A data binding converts request and response data between the canonical SDO DataObject form used by the service interface and the native object form used by the CCI of the adapter. The data binding is provided by the Discovery Service provider or code-generated by the Discovery Service at run time. It can accept a DataObject request and convert it to a native object request. It also can accept a native object response and convert it to a DataObject response. The generated service implementation uses the data binding directly to perform these conversions.

The discovery service provides an object called a *data-binding generator* that can produce the proper data-binding implementation for use with a given data description. The tool calls the data-binding generator when generating the service implementation. The returned data binding is used within the generated service implementation.

The service-method implementation creates an instance of the data binding indicated by the request data description. The method implementation takes its request DataObject and sets it on the data binding. This gives the data binding the opportunity to convert the DataObject to native objects.

The implementation has two choices of how to pass the data to the adapter's CCI. It can pass the instance of the data-binding class, which implements the *Record* interface, or it can retrieve a native version of the request (as a CCI Record instance) from the data binding and pass it to the adapter's CCI. The choice is based on the type of implementation method used by the resource adapter. For the input-only method, the implementation passes to the CCI a native version of the request. For the input and output method, the data binding is passed. For the response, the method implementation creates an instance of the data binding indicated by the response data description and again, depending on the implementation method, either passes the response data binding to the adapter CCI or takes the native response (as a CCI Record instance) and sets it on the response data binding. Either method gives the data binding the opportunity to convert the native response into a DataObject response. The DataObject instance is then returned directly from the method to the caller or client.

Editing data descriptions

In some cases, the data description defines an SDO DataObject interface that is not particularly useful or understandable to the tool user. In some cases, simply changing the names of elements and attributes in the schema for a data description can solve this. However, any change to a schema can break the implicit contract between the adapter and the client at run time. The adapter may not be aware that the schema content has been changed and fail trying to find elements in the DataObject based on old element names.

To address this, it is recommended that schemas provided by the Discovery Service in the data description contain annotations that hold the original element or attribute names, and any other information the adapter might need to generate a data binding that can properly handle marshaling. With the inclusion of annotations in the schema (containing the original element or attribute names), it is now possible to allow the tool user to edit the element or attribute names in the schema and still retain the names needed by the data binding and adapter at run time.

However, the data binding generated for the original schema may now also be inappropriate for the new schema. For this reason, the tool should always query the data-binding generator for an updated data binding, specified for the data description, whenever the schema content is edited. See the *Data binding* section for more details.

Handling Java objects and other special content

In some special circumstances, an adapter may not wish to convert its native objects to DataObjects. For example, it may be very inefficient to parse a non-XML data stream and store it in a DataObject. An example of such a stream would be large audio or video streams. These streams are essentially unstructured and do not match the highly structured model used by DataObjects. Another case where DataObjects may not be appropriate is when the native object defines complex behavior and is not simply a container for data. In this case, converting the object to a DataObject would constitute a loss of functionality.

In these cases, the Discovery Service can generate a data description that contains a schema with special annotations in it. These annotations are specific to, and defined by, the discovery service itself. They can contain any information the discovery service wants to put in them. The annotations are used to signal the data-binding generator to handle the data description in a particular way.

When the data-binding generator detects that special handling is required (through the information the Discovery Service put into the schema annotations), it generates a CCI Record JavaBean instead of a data binding. The CCI Record JavaBean must have a public default constructor and *getter* and *setter* methods for each property mapped from the member elements of the XML schema. It may also have additional methods generated from the application-specific information contained in the schema. In the audio and video example mentioned previously, the generated CCI Record JavaBean would provide enhanced support in streaming large audio or video streams. This might include a method to get a Java InputStream instance from the CCI Record to allow for efficient retrieval of the audio or video content. In the case where the native object defines complex behavior, the CCI Record JavaBean might define a method to return the native object for direct use by the client.

During generation of the service implementation, if the tool sees that the data-binding generator did not generate a data-binding implementation, it uses the generated CCI Record JavaBean type directly. This allows the generated method implementation to bypass the DataObject-to-Record and/or Record-to-DataObject conversion described in the *Data binding* section earlier. This means that the generated service interface can use the CCI Record JavaBean type as the argument type for the method. This also means that the generated implementation of that method can pass the CCI Record instance directly to the CCI interface of the adapter (for requests) and return the CCI Record instance directly to the caller (for responses). This allows a client using the service interface to have strong type information and easily locate and use extra methods such as the audio and video streaming methods and native object access methods described previously.

Sample scenario

The following sections describe a typical enterprise application integration (EAI) scenario and how the metadata discovery and import extensions described previously are used to provide tool pluggability for the adapters involved in the scenario. The scenario describes a common user experience offered by tools based on the ideas in this white paper. This user experience is enabled by the metadata discovery and import capabilities described previously, and would not be possible from a generic tool using only the facilities defined by CCI.

The scenario description is broken down into three pieces:

- *A description of the tool's interaction with the Metadata Discovery Service. This piece of the scenario represents facilities that are of importance to tool providers.*
- *An example of what discovery services do to generate the service descriptions created by the user during the discovery and import phase. This piece of the scenario represents facilities that are of importance to discovery-service and resource-adapter providers.*
- *An example of what the tool does to generate a service implementation based on the service descriptions defined during the metadata discovery and import phase. This piece of the scenario represents facilities that are of importance to tool providers.*

Roles

The scenario involves interaction with several human roles, described later. These roles include:

- *Business analyst, or EIS subject matter expert*
- *Java developer*
- *System administrator*

Ann Analyst (business analyst, or EIS subject matter expert)

Ann Analyst understands enterprise information systems and the overall business processes in the enterprise. She is well-versed in the client tools of the major enterprise systems in use within her company. Generally, she deals directly with the EIS client tools, but on occasion uses the client tools for an integration environment or application server that links those enterprise systems together. The latter point underscores the fact that the domain expert often does not communicate her domain expertise directly into the client tools for the integration environment. Rather, she often communicates her expertise through a third party that is focused on working with the tools provided by the integration environment.

