Black Pearl Knowledge Broker

Programmer's Reference

Version 1.3.1

Contents

List of Tables

List of Figures

x

Introduction

Welcome to the *Black Pearl Knowledge Broker Programmer's Reference*. It contains technical information intended to provide programmers, knowledge engineers, and system deployers with sufficient understanding to deploy and integrate the Black Pearl™ Knowledge Broker™ within enterprise environments.

How This Guide Is Organized

Chapter 1	*Understanding the Knowledge Broker* provides an overview of the Knowledge Broker subsystems and platform architecture.
Chapter 2	*Understanding the Ontology* provides an introduction to the Knowledge Broker's ontological approach and some of the technologies used to achieve this.
Chapter 3	*Understanding the DIS* describes the underlying Data Information System, the heart of the Knowledge Broker's data mapping and transformation infrastructure.
Chapter 4	*Descriptors in Detail* describes how to access Knowledge Broker's subsystems using descriptors, an XML-based approach to rapid application development.
Chapter 5	*Building an Application* outlines a simple demo application for the Knowledge Broker using descriptors and also provides some example Java code that ullustrates how to program the control flow.

Audience

This guide is intended for those people who will use the Black Pearl Knowledge Broker to perform a variety of tasks:

User	Tasks	Important Chapters
System Integrators Deployment Specialists	Install and integrate the Black Pearl Knowledge Broker within an enterprise	• Chapters 1, 4
Programmers	Code the control flow and application logic	• Chapters 1, 2, 3, 4
Knowledge Engineers	Code the business logic	• Chapters 2, 4

System integrators and deployment specialists reading this guide should have at least some knowledge of the Java 2 Enterprise Edition platform architecture, or a similar distributed, middleware architecture system. Programmers should have knowledge of Java and Enterprise JavaBeans, XML, and some knowledge of message-oriented middleware (MOM) systems.

Document Conventions

This guide uses a variety of formats to identify different types of information.

Convention	Function
`courier`	Identifies syntax statements, on-screen computer text, and path, file, drive, directory, database, and table names.
`<courier>`	Identifies variable names.
`bold courier`	Identifies text you must type.
italics	Identifies document and chapter titles, special words or phrases used for the first time, and words of emphasis.
underline	Identifies URLs, domain names, and email addresses.
Initial Caps	Identifies Window, menu, command, button, option, tab, keyboard, and product-specific names.
ALL CAPS	Identifies acronyms and abbreviations.
[ ]	Identifies an optional item in syntax statements.
{ }	Identifies an optional item that can be repeated as necessary within a syntax statement.
>	Identifies a separation between a menu and an option.
|	Identifies a separation between items in a list of unique keywords when you may only specify one keyword.

Special Message Conventions

Identifies information that will help prevent equipment failure or loss of data.

Identifies information of importance or special interest, including Notes and Tips.

Menu Conventions

This guide uses the Menu > Option convention. For example, "Click Format > Style" is a shorthand instruction for "Click the Format menu, then select the Style option."

Additional Help

For additional information or advice, contact:

Contact Information

Application	Online Help and Readme text file
Phone	(415) 357-8300
Facsimile	(415) 357-8399
Internet	http://www.blackpearl.com
Email	sales@blackpearl.com techsupport@blackpearl.com techpubs@blackpearl.com
Postal	Black Pearl, Inc. 400 Second Street, Suite 450 San Francisco, CA 94107

Chapter 1

Understanding the Knowledge Broker

This chapter introduces the Black Pearl Knowledge Broker platform architecture. Through integrating a distributed systems approach, intelligent agents, an ontology or "data representation layer", and rules processing, the Knowledge Broker enables the collective expertise of enterprises to be brought to bear in electronic markets to create more intelligent transactions.

Application View of the Knowledge Broker

The Knowledge Broker can be seen by external applications as an inferencing and recommendation service. External applications and services send requests to the Knowledge Broker and receive responses. This is a synchronous mode of operation. External applications and services can also communicate with the Knowledge Broker asynchronously, as either publishers of or subscribers to the Knowledge Broker's information services.

The Knowledge Broker gathers input data from enterprise information resources (EIS) such as database systems (RDBMS), structured data files such as XML by making query-like requests. The Knowledge Broker's architecture enables external applications to be abstracted into information resources. Thus, the Knowledge Broker can interact with message-oriented middleware (MOM) systems and their message streams such as RV or MQ.

The Knowledge Broker's services execute on a server, within a *container* in the Java 2 Enterprise (J2EE) sense of an application server that provides the following services in an implementation-specific way:

- Transaction Management
- Security Management
- Connection Pooling

In the J2EE Connector Architecture (JCA), the container communicates with a resource adaptor using these system-level contracts. The resource adaptor then provides a platform- and language-neutral bridge between the application components and the various EIS using, respectively, application-level contracts and EIS-specific interfaces.

Additionally, application components communicate with the container using container-component contracts.