Ann is an expert in enterprise systems such as enterprise resource planning (ERP) and customer relationship management (CRM), and knows which business objects offered by those systems are appropriate for use in business processes hosted in the integration environment or application server. She can look at the data definitions for a business object or function and understand the elements contained in them. She also is responsible for describing to developers the usage of business objects in the enterprise system. The most important characteristic of an integration tool environment to Ann is that it makes it easy for her to communicate information about business objects and functions to developers. The developers can then take responsibility for implementing new interfaces to the enterprise systems. This leaves Ann free to concentrate on the overall use of enterprise data and not the particulars of the interfaces to the enterprise systems that host it.

Closest J2EE Role: Ann's role does not map to a J2EE role.

Sally Scripter (Java developer)

Sally Scripter is the tool user. She is proficient at using the tool environment to create process-based solutions. She has a good understanding of how the tool environment allows her to make outbound calls to an EIS and receive inbound notifications and calls from the EIS. However, Sally looks to Ann Analyst to provide connection details for each EIS involved in a business process and the specific functionality or business objects to use within the process. She also looks to Ann to guide her on the interpretation of business objects, data types, and so on. The most important characteristics of an integration tool environment to Sally are:

- *Make it easy to communicate with Ann about business objects and their use*
- *Make it easy to define new services backed by enterprise systems.*

Closest J2EE Role: Sally would be considered to be an application component provider and application assembler.

Allie Admin

Allie manages day-to-day administration of applications. She generally takes over an application after Sally has finished developing and testing it. Allie uses the application server's management console to stop and restart applications, handle outage from an EIS going down, change an existing application to point to a different instance of an EIS, set user roles and permissions, and provide status and statistics on the general health of the system. Allie may also be involved with installing and configuring resource adapters within a tool environment.

Closest J2EE Role: Allie would be considered to be a deployer and system administrator.

ERP-to-CRM synchronization

In this scenario, this white paper discusses the synchronization of customer account information between a CRM system and an ERP system. The CRM system originates the customer information, and is considered to be the master of customer information in this scenario. The ERP system needs the customer information contained in the CRM system to properly fulfill orders from the customer. For example, the ERP system needs information about the customer's name, address, and even possibly their tax-identity information (e.g., Social Security number). The ERP system may use a customer's tax-identity number to conduct credit checks before authorizing the completion of an order. To actually ship the order to the customer, the ERP system needs the customer's address.

The problem to be solved in this scenario is caused by information being duplicated in more than one system in the enterprise. Thus, a change to customer information in the CRM system (e.g., the customer moves and his or her address changes) leaves the ERP system's customer information in a stale state. This can lead to delays in shipping and annoying calls to the customer to obtain updated information. To avoid this problem, the customer's information needs to be synchronized between two different customer databases. This synchronization involves detecting that the customer's information has changed in the CRM system, and then getting the updated customer information propagated into the ERP system.

The initial impetus to create a synchronization solution comes from Ann Analyst. Ann determines that the shipping delays and inconvenience placed on a customer without such a solution is intolerable. Ann Analyst commissions Sally Scripter to develop a solution to synchronize customer information from the corporate CRM system to the corporate ERP system. Ann tells Sally that the integration solution must respond to the creation of new customers or the update of existing customers in the CRM system. The response to these conditions should be to create a new customer or update an existing customer in the ERP system, respectively.

Sally Scripter gets some preliminary information from Ann about which systems are involved, and which business objects are involved in the integration solution Ann has requested. Sally decides to use the integration environment's business-process tool and adapters to implement the integration solution. Sally checks with Annie Admin to make sure the needed adapters are available and installed in the integration environment (e.g., tool and run-time server) she plans to use for development of the integration solution.

For her application, Sally needs to:

- *Create a process to host synchronization business logic.*
- *Define a service for the CRM system that receives an event when customer information has changed.*
- *Define a service for the ERP system that propagates the updated customer information.*

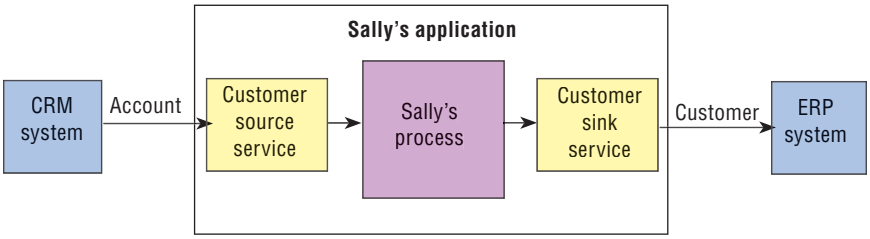

Figure 7. Creating a new application to synchronize customer information company-wide.

Create a process to host synchronization business logic

First, Sally creates a business process that can use the services she defines to the CRM and ERP systems. Tools from different vendors implement the concept of a business process in different ways. For this reason, this scenario does not cover how Sally creates the business process, or the form the business process takes.

All that's certain is that the business process is intended to respond to the creation of new customers or the update of existing customers. To support this, Sally must develop a service to the CRM system that defines events (called *AccountCreated*, and *AccountUpdated*) for these conditions in the CRM system.

Sally's business process must respond to AccountCreated and AccountUpdated events by creating or updating customer information in the ERP system. To support this, Sally must develop a service to the ERP system that defines functions (called customerCreate, and customerUpdate) to provide these capabilities with respect to the ERP system.

Defining a service for the CRM system

The following sections describe the creation of the service for the CRM system from the perspective of the tool user, and then the discovery service or adapter developer. The creation of the service, from the tool developer perspective, is described in the *Generate the service in the tool* section.

User perspective

The following sections describe what Sally Scripter does in the tool environment as she defines services representing the CRM and ERP systems. These services are to be employed in the business process used to integrate customer data between the two systems. The steps described in the following sections are all supported by the metadata provided by the adapter through its Metadata Discovery Service.

1. Pick outbound or inbound service
Sally chooses to define a new inbound service to allow her business process to respond to events from the CRM system. Note, some tools or resource adapters' discovery services may not require Sally to choose an inbound or outbound service until the first function is imported to the service below.

2. Name service
Sally names the service CustomerSource to indicate that customer information originates from this service (and is consumed by the ERP service). Note that some tools may not require Sally to choose a name for the service until it is saved, or the tool generates the implementation for the service.

3. Pick adapter
Sally asks the tool for a list of available adapters. The tool searches for all RAR modules that have been registered with the tool (prior to Sally starting the tool, and in a tool-specific fashion). For each RAR module that is found, the tool locates the Metadata Discovery Service implementation for the adapter module (through information registered at install time in the tool). The instance of this implementation that is to be used from here on is referred to simply as the *Discovery Service*.

The list of adapter modules is shown in the tool. It can be assumed that the list contains two adapter modules, *CRM system*, and *ERP system*. Sally sees that the list contains the entry *CRM system*, and selects this adapter module as the adapter module she wants to work with.

4. Connect to EIS instance through discovery connection
After picking the adapter module to work with, the next step is for the tool to obtain an outbound connection (the discovery connection) to the EIS for use in browsing and importing metadata. This browsing exercise allows Sally to pick the business objects from the CRM system that represent a customer (as she was instructed to do by Ann Analyst).

The tool attempts to make this step easy for Sally to complete by providing a list of named connections that have been used previously in the tool to connect to this type of EIS. See the *Saving metadata discovery connection configuration* section for more information on this subject.

It can be assumed that this is the first time the CRM system has been used from the tool in this environment, and because of this, Sally asks the tool to create a new connection. The tool queries the Discovery Service for specific connection types that are available for the selected adapter type. The Discovery Service provides a set of connection type descriptions that define the information needed to establish a new instance of this type of connection.

For each connection type returned, the tool checks to make sure that it is supported for browsing purposes (some enterprise systems allow only specific connection types to browse their metadata, and use different connection types for run-time invocation). The tool shows Sally a list of possible connection types. The list includes TCP/IP socket and offline metadata cache.

In this case, Sally wants to see the most up-to-date metadata, so she chooses the TCP/IP socket connection type. This connects her to a live CRM system instance, and she can use the connection to interrogate what business objects are available to her. Note that the offline metadata cache connection type might be useful if the CRM system were unavailable for some reason. This connection type might use a file-based repository of information that is a snapshot of the EIS metadata at some point in the past.

The tool interrogates the connection type to find what connection parameters are required to establish a TCP/IP socket connection, and then presents Sally with a form suited to obtaining this information. It can be assumed that the TCP/IP socket connection type requires the parameters shown in Table 4.

Property Name	Type	Description
Host	String	IP address or host name of the EIS host machine
Port	Integer	Port number on the host on which connections are to be accepted
UserName	String	Name of the user for which the connection are to be established
Password	String	Password for the user

Table 4. TCP/IP socket connection parameters

Sally enters the required information that she obtained from Ann Analyst during their initial consultation. Sally then clicks a *connect* button on the form to establish the discovery connection. The tool asks the discovery service to establish the physical connection to the CRM system using the parameters Sally entered and stores a reference to this connection for Sally's use.

5. Save the discovery connection (optional)
At this point, the tool could offer Sally a way to save the discovery connection configuration she has just created. This may be as simple as a *Save Current Configuration* menu or button accessible from somewhere in the tool, or the tool may explicitly ask Sally if she wants to save her outbound connection at this point.

It can be assumed that Sally chooses to save her outbound connection. When prompted for a name by the tool, Sally enters the name *East Coast CRM* and it saves the configuration under that name. This makes the East Coast CRM connection available during the next use of the tool.

6. Browse metadata tree
The tool, having established a discovery connection, asks the discovery service to return information on the business objects, functions and so on available from this instance of the CRM system. The tool fetches a representation of the CRM system's metadata model one level at a time, and asks the metadata service to list the root objects of the CRM system's metadata model. It then displays these objects (probably within a tree user-interface [UI] component) to Sally.

Sally then selects a node in the metadata tree, and drills in one level at a time until she reaches the designated business object representing a customer. Ann Analyst gave this information to her. Each time Sally drills in to another level in the CRM metadata model, the tool again asks the discovery service to return a list of objects, giving the currently selected object as context for the list option. The discovery service uses the location of the currently selected object to find the objects that exist as children of the object in the metadata tree. This allows the tool to fill out its tree UI component one level at a time (It also allows a resource adapter to avoid, when it is costly, a depth-first traversal through a potentially very large metadata model).

Sally eventually picks a business object called *Account* from a location in the CRM metadata model called *Business Objects->Standard*. This indicates that Sally has navigated first to the Business Objects node in the UI tree, then to the *Standard* node, and finally to the *Account* node.

7. Pick object and interaction style to represent a function
Sally clicks on the Account node in the tree to see what styles of interaction she can have with this object. The tool directs the Discovery Service to return detailed information about the object Sally has selected. The returned object indicates that this object supports the inbound-event and outbound-method styles of interaction. Sally wishes to receive events from the Account business object, so she selects the *Inbound Event* style.