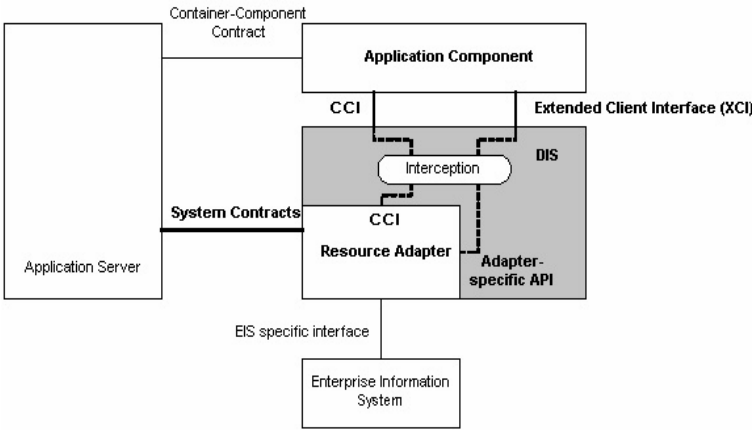
Figure 1-1. Overview of the Connector Architecture

You can learn more about the J2EE Connector Architecture from the Javasoft site at:
http://www.javasoft.com/j2ee/white/connector.html
http://java.sun.com/j2ee/download.html#connectorspec

The application contracts form the Application Programming Interfaces (APIs) that lie between the resource adapter and the application components; these APIs define a Common Client Interface (CCI). The Knowledge Broker uses the Extended Client Interface (XCI) J2EE architecture specification to transparently intercept calls between the resource adaptor and external application components. The XCI is the gateway to the

Knowledge Broker's inferencing and data federation services that extend enterprise systems' functionality far above that available from most J2EE and middleware vendors.

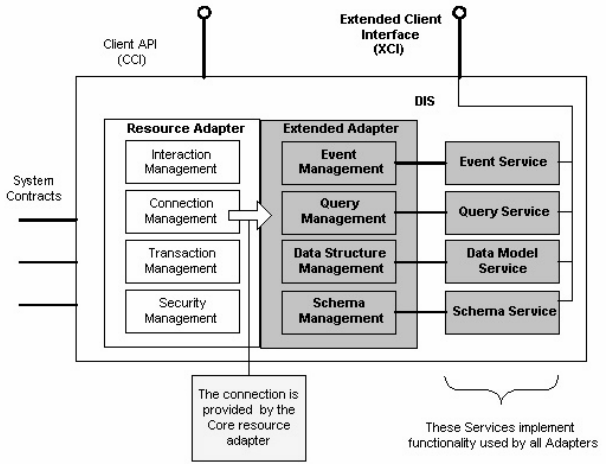
Figure 1-2. Overview of the Connector Architecture



By intercepting these calls (within its Data Information Service, or DIS) and transparently routing them to its own execution services, the Knowledge Broker can add its own unique services. These are:

- Event Service
- Query Service
- Data Model Service
- Schema Service

Figure 1-3. Extended Client Interface



The Knowledge Broker is not, however, tied to the J2EE architecture. A basic execution environment enables the Knowledge Broker to provide its infrastructure services across a message bus such as TIB/RV.

The Knowledge Broker provides for rapid application modeling and deployment by taking a descriptive approach to application development. The application model uses descriptive data files (called *descriptors*) to initialize the infrastructure "plumbing" that provide the services and application logic. In many cases, these descriptors take the form of ready-made templates. Application development for specific configurations of EIS or application components thus often requires that a deployment editor inputs deployment parameters. The difficulties of connecting to diverse EIS are encapsulated behind these descriptors, and application development proceeds rapidly.

Separately, a model-based approach uses a different level of descriptors to define the business logic and inferencing services. Thus, business analysts/domain experts/ knowledge engineers can concentrate on defining and refining the logical model. this parallel mode of solution development and deployment is flexible and fast.

System Components

This section outlines the Knowledge Broker's various functional components from an application-centric point-of-view. There is a deeper, connector-based view where the details of the container-connector contract and APIs are not encapsulated behind descriptors but instead rely on developers' familiarity with JCX. However, the vast majority of Knowledge Broker application development and deployment can be accomplished through descriptor authoring and configuration.

The eight main components of the Knowledge Broker are the:

- Container
- Application Components
- Descriptor Components
- Object Factories
- Reasoning Services
- Query and Transformation Service (DISQE)
- Connectors
- Administration Service

Container

The *container* hosts all application components, descriptor components, and services. It creates and registers services and components and, on request, locates these objects for the requestor. This is called *hosting*. Containers generally pool these resource in an efficient manner and balance access to them.

The container also manages execution threads for components and services, and dispatches requests that come from external processes through a connector.

A Unique Resource Identifier (URI) uniquely identifies every component with a name. This mechanism enables multiple containers to host components in a distributed fashion.

Sophisticated containers also manage sessions and private state. This means that a request to locate a service or component will search within the requestor's current session context. This provides transactional isolation between sessions and transactions. Transparent fail-over and load balancing are advanced container options and depend greatly on container implementation and deployment.

From the application's point-of-view, the container offers at least the following functions:

- Named access to descriptor components. The container returns a handle that implements the component interface; calls to the implementation object can thus be transparently marshalled.
- Configures the dispatch requests to application components.

Not all containers provide transparent remote method invocation (RMI). The descriptor components have proxy counterparts that organize the implementation to the counterpart objects. A registered application component must satisfy at least one of these three conditions:

- Be local.
- Provide a local proxy that marshals requests to the original component.
- Be registered within a container that offers remote location and invocation services.

Currently, the Knowledge Broker supports one container:

- TIB/RV Container

TIB/Rendezvous Container

The *TIB/RV Container* executes components based on the arrival of TIB/RV messages. One or more application components register interest in specific messages using one or more interaction descriptors. These descriptors encapsulate and configure communications with the TIB/RV connector. From the TIB/RV point-of-view, the application component is a message consumer.