8. Import object or function into a service as a new function
Sally then indicates she wants to import the selected Account business object and inbound event interaction style into the *CRMSource* service description she is defining. At this point, the tool directs the Discovery Service to describe any extra information that is needed to carry out the delivery of the event.
In the case of the Account/Event interaction, the CRM system needs to know what type of event to send. It is capable of sending events when an account object is created, updated, or deleted. The Discovery Service indicates the required information by directing the tool to provide values for a specific set of named properties. The discovery service for the CRM adapter returns a set of properties containing one item, shown in Table 5.

Property Name	Type	Description
AvailableEvent	String	Enumerated Values: Created, Deleted, Updated

Table 5. Values provided for a specific set of named properties

The tool notes that the interaction requires extra information from Sally and displays a dialog to allow Sally to provide that information. This dialog likely shows a pick list for the AvailableEvent property, and shows the three enumerated values provided by the adapter. In this case, Sally is interested in created events, so she selects *Created*.

The tool passes the property values for the interaction back to the Discovery Service and asks the Discovery Service to generate a function description that can be added to the CRMSource service description Sally is working on. The Discovery Service picks a default name for the function based on the object type and interaction style Sally chose. In this case, the Discovery Service picks the function name *AccountCreated* to signify that the event represents the creation of a new customer record (called Account in CRM) in the CRM system.

The Discovery Service uses the information gathered for the interaction (including the business object Account, AvailableEvent Created, and default function name AccountCreated) to generate a function description that reflects the inbound function the adapter will invoke at run time to deliver events when a new customer (Account) entry is created. The function description defines the shape of the event by generating a data description that features an XML schema instance defining the structure and content of the event data that arrives from the EIS at run time. This data description contains the schema and a name for the schema (similar to a file name for the schema).

It can be assumed that the Account business object in the CRM system defines a customer's name, address, and other information such as the last time a sales representative contacted the customer. The schema generated for the data description reflects the structure of the Account business object and looks something like this:

```
Schema name: Account

<?xml version="1.0?>
<s:schema targetNamespace="java:/com/myco/crm2erp"
       xmlns:ns="java:/com/myco/crm2erp"
       xmlns:s="http://www.w3.org/2001/schema">

  <s:element name="Account" type="ns:AccountType"/>
```

```
  <s:complexType name="AccountType">
    <s:sequence>
      <s:element name="Name" type="s:string">
      <!-- This is an example of application-specific field info -->
        <s:annotation>
             <s:appinfo xmlns:asi="urn:AppSpecificInfo">
                  <asi:property>
                    <asi:nativeName>FullName</asi:nativeName>
                  </asi:property>
          </s:appinfo>
        </s:annotation>
         </s:element>
      <s:element name="Address" type="ns:AddressType"/>
      <s:element name="LastContacted" type="s:dateTime"/>
    </s:sequence>
  </s:complexType>

  <s:complexType name="AddressType">
    <s:sequence>
      <s:element name="Street" type="s:string"/>
      <s:element name="City" type="s:string"/>
    </s:sequence>
  </s:complexType>

</s:schema>
```

The tool then returns Sally to viewing the current state of the service description she is working on. The service description now contains a single function called *AccountCreated* and that function takes an SDO data object as its single parameter. The SDO object's type is determined by the name of the schema that is returned in the data description for the function. The tool is responsible for providing the SDO implementation. The tool can choose to generate the SDO classes or provide a generic implementation at any point up until the service is actually deployed and used. Typically the tool designates the SDO classes at the time Sally invokes the tool's build action to cause the service description itself to be compiled into usable Java interfaces.

9. Repeat object and interaction style selection and import as needed
Sally repeats the browse and import scenario for the Account/Updated event. Note that in the case of an Updated event, the CRM adapter provides two data elements containing both the new and old versions of the effected Account object. The schema might look something like this:

```
Schema: AccountUpdate.xsd

<?xml version="1.0?>
<s:schema targetNamespace="java:/com/myco/crm2erp"
       xmlns:ns="java:/com/myco/crm2erp"
       xmlns:s="http://www.w3.org/2001/schema">

 <s:element name="AccountUpdate" type="ns:
AccountUpdateType"/>

 <s:complexType name="AccountUpdateType">
  <s:sequence>
   <s:element name="New" type="ns:AccountType"/>
   <s:element name="Old" type="ns:AccountType"/>
  </s:sequence>
 </s:complexType>

 <s:complexType name="AccountType">
  <s:sequence>
   <s:element name="Name" type="s:string"/>
   <s:element name="Address" type="ns:AddressType"/>
   <s:element name="LastContacted" type="s:dateTime"/>
  </s:sequence>
 </s:complexType>

 <s:complexType name="AddressType">
  <s:sequence>
   <s:element name="Street" type="s:string"/>
   <s:element name="City" type="s:string"/>
  </s:sequence>
 </s:complexType>

</s:schema>
```

10. Select the inbound connection type and configure
The CustomerSource service is an inbound service. As such, it requires an inbound connection to deliver inbound communication to the CustomerSource implementation Sally is writing to handle AccountCreated and AccountUpdated events. Selecting the type of inbound connection and configuring it may be done during the import step or it could be deferred until the first time the service is actually used. This topic is covered here to illustrate what is involved.

Different tools handle associating an inbound service with an inbound connection using different UI mechanisms. Some tools may not even support associating an inbound connection with a service and may leave this task to an administration tool at run time. It can be assumed that this tool allows you to associate an inbound connection with the service. At a later time, administration tools may allow the inbound connection's configuration to be changed.

The tool asks the discovery service for a list of inbound connection types. The Discovery Service responds with a single inbound connection type called *CRM Inbound Connection*. The CRM Inbound Connection type defines a number of properties needed to configure it, as shown in Table 6.

Property Name	Type	Description
Port	Integer	Port number on which this inbound connection accepts incoming outbound connections. Note that this inbound connection type does not define a host property, because it is a server, and runs on the local host.
ExceptionHandling	String	*Log and Continue, Shutdown*. What the inbound connection should do if an error is encountered while receiving incoming communication.

Table 6. CRM Inbound Connection types

Note that these properties do not identify the CRM instance from which this inbound connection receives events. In this case, it can be assumed that there is some configuration required on the CRM system to fully describe this connection.

Sally picks the CRM Inbound Connection type. The tool then shows her a dialog that allows her to specify values for the required properties. Sally enters *1001* for the port number and selects the *Log and Continue* value for ExceptionHandling. It can be assumed that she was told what values to provide for the inbound-connection properties during her initial consultation with Ann Analyst. She accepts the values she's entered.

The Discovery Service then creates an inbound-connection description containing the inbound-connection properties Sally entered. This inbound connection description is associated with the service description for CustomerSource.

11. Build service in the tool
The service description CustomerSource now has two inbound functions: AccountCreated and AccountUpdated. Sally then asks the tool to build the service. This build process is where the tool prepares whatever it needs to run the service. It could generate tool-specific interfaces and implementation classes representing the information in the service description. It could also build SDO interfaces and implementations that allow the functions on the service to shuttle data back and forth between the client and EIS.

```
CustomerSource.java:

package com.myco.crm2erp;

public interface CustomerSource
{
   /**
    * Indicates an Account object has been created.
    * @param account
    */
   public void accountCreated(Account account);
```

```
   /**
    * Indicates an Account object has been updated. Both the new
    * and old versions of the Account object that was effected are
    * provided.
    * @param accountUpdate
    */
   public void accountUpdated(AccountUpdate accountUpdate);
}
```

Account.java:

NOTE: In all the Java interfaces generated from data descriptions, the interface may optionally extend SDO DataObject, but the implementations of those interfaces *must* implement DataObject.

```
package com.myco.crm2erp;

import java.util.Date;

public interface Account
{
  public String getName();
  public void setName(String name);
  public Date getLastContactDate();
  public void setLastContactDate(Date lastContactDate);
  public Address getAddress();
  public void setAddress(Address address);
  …
}
```

```
Address.java:

package com.myco.crm2erp;
public interface Address
{
   public String getStreet();
   public void setStreet(String street);
   public String getCity();
   public void setCity(String city);
   …
}

AccountUpdate.java:

package com.myco.crm2erp;

public interface AccountUpdate
{
  public Account getNew();
  public Account getOld();
  …
}
```

Discovery Service perspective

As Sally browses the CRM metadata tree and imports functions into the CustomerSource service description, the CRM Discovery Service must respond to these actions by providing the correct type of metadata.

Handling connection type discovery requests

In step 4 described previously, Sally is trying to configure a discovery connection that allows her to browse the metadata tree for the CRM instance she is interested in. First, the Discovery Service returns a list of supported outbound connection types. Each outbound connection type describes the properties that need to be provided to create an instance of this type of outbound connection.

It can be assumed that the CRM Discovery Service creates two outbound connection type instances (one each for TCP/IP socket and offline metadata cache). Each instance is filled in with the proper property instances to define their properties. It can also be assumed that the knowledge of the two outbound connection types and the properties they require is contained in the source code for the discovery service (i.e., the Discovery Service does not look it up elsewhere).

Handling browse requests

In step 6 described previously, Sally traverses the objects in the CRM metadata tree. The CRM Discovery Service must create or return the correct objects to populate this metadata tree. After Sally connects to the CRM instance, the tool asks the Discovery Service to list its root objects, given the outbound connection Sally has established.

The Discovery Service knows there is only one root object for all instances and versions of the CRM system, so it always returns a root object named *Business Objects*. However, the CRM system has a user-customizable object model. This means that users can create new object types within the CRM system and that the CRM metadata model reflects both built-in and user-defined objects. The CRM system allows user-defined objects as children of the root Business Objects object, but not above this level. The CRM system also allows users to define new properties for the root object (though they cannot remove it or any built-in properties for it).

Because of this, the Discovery Service cannot know how many objects exist under the root object or how many properties are defined on that object until it actually connects to the CRM system to check. It also must assume that the number of child objects and properties for the root object may be large. Therefore, the Discovery Service returns only a basic summary of information on the root object as a response to the first call from the tool to list root objects. This summary includes the name of the root object, a description of it and an indication that the root object has at least one child. This allows the Discovery Service to satisfy the needs of the tool (listing the objects at the root of the metadata tree) without causing undue overhead associated with retrieving large amounts of data.

If Sally selects the root object in a way that indicates she wants to show full details for it (which depends on the tool user interface), the Discovery Service then retrieves all the properties for the root object and an accurate count of objects defined under it. When Sally sees the Business Objects root node in the tree, she sees an indication that the node has children (possibly indicated by a plus sign next to the node). When she expands this node, the tool directs the Discovery Service to list all the child objects for the root object.

The Discovery Service then checks the location of the current (root) object and determines that it needs to list objects under Business Objects. It calls an API in the CRM client interface to retrieve a list of objects under the root object. It can be assumed that the CRM system has a way to summarize the objects at any level in its tree, and can thus avoid retrieving large amounts of property and other data from each object under the root object. For some EISs, this is not the case, or objects in the tree are small enough to retrieve full information for them. In either case, the Discovery Service must do what it can to retrieve objects efficiently using the client API available to them. Each time Sally drills in to another node in the tree, the Discovery Service repeats its search for the correct objects under the currently selected object in the tree.

When Sally finally arrives at the Business Objects->Standard->Account object, she selects it in a way that requires full details to be displayed. These details include any properties defined for the object, the object's children, and supported interaction styles. The Discovery Service makes sure all the required data has been retrieved from the CRM system through the CRM client API.