Application Components

Application components implement the basic control flow of a Knowledge Broker application. The set of descriptor components remove much of the basic implementation work from the application components. These lightweight application components contain business logic that cannot be expressed using descriptors or that cannot be provided (inferred) using the Knowledge Broker inferencing services.

Descriptor Components

Descriptor components (also simply named *descriptors*) offload most of the infrastructure code from the application components. The fact that descriptors are regular JavaBeans (that implement interfaces) shields the descriptor user from specific implementations. A descriptor bean could be located on a remote server or loaded by a specific container, but this is transparent to the user.

Descriptors encapsulate within their bean properties most of the parameters usually considered to be environment settings. The application code, for example, rarely sets connection URLs or the descriptions of message content types; this role is left to the descriptors. JavaBean properties can be set using a variety of graphical tools. An application typically accesses a fully configured descriptor because of this persistent descriptor storage.

Applications use descriptors mainly to invoke artificial intelligence (AI) services such as the inference engines. They also use descriptors to provide a high-level interface to drive the low-level connector API. Applications do not need to be aware of these low-level details, or of how to obtain and set the configuration parameters. Instead, applications use qualified names to reference the descriptors loaded by the container. There is flexibility: applications can choose to overwrite the properties that were loaded from the model storage.

There are eight main descriptors:

- Connector Descriptor
- Connection Descriptor
- Interaction Descriptor
- Message Descriptor
- Schema Descriptor
- Mapping Descriptor
- Ontology Descriptor
- Inference Engine Descriptor

Connector Descriptor

The *connector descriptor* encapsulates the parameters required to manage connectors. Typically, an application does not use connector descriptors unless it requires access to the low-level connector API.

There are four key connectors:

- TIBCO
- RDBMS
- WWW
- File

The JavaBean properties used by the connector descriptor include but are not limited to:

Table 1-1. Connector Descriptor JavaBean Properties

Property	Description
Name	The prefix of the connection URLs handled by the connector. For example, the name of the connector to the TIB/RV bus is `tibrv`, while for DB2 the connector's name is `jdbc:db2`.
Connection URL Strings	The names of the properties required to form a connection URL. For example, the Oracle type-4 JDBC connector requires all of these to form a connection string: TCP server name, port, and SID.
System-level implementation class names	These are the names of the classes that implement the system level interfaces. These load dynamically and enable new connectors to be plugged-in.

Connection Descriptor

The *connection descriptor* encapsulates the parameters that are required to manage connections. Typically, an application does not use connection descriptors unless it requires access to the low-level connector API.

The JavaBean properties used by the connection descriptor include but are not limited to:

Table 1-2. Connection Descriptor JavaBean Properties

Property	Description
Name	Uniquely identifies the connection descriptor.
Connector name	Uniquely identifies the connector descriptor.
Connection properties	These are the names of the classes that implement the system level interfaces. These load dynamically and enable new connectors to be plugged-in.
Login string	The user name and password response to a challenge/response connector.
Schema connection descriptor	The name of the connection descriptor that stores the schema. Some connections do not provide schema information. In this case, after assertion to a connection, the schema information is retrieved from a schema repository.

Interaction Descriptor

The *interaction descriptor* encapsulates the information required to use an external resource (that is, external to the Knowledge Broker). Database queries are one example of an external resource interaction. Another is an external application invocation where the application integrates with the Knowledge Broker through the connector framework.

There are four types of interactions:

Table 1-3. Types of Interactions

Property	Description
Synchronous request/response	These are similar to method calls or external resource queries. The outgoing part contains the parameters while the incoming part contains the results.
Synchronous one-way	These pass information to an external resource. Storing information in a database or publishing information to a message queue are good examples.
Notifications	These consume information that an external resource has asynchronously produced.
Solicit/Response	These serve external requests that require a response. The Knowledge Broker application offers to serve these requests. An application component invocation initiated by an external application or client (say, a request to invoke a Servlet and return a response) is a typical example.

Interactions generally bundle one or two abstract messages. Even queries are modeled by a pair of messages; the outgoing message contains the parameters while the incoming message contains the results.

The four interactions use the utility interfaces listed in Table 1-4. These interfaces describe how both discrete and document data can be passed to and from interactions:

Table 1-4. Utility Interfaces

Interface	Description
`DataInput`	The `DataInput` interface reads data from the implementer. The actor implementing the `DataInput` interface plays the role of the data producer, that is, the side from where the input comes. The actor using the `DataInput` interface plays the role of the data consumer, that is, the side to where the input goes. It pulls the data from the data source (Cursor).
`DataOutput`	The `DataOutput` interface writes data to the implementer. The actor implementing the `DataOutput` interface plays the role of the data consumer, that is, the side to where the output goes. It pulls the data from the source (Cursor). The actor using the `DataOutput` interface plays the role of the data producer, the side from where the output comes.
`DocumentInput`	The `DocumentInput` interface reads data from the implementer. The actor implementing the `DocumentInput` interface plays the role of the data producer, that is, the side from where the input comes. It sends the data to the Content Handler. The actor using the `DocumentInput` interface is responsible for connecting the listener actor, also called the data consumer. This is the side to where the output goes. The data consumer (ContentHandler) receives the document data. It can also assume the role of the data producer itself.
`DocumentOutput`	The `DocumentOutput` interface writes data to the implementer. The actor implementing the `DocumentOutput` interface plays the role of the data consumer, that is, the side to where the output goes. It receives the data via the ContentHandler 'sink'. The actor using the `DocumentOutput` interface plays the role of the data producer, that is, the side from where the output comes. It pushes the data into the sink (the ContentHandler).