In the case of the Account object, it can be assumed that the Discovery Service just knows this object supports inbound events for creation, update and deletion. For some EISs, the Discovery Service might have to interpret metadata about the current object to make this determination. For example, certain object types in the CRM system might support these events. If the Account object were found to be of a matching type, the Discovery Service could assume these events are supported. The Discovery Service then indicates the interaction style *Inbound Event* on the metadata object returned for the Account object.

When Sally right-clicks the Account object, the tool shows her the interaction styles set by the Discovery Service for this object. Sally picks the Inbound Event style, so the tool then directs the Discovery Service to begin the process of importing the Account object with the *Inbound Event* interaction style.

The Discovery Service knows it can support one of three event types, so it knows it needs more information from Sally to complete the import. It also knows that it needs a name for the function description created by the import. It begins the import process by returning an object representing the properties required to define the interaction between the client and metadata object. This object is called an *interaction description*.

The Discovery Service adds a property to the interaction description for the inbound event type (called *AvailableEvent*) that has enumerated values of Created, Updated and Deleted. The AvailableEvent has a Java type of string. The Discovery Service picks a default value for the function name formed from the object name and interaction style (e.g., AccountCreated), and therefore does not need to ask Sally for this information.

The tool displays the interaction description in a dialog (the exact form of which is tool-specific). Sally picks the created inbound event. The tool then asks the Discovery Service to complete the import process given the newly acquired interaction information provided by Sally.

Handling import requests

The Discovery Service generates service and function descriptions in response to Sally's activities in the tool environment. The artifacts it produces in this scenario are described below.

- Service description
  The service description in our scenario is named CustomerSource and contains two functions named AccountCreated *and* AccountUpdated. *This service description is for use with EIS-initiated communication and is therefore an inbound service description.*

- Function description
  Two function descriptions (AccountCreated and AccountUpdated) are generated. These function descriptions each reference a single DataDescription.
- Data Description
  See step 9 in the Defining a service for the CRM system *section of this scenario for information on the schemas generated by the Discovery Service and held within the data descriptions. For more general information on data descriptions, see the Data Description section.*
  Here is what the adapter does when Sally imports the Business Objects->Standard->Account object.
- *Creates a function description object to represent the interaction (defined by the imported object, interaction style and properties contained in the interaction description).*
- *Generates data descriptions and the schemas for them. It can be assumed that the Discovery Service obtains some metadata description of the account object from the CRM. This CRM's metadata format need not be XML schema. The Discovery Service converts the CRM metadata to an XML Schema in a CRM-specific way. It can be assumed that the CRM metadata format is like the Java reflection model (e.g., objects have a class with methods and properties). It can also be assumed that the customer object has a class* Customer, *and properties for name, address, and last contact date. The Discovery Service creates an XML schema with complex types for each Java type (e.g., Account becomes a complex type with name* AccountType*; address becomes a complex type with name* AddressType*) and an element for each property on the Java type (e.g., name becomes an element with name name and address becomes an element with name* address *and type* AddressType*). Each element in the schema contains an annotation with the original element name. This allows the schema content to be edited without breaking the run-time assumptions described later for the data binding.*
- *Generates data binding and provides the data-binding generator. The Discovery Service uses a built-in (not generated) data binding that maps properties from the run-time native CRM objects by name into the DataObjects seen by the client. The data-binding generator simply returns the built-in data binding. The name used for the mapping comes from the schema's annotations, and not the element names (which may have been edited).*

Defining a service for ERP system

The steps for defining the service for the ERP system are similar to those for defining the service for the CRM system described previously. The descriptions in this part of the scenario are sparser than those in the CRM part of the scenario. If needed, refer to the CRM part of the scenario for more detail.

User perspective

1. Pick outbound service
2. Name the service CustomerSink
3. Pick adapter

   Sally selects the ERP System adapter.
4. Connect to EIS instance

   Sally selects a outbound connection type for the ERP System (called, in this case, Default*) and creates a new outbound connection of this type to the ERP system and specifies values for the properties described in Table 7.*

Property Name	Type	Description
Host	String	IP address or host name of the ERP host machine
Port	Integer	Port number on the host on which outbound connections are to be accepted
UserName	String	Name of the user for which the outbound connection is to be established
Password	String	Password for the user

Table 7. Outbound connection properties

5. *Browse metadata tree*

   Sally navigates to the Financials/General level in the tree and selects the Customer object under General. The Customer node has three child nodes, one for Create, Update and Delete.
6. *Pick object and interaction style to represent a function*

   Sally right-clicks the Customer node, and sees an import menu with a single item called Request/Response. *This indicates that the functions on the Customer node only support synchronous two-way outbound interactions. Sally clicks on Request/Response.*
7. *Import service*

   No further information is needed to perform the import. The ERP discovery service imports the service with function descriptions for Create, Update and Delete. The schemas for the types have the following content:

```
Schema name: CustomerCreate

<?xml version="1.0?>
<s:schema targetNamespace="java:/com/myco/crm2erp"
       xmlns:ns="java:/com/myco/crm2erp"
       xmlns:s="http://www.w3.org/2001/schema">

 <s:element name="CustomerCreate" type="ns:
CustomerCreateType"/>

 <s:complexType name="CustomerCreateType">
  <s:sequence>
   <s:element name="Name" type="s:string"/>
   <s:element name="Address" type="ns:AddressType"/>
   <s:element name="DateOfBirth" type="s:dateTime"/>
  </s:sequence>
 </s:complexType>

 <s:complexType name="AddressType">
  <s:sequence>
   <s:element name="Street" type="s:string"/>
   <s:element name="City" type="s:string"/>
  </s:sequence>
 </s:complexType>

</s:schema>

Schema name: CustomerUpdate

<?xml version="1.0?>
<s:schema targetNamespace="java:/com/myco/crm2erp"
       xmlns:ns="java:/com/myco/crm2erp"
       xmlns:s="http://www.w3.org/2001/schema">
```

```
  <s:element name="CustomerUpdate" type="ns:
CustomerUpdateType"/>

  <s:complexType name="CustomerUpdateType">
    <s:sequence>
      <s:element name="FieldUpdate" type="ns:FieldUpdateType"
              minOccurs="1" maxOccurs="unbounded"/>
        </s:sequence>
  </s:complexType>

  <s:complexType name="FieldUpdateType">
    <s:sequence>
      <s:element name="FieldName" type="s:string"/>
      <s:element name="NewValue" type="s:string"/>
    </s:sequence>
  </s:complexType>

</s:schema>
```

Schema name: CustomerDelete

```
<?xml version="1.0?>
<s:schema targetNamespace="java:/com/myco/crm2erp"
       xmlns:ns="java:/com/myco/crm2erp"
       xmlns:s="http://www.w3.org/2001/schema">

  <s:element name="CustomerDelete" type="ns:
CustomerDeleteType"/>

  <s:complexType name="CustomerDeleteType">
    <s:sequence>
      <s:element name="Name" type="s:string"/>
      <s:element name="Address" type="ns:AddressType"/>
      <s:element name="DateOfBirth" type="s:dateTime"/>
    </s:sequence>
  </s:complexType>
```

```
  <s:complexType name="AddressType">
    <s:sequence>
      <s:element name="Street" type="s:string"/>
      <s:element name="City" type="s:string"/>
    </s:sequence>
  </s:complexType>
</s:schema>
```

8. Build service in the tool

Sally clicks a *build* button in the tool to generate the service. This build process could generate tool-specific interfaces and implementation classes representing the information in the service description. It could also build SDO interfaces and implementations that allow the methods on the service to shuttle data back and forth between the client and EIS.

```
CustomerSink.java:

package com.myco.crm2erp;

public interface CustomerSink
{
   public void customerCreate(CustomerCreate customerCreate);
   public void customerUpdate(CustomerUpdate customerUpdate);
   public void customerDelete(CustomerDelete customerDelete);
}

CustomerCreate.java:

package com.myco.crm2erp;

import java.util.Date;
```

```
public interface CustomerCreate
{
 public String getName();
 public void setName(String name);
 public Date getDateOfBirth();
 public void setDateOfBirth(Date dateOfBirth);
 public Address getAddress();
 public void setAddress(Address address);
    }
```

Address.java:

```
package com.myco.crm2erp;
public interface Address
{
   public String getStreet();
   public void setStreet(String street);
   public String getCity();
   public void setStreet(String street);
}
```

CustomerUpdate.java:

```
package com.myco.crm2erp;

public interface CustomerUpdate
{
 public FieldUpdate[] getFieldUpdates();
 public FieldUpdate addFieldUpdate();
}
```

```
FieldUpdate.java:
package com.myco.crm2erp;
public interface FieldUpdate
{
   public String getFieldName();
   public void setFieldName(String fieldName);
   public String getNewValue();
   public void setNewValue(String newValue);
}
```

Discovery Service perspective

Refer to the *Discovery service perspective* section in the ERP portion of the scenario.

Generate the service in the tool

The service is generated using information from the service description. For example, the generated service could provide a Java interface that may be used by the developer to represent the service. This interface reflects the metadata held in the service description for the service. This white paper does not constrain how the tool maps service, function and data descriptions to the service. However, examples of the interfaces that may be generated by the tool are given in step 9 of the *Defining a service for the CRM system* section of this scenario.

Generally speaking, the service reflects the name of the service description and contains a single function for each function description on the service description. However, the implementation for a service may vary widely between tool vendors. Figure 7 shows the major elements of the service implementation that might be generated by a tool vendor.

This CustomerSource service is an inbound service, and thus requires a JCA, Version 1.5 compliant adapter to support the inbound messaging required to deliver the events defined by the service. The CustomerSink service is an outbound service, and thus requires at least a 1.0 style ManagedConnectionFactory (though a JCA, Version 1.5 technology-compliant adapter may be available for the ERP system).

Figure 7 shows the major objects involved in the implementation of the CustomerSource service, the CustomerSink service, and the business process that ties them together.