The properties used by the interaction descriptor greatly depend on which connector handles the interaction. The JavaBean properties used by the interaction descriptor include but are not limited to:

Table 1-5. Interaction Descriptor JavaBean Properties

Property	Description
Name	Uniquely identifies the interaction descriptor.
Connection name	Uniquely identifies the connection descriptor (and therefore the connection that performs the interaction).
Timeout	The length of time the interaction will wait.
Connector-specific parameters	These include parameters such as a query string, or a subject message and filter.

Message Descriptor

The *message descriptor* describes an incoming or outgoing message. It is tied to both an interaction and a type descriptor (that describes the data of the message content).

Message descriptors access data in a generic format, considered to be a low-level interface. The object factory sub-system constructs specific instances of Java classes to perform the generic data access. There are two main styles of data access: a *cursor* and a *content handler*. An application uses a cursor to iterate over a complex, tree-like data structure. By comparison, an application using a content handler implements callbacks to iterate over the data structure. The Knowledge Broker content handler is a regular Simple API for XML 2.0 (SAX). XML applications usually consume data using content handlers and many applications implement SAX handlers.

For more information about SAX, see
`http://www.megginson.com/SAX/index.html`

The JavaBean properties used by the message descriptor include but are not limited to:

Table 1-6. Message Descriptor JavaBean Properties

Property	Description
Name	Uniquely identifies the interaction descriptor.
Connection name	Uniquely identifies the connection descriptor.
Type	The type of the content.
Optional recipient	Some messages require specific routing information in addition to the information available through the interaction or the connection.

Schema Descriptor

The *schema descriptor* (also called a schema) is a collection of type descriptors. A resource adaptor stores the schema externally. Some resource adapters, such as relational databases, store the schema information with the primary data whereas others require a schema that is asserted to the connection. The schema associated with the interaction connection contains the interaction message types.

Schema information can be externalized using the XML Schema Description Language (XSDL).

The JavaBean properties used by the schema descriptor include but are not limited to:

Table 1-7. Schema Descriptor JavaBean Properties

Property	Description
Name	Uniquely identifies the schema descriptor.
Version	Uniquely identifies the schema version.
List of types	A list of the schema types.

Mapping Descriptor

The *mapping descriptor* specifies how one type data maps (or transforms) to another data type. This descriptor can also define heterogeneous joins of multiple data sources. Applications use mapping descriptors to specify to the DIS Query and Transformation Engine (DISQE) how complex information assets should be assembled or transformed. Finally, the mapping descriptor enables filter and view specification.

Mapping descriptors can be externalized using the XML Query Language (XQuery).

For more details about XQuery, see
`http://www.w3.org/TR/xquery/`

A mapping descriptor associated with an interaction descriptor forms a complex query. When selecting which descriptor to use in complex queries, the mapping descriptor takes precedence over the interaction descriptor.

Ontology Descriptor

The *ontology descriptor* (or ontology) is a collection of concepts and relations between these concepts. Ontologies are a key component of the Knowledge Broker's knowledge base.

The JavaBean properties used by the ontology descriptor include but are not limited to:

Table 1-8. Ontology Descriptor JavaBean Properties

Property	Description
Name	Uniquely identifies the ontology descriptor.
Version	Uniquely identifies the ontology version.
List of Concepts	A list of the concepts and relations.

Inference Engine Descriptor

The *inference engine* descriptor configures an inference service. An application does not have to configure all properties. As with all other descriptor objects, configuration can take place through a graphical user interface (GUI). The GUI configuration output is then stored in the model storage.

The JavaBean properties used by the inference engine descriptor include but are not limited to:

Table 1-9. Inference Engine Descriptor JavaBean Properties

Property	Description
Name	Uniquely identifies the inference engine descriptor.
Ontology name	Uniquely identifies the ontology descriptor.

Object Factories

In addition to descriptors, *object factories* are used when specific classes must be instantiated and filled with data from external interactions.

Object factories do not work only in conjunction with descriptors. They can also be used to gain access to objects already instantiated in memory. The Knowledge Broker is thus able to access objects controlled by other applications. In this scenario, an interaction descriptor identifies only a concrete instance.

Reasoning Services

Reasoning engines provide the core of the Knowledge Broker advanced artificial intelligence (AI) services. These technologies set the Knowledge Broker apart from standard middleware platforms. When combined with the Knowledge Broker's infrastructure services (heterogeneous data access, message-oriented middleware (MOM), and EIS connectivity), AI services can be integrated within enterprise architectures.

To reason, inference engines require access to rule objects and concept/relation objects. Descriptors can conveniently retrieve both types of object. Object factories deserialize XML representations into rules and construct JavaBeans from data received through connectors.

Rete Inference Engine

The *Rete inference engine* is the primary inference engine. It computes a set of atomic formulas created by a set of facts and rules in implicative normal form. The facts are fully instantiated atomic formulas that can be retrieved from a data source or assembled by a program.

The Rete Engine accesses all working memories. The memories consists of two parts: global facts and local evidence. *Global facts* are shared between sessions. *Local evidence* scope is restricted to each rule engine invocation. An `Advisor` Java class wrapper accesses the inferencing engine.

The standard reference for the Rete algorithm is:
C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

Query and Transformation Service

The *Query and Transformation Engine* (DISQE) maps or transforms data structures into other data structures, using the low-level generic data representations: cursor and content handler. Frequently, a connector does not structure information in the appropriate format required by application or object factory. The DISQE assembles new structures out of existing structures.

Administration Service

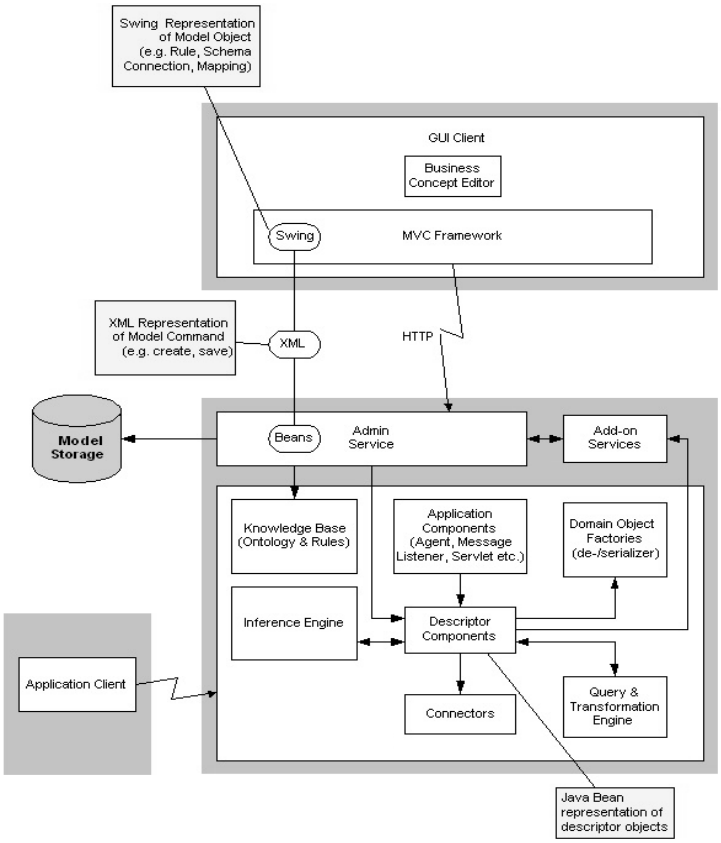
The *Administration Service* (AS) manages the collection of descriptor objects. The collection (which includes the ontology) is called the Application Model.

The AS communicates with the graphical user interface (GUI) used for modeling, deployment and monitoring. It also manages the storage and retrieval of descriptor objects using an internal command language.

The AS uses the object factory subsystem to deserialize a model from the persistent storage format: an XML file. It partitions the XML file into several physical files and transparently links them together. A good analogy here is to visualize how a browser build a single display screen from multiple entities such as text, images, frames, and so on.

Figure 1-4 is a schematic representation of the Knowledge Broker's functional components.

Figure 1-4. Knowledge Broker Functional Components



Chapter 2

Understanding the Ontology

The *ontology* lies at the heart of the Knowledge Broker. It is a data abstraction and modeling approach based not on normalized relational tables but instead on an ontological hierarchy. Application programmers do not interact with the ontology directly, instead using conventional descriptor objects to trigger instantiationnand inferencing across an ontological domain. Nevertheless, it is useful to understand the abstract logical structure that makes the Knowledge Broker's solution so flexible, extensible, and scalable.

Ontology Structure

Why an Ontology?

The Knowledge Broker has three core functions:

1. transforming raw data into information
2. transforming information into knowledge
3. transforming knowledge in a concise, relevant, and active format

The Knowledge Broker's transformation and information exchange services operate using an actor communication model. Several internal Knowledge Broker subcomponents communicate with each other, and may also communicate with external actors (data sources, news feeds, and so on).

At the core of every business model there is a conceptual representation of its domain-specific objects and their relationships with other objects. This model is stored within an ontology: a logical structure that defines relationship of different entities in the modeled domain to each other. Managing and merging multiple ontologies allows the Knowledge Broker to extend its reasoning expertise across many domains.

Ontologies (and similar data structures) have been used extensively through history for knowledge management and exploration. Dictionaries using the lexical relationships embedded within the alphabet are a predecessor of ontologies, while programmers' object-oriented class definitions are a type of strict hierarchical ontology.

An ontology is a set of objects and their relationships, a semantic web of taxonomic information with additional links representing associated properties or attributes. An object can be "real", that is, referring directly to entities or qualities with physical manifestations and capable of operationalization (or reification where data instantiates a generic object description into a concrete instance of that object). Or an object can be abstract, referring to other objects and collections of objects. A relationship is a named set of links between objects, with the most basic relationship being a parent-child linkage. Ontologies can thus be represented as nodes and arcs; this is a graph-theoretical approach.

A domain can be modeled with a set of facts and concepts that are important to the domain and a semantic vocabulary that expresses these concepts. The vocabulary translates important domain-level concepts into logic-level names. An ontology also lists the relationships between these terms.

Using the terms and relationships defined in the ontology, business analysts can encode business logic through writing logical sentences using the supplied vocabulary. These logical statements are *business rules*. These rules combined with the concepts and relationships embedded in the ontology comprise the *knowledge base*. This knowledge

base expresses the logic clearly and in a semantically relevant form compared with coding business logic using application programming semantics. This facilitates direct involvement by domain experts in analysis and problem solving.