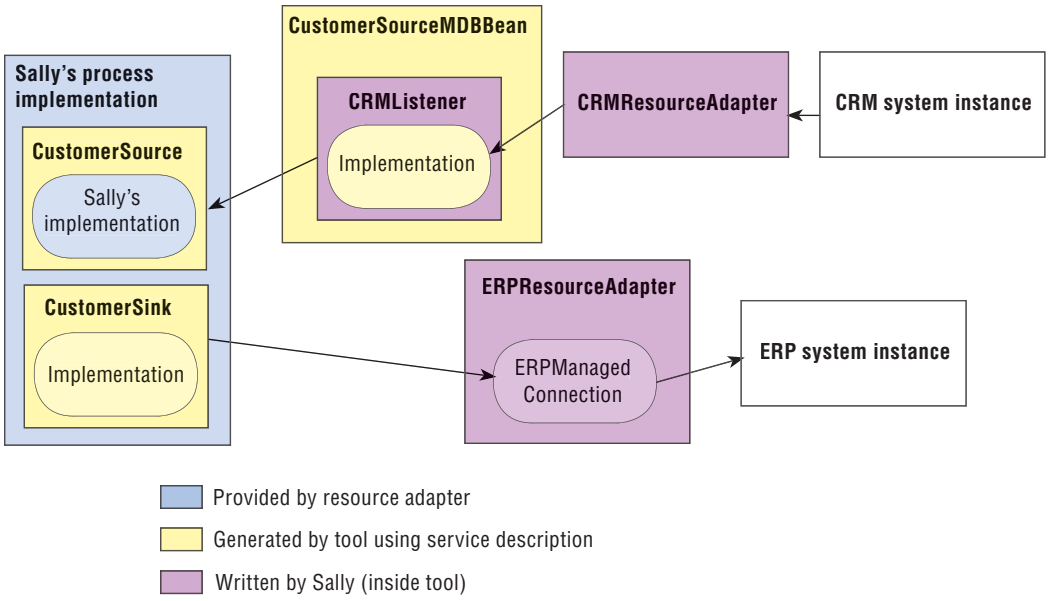

Figure 7. Application Components For CRM to ERP Scenario

Relationship to other technologies

Enterprise Metadata Discovery touches on concepts that are similar to, or relevant with respect to, several other technologies. These include service-oriented architectures, SDO, Web services, Java Metadata Interface (JMI), Java API for XML Registries (JAX-R), and others. Each technology is discussed briefly in the sections below.

Service-oriented architecture

Describing the interface provided by adapters is only useful if there is an implementation framework that can use the interface definition that is created. Two aspects are relevant: a programming model and a data model. The programming model should support a service-oriented architecture.

SDO

The emerging data model for service-oriented architectures is SDO [2]. SDO is designed to simplify and unify the way in which applications handle data. Using SDO, application programmers can uniformly access and manipulate data from heterogeneous data sources, including relational databases, XML data sources, Web services, and enterprise information systems. For more information about the goals and architecture of SDO, see the white paper *Next-Generation Data Programming: Service Data Objects*.

Web services

Web services standards and technology (such as Web Services Description Language [WSDL][4]) are related to JCA. Web services can also provide EIS connectivity. Web services is a complementary technology to JCA in that it provides standards for connectivity in a loosely coupled model that can be used to bridge between programming languages and platforms.

However, many EISs may not be amenable to the changes necessary to expose a direct Web services interface. In these cases, an adapter that supports Enterprise Metadata Discovery can be interposed between the EIS and the client. With respect to Enterprise Metadata Discovery, Web services do not provide a way to discover data and functions in an EIS, nor do they provide a facility for defining new interfaces to an EIS. Enterprise Metadata Discovery provides both.

The WS Metadata Exchange specification defines a mechanism for retrieving WS-Policy information about an endpoint or target namespace, WSDL about an endpoint or a target namespace and XML schema for a target namespace. This is done using a Simple Object Access Protocol (SOAP) implementation of three WSDL operations: GetPolicy, GetWSDL, and GetSchema. It's a targeted query that does not provide a browse or discovery mechanism. It requires the service to be described using WSDL.

Enterprise Metadata Discovery is different from WS Metadata Exchange in the following ways:

- *It uses a discovery service provided by a resource adapter that can connect to metadata in an EIS system or alternate metadata repository. It could use SOAP to connect to the repository, but that would be an implementation hidden in the discovery service.*
- *Discovery service supports browsing metadata to find the service or business object you want*
- *Discovery service supports defining a new service with an EIS JCA binding.*
- *It supports editing of the type and its application specific information, the use of SDO as a data programming model, with the use of a data binding to transform a DataObject to the native form expected by the resource adapter (i.e., cci.Record).*
- *Policy information is not discovered.*

JMI

The Java Metadata Interface (JMI) Specification, JSR040, defines a Java API for OMG's Meta Object Facility (MOF). It is aimed at providing a standard for metadata and metamodel interoperability. The goal of having a common framework for metadata interchange is to support Model Driven architecture (MDA), service-level application interoperability and metadata warehouses. Current approaches to dealing with metadata often rely solely on an XML-document type definition (DTD) or schema to define data and operations on some resource. This is considered technical metadata and is a first step in achieving interoperability. Higher-level semantic models for interoperability require something like the MOF. The Enterprise Metadata Discovery for JCA defined here is complimentary to the approach proposed by the MOF and JMI in the sense that it is primarily a low-level model for resource adapter metadata. It does not attempt to address the issues of model interoperability and thus does not conflict with JMI. It is entirely possible that a translation layer could be added to an implementation of the JCA metadata facility at a later date to provide a JMI interface to the metadata and models it defines.

JAX-R

JAX-R [7] states its main goal as providing a common API for accessing business registries such as Universal Description, Discovery, and Integration (UDDI) and Electronic Business XML (ebXML). UDDI is a fairly basic registry, and ebXML is a much broader and rigidly defined registry. It is the ebXML influence that makes JAX-R a rich and general-purpose registry abstraction.

JAX-R allows for the definition, organization and access of business data, functions, and objects (or rather metadata about them). This metadata is then organized into units of related objects through a very rich classification system. In some ways, the metadata tree described in this white paper could be thought of as a specific JAX-R classification scheme that organizes a fairly small set of object types into a hierarchical structure resembling a file system.

JAX-R also defines other facilities not described in the metadata tree discussion in this white paper. These facilities include the definition of relationships between objects in the registry, and the ability to classify a single object into more than one classification scheme. All these facilities may prove useful as the specification for Enterprise Metadata Discovery progresses.

It is IBM's and BEA's intent to align the forthcoming specification with the concepts of JAX-R and other technologies.

References

[1] J2EE Connector Architecture, Version 1.5 specification
[2] Next-Generation data programming: Service Data Objects
[3] XML schema
[4] Web Services Definition Language
[5] WS Metadata Exchange
[6] Java Metadata Interface (JMI)
[7] Java API for XML Registries (JAX-R)

For more information

To learn more about Enterprise Metadata Discovery, visit:

ibm.com/developerworks/java/library/j-emd/

[1] As a notational convention, from this point forward whenever we use the term “connection” it will mean “inbound and outbound connection” unless we explicitly indicate the type.