Using the rules of inferencing, the Knowledge Broker can also inference across the existing statements and automatically derive new consequences from the knowledge base. These inferences can be acted on immediately, as recommendations, or entered into the knowledge base to produce further inferences.

Concepts

More specifically, an ontology describes a hierarchy of *concepts* related by subsumption relationships; in more sophisticated cases, suitable axioms can be added to express other concepts relationships (also called context). This contextualization also constrains the range of the concepts' intended interpretation.

Relations

Within an ontology, nodes represent concepts. Lines joining nodes ("arcs" or "edges") represent the relationships between concepts. These relationships are typically meronymy and hyperonymy.

- *Meronymous relationships* describe the belonging between a whole and a part. A whole HAS_A part. A part IS_PART_OF a whole.
- *Hyperonymous relationships* describe the equivalence between a concept and a subconcept. A subconcept IS_A concept.

Because Knowledge Broker's models tend to be semantic and lexical (based on English usage), it's valuable to think of how these relationships map into English:

Table 2-1. Relationship Types

Relationship	Applicable To	Examples
Hyperonymy	noun-to-noun verb-to-verb	car/automobile walk/move
Meronymy	noun-to-noun	head/nose

Rules

The business rules within the knowledge base have the following structure:

If *antecedent* then *consequent*

The consequent is composed of an action and a concept. Both the antecedent and the consequent are terms defined in the ontology.

Typically, these rules take the form of:

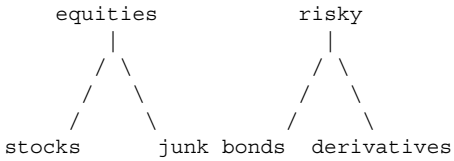
- If (customer) then recommend hot stocks
- If (nervous customer) then sell highly volatile stocks

Encoding the Ontology Using Directed Acyclic Graphs

Many programmers are intimately familiar with the concept of trees, commonly used to indicate relationships between objects, such as files and directories (for example, Windows File Explorer tree view). The Knowledge Broker uses a more complex data structure, known as a Directed Acyclic Graph, or DAG, to represent objects in the ontology.

Tree nodes can have at most one parent. By comparison, DAG nodes can have multiple parents, and more than one root node.

Figure 2-1. A Simple DAG



Every DAG can be represented as a tree. In this representation, a node can be multiply listed as a child node of each of its parents. This DAG can be represented in Figure 2-2.

Figure 2-2. Simple DAG, Converted to a Tree

```
+--equities
|  +--stocks
|  +--junk bonds
+--risky
   +--derivatives
   +--junk bonds
```

An ontology therefore can be ideally represented by a DAG.

A graph structure where all of the relation arrows (or edges) can be ordered to point in the "same" direction is known as *topologically sorted*. Because DAGs can always be topologically sorted using recursive descent (depth-first search or DFS), they are optimal for many computational tasks, such as critical path analysis, expression tree evaluation, and game evaluation. All of these tasks, and more, are necessary for the Knowledge Broker to maximize performance as it accesses complex ontologies. Cyclic graphs cannot be topologically sorted

Figure 2-3. DAG Model, topologically sorted

The DAGs provide the following ontologically related functionality:

- Add a new concept to the graph.
- Delete a term from the graph. All subsequent terms related to this term would also be deleted.
- Change the meaning of a term in the graph.
- Add/Delete/Change an arc, attribute, or concept.
- Add/Change ontological filters.

Examining the Knowledge Broker's DAGs

The Knowledge Broker represents ontologies using the following DAG constructs:

- Nodes
- Arcs
- Direction

A *node* represents a concept in the domain, for example, "hot stocks", "rich customers", "sectors", "internet stocks". A node has certain attributes associated with it that can be evaluated. For a Customer, for example, the attributes could be age, income, risk profile, location, net worth, and so on.

An *arc* represents a relationship between the nodes. The arc is a function or transformation that produces a child node from the parent node. A child can have multiple parents. An example of a transformation is "all customers with income greater than one million dollars". Applying this transformation on the "customers" node will create a new concept that could be named "high net worth customers" (of course, the number of instances of this new concept will be zero or null if no customers' incomes satisfy the transformation condition).

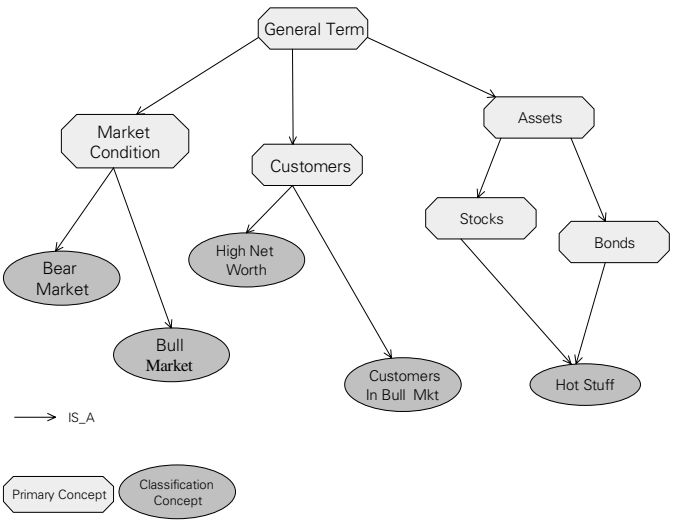
The *direction* of the arc provides the relationship between the parent and the child nodes.

The Knowledge Broker's DAG representations exhibit some other features:

- There are two types of nodes. The first is a *primary concept node* or *extent* (for example, "Customer", "Assets", and so on) and corresponds to Data Concepts associated in the Data Concept Editor with datasource items can be retrieved from an external data source. The second type of node is a *classification concept* (for example, "Hot Stuff", "Rich Customer") and corresponds to Secondary Business Concepts defined in the Business Concept Editor.
- Arcs represent an IS_A relationship between the parent and the child. For example, hot stock IS_A Stock. Other arcs, while supported, are not currently implemented.
- The DAG captures the inter-relationships between the terms. Every node in the DAG evaluates to specific instances and these have to be mapped to their data sources.

The following diagram illustrates a simple ontology. "Stocks", "Market Condition", and "Customers" are three primary concepts within a domain.

Figure 2-4. Simple Ontology Diagram



Suppose a business analyst entered the following business rule:

```
If (High Net Worth Customer) then recommend hot stocks
```

At inference time, the Knowledge Broker will read in data from external data sources to create an instance of Customer, perform a test to determine whether the customer is high net worth, and generate a hot stock recommendation if the test is successful.

Appreciating the Ontology Advantage

It is possible to construct advice and personalization systems using rules-based syntax but omitting any ontological representation. Unfortunately, these systems lack transparency, do not scale well with increasing numbers of rules, and are brittle when assumptions must be changed.

For example, a typical non-ontology ecommerce rule might resemble something like a sequence of simple condition-action pairs such as:

```
If Customer lives in California &&
They are 18-30 years of age &&
They selected a backpack
Then
Recommend tents with a priority of 10
```

These rules may have been derived from simple data-mining, or through business analysis. Entering a large business rules is going to be tedious, inflexible, non-intuitive, and error-prone.

By contrast, the Knowledge Broker's ontological-based approach uses a simple ontology to express this rule. This ontology contains "Customers", "Retail Goods", "Back Pack", and "Tents" as primary concepts (derived from data sources). The business analyst can define a new concept, "Young California Backpacker", which are those customers that live in California, are between the ages of 18 and 30, and have selected "Backpacks" in their shopping carts.

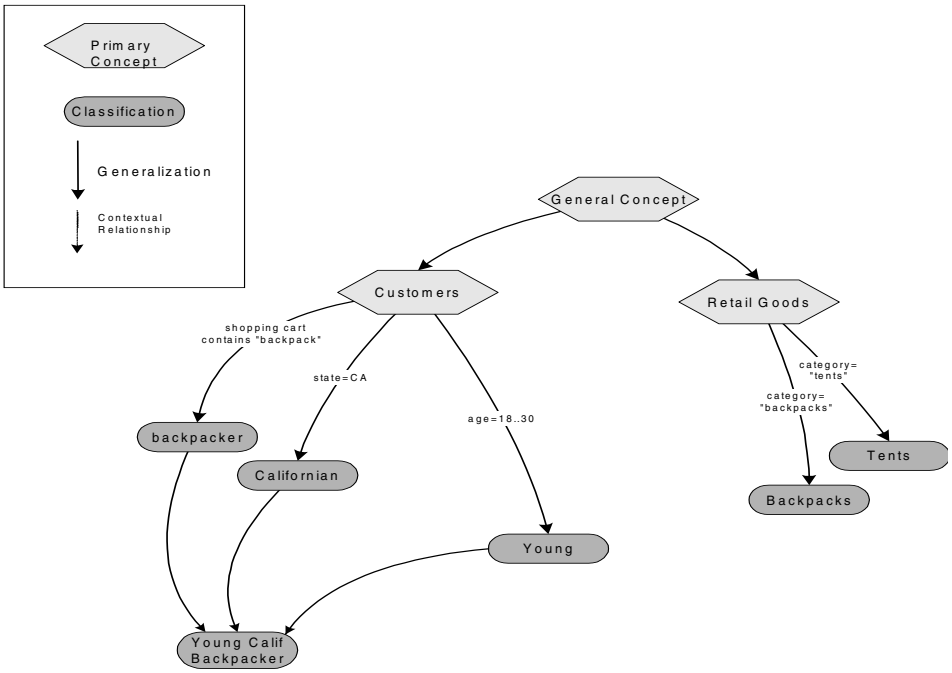
Then the same business rule can be stated as:

```
If  "Young California Backpacker"
Then
Recommend "Tents" with a priority of 10
```

By abstracting the business logic into the ontological domain, the system can express statements about the domain in semantically precise language suitable for consumption by domain experts without further post-processing or analysis.

Figure 2-5 shows the DAG for this ontology.

Figure 2-5. DAG Ontology



Another advantage of the ontological system lies in its flexibility. Suppose a business requirement arose where the policy was now to recommend tents to customers living in New York, or example, instead of California.

In the non-ontological system, every business rule would have to be scanned for the presence of the text symbols or codes corresponding to "California", the rule's context checked, and the necessary substitution made. This process is tedious and vulnerable to errors and unforeseen interactions. This same change can be made by simply entering the ontology editor (the Business Concept Editor) and changing the expression for the arc from "Customer" to "Young California Backpacker" to location being "New York" and changing the name "Young California Backpacker" to "Young New Yorker Backpacker".

Finding the Database Relation

One further advantage of the ontology is that you can easily query the system for instances corresponding to each of the concepts. This can provide useful population and percentage information and is similar to querying a relational database, but with certain unique characteristics.

Relational databases, with their normalized forms, schemas, and associated entity-relationship models, also represent ontologies. However, they are more limited for knowledge and semantic management than ontologies (where Knowledge Broker excels), because they are typically limited to the abstractions represented in the databases. Context and consistency have to be applied using external maps, and most schemas lack explicit enumerations of the values or domain definitions. Formulating queries implies too much knowledge of the underlying data structure. Ontologies are a way to group database records in a semantically meaningful way. Semantic grouping makes it trivial to retrieve a set of records that are associated with a particular concept; this is analogous to the way b-trees allow the retrieval of a range of data.

Much information (residing in databases or XML files or structured messages) takes the form of attribute-value *tuples*. The Knowledge Broker's ontology contains concepts, where each concept has a name or ID, a list of sub-concepts, and a list of attributes. These concept node objects feature "empty" attribute values. Retrieving values from the underlying data sources and inserting these values into the attribute "slots" creates object node instances.

Traversing an ontology therefore can involve several distinct operations:

- Retrieve any or all direct instances of a concept.
- Retrieve any or all sub-concepts of a concept.
- Retrieve any or all transitive sub-concepts, that is, any or all sub-concepts reachable through following any or all possible directed links.

Traversing the Ontology

The Knowledge Broker DIS handles the underlying data access, marshalling, and federation necessary to map external data to the internal logical ontology representation. In the Knowledge Broker, you communicate with the DIS using cursors. *Cursors* are objects that are used to traverse abstract data structures. They return result sets that form a strict hierarchy with traversal rules.

The DIS objects are traversed in a normalized order that is equivalent to a recursive descent (that is, depth-first searching or DFS). Thus, a Cursor traverses the object's children (in a preset order) before it traverses any sibling (the next child of the parent of the current object).

This maps well to the topological sorting properties of DAGs. Using recursive descent, looking at a previously-examined node would imply that there was a cycle in the graph. But this is impossible because the Knowledge Broker represents the ontology as directed and acyclic graphs.

Specialized cursors insert and extract objects from the ontology and rulebase. You do not interact directly with the ontology but instead access it through the Data Management Objects of the DIS.

Serializing the Ontology

The ontology can be serialized and written out to external storage. This is implementation-dependent. The ontology can be stored as an XML file, on a network as a URL, or within a suitable RDBMS (using the appropriate resource adaptor).

Chapter 3

Understanding the DIS

The Distributed Information System (DIS) is the core data federation and transaction and query processing Knowledge Broker subsystem. It orchestrates and enables communication between all the subsystems, and with external applications. A general understanding of the DIS architecture is valuable, although the Descriptor approach described in the following chapter "shields" application component developers from much of the underlying DIS complexities while facilitating rapid application development and deployment.

DIS Architecture

The DIS is the "glue" that binds the logical ontological models contained within the Knowledge Broker to data sources in the external world. More formally, the DIS provides connectivity to heterogeneous data sources. It adds value to the Java 2 Platform, Enterprise Edition (J2EE) by offering extended functionality not found in the standard Java 2 Connector Architecture (JCX) or in freely available resource adapters. The DIS follows the principal goal of the connector architecture by presenting a uniform API enabling access to heterogeneous information systems.

The J2EE platform additionally specifies a way to extend the containers by accessing external Enterprise Information Systems (EIS). EISs range from relational databases over messaging systems to packaged back-office applications (for example, Enterprise Resource Platform, or ERP, systems). The part that specifies how the J2EE platform provides connectivity to EIS is called J2EE Connector Architecture (JCX).

Overview

The Knowledge Broker is built on the J2EE platform. It adds a number of services to the platform using the DIS component to provide connectivity to heterogeneous data sources. The DIS adds value to J2EE by offering extended functionality absent from the standard connector architecture or freely available resource adapters.

This document describes the JCX architecture in detail:
`http://java.sun.com/j2ee/download.html`

The J2EE specification prescribes a single uniform procedure to access enterprise information systems. The purpose of the connector architecture is that implementation is not tied to a specific application server or vendor but instead is applicable to all J2EE platform-compliant application servers from multiple vendors.

The DIS architecture allows further leveraging of development effort by using existing J2EE standard connectors. The DIS extends the JCX architecture in three areas:

- Richer interaction support
- Mapping support
- Federation

Richer interaction support

The DIS enhances JCX client support with four new, rich interactions:

- Schema Support
- XML Support
- Java Object Support
- Unified Address Support

Schema Support

In a standard J2EE platform, a client application cannot programmatically discover schema information contained within an EIS. By contrast, the DIS offers a schema discovery API. Additionally, it offers a uniform API for creating a schema in an external system (if the system supports a native API to accomplish this task). Relational database management systems (RDBMS), for example, often offer proprietary schema creation and manipulation mechanism.

XML Support

Enterprise Application Integration (EAI) has moved to an XML-based approach. The integration of enterprise-wide information flows demands XML support, while some external systems, such as e-commerce exchanges, offer only XML-based interfaces. Applications and components requiring connectivity are often XML-based as well. The DIS offers XML client support, such as Simple API for XML 2.0 (SAX) and Document Object Model (DOM).

For more information about SAX, see
`http://www.megginson.com/SAX/index.html`

For more information about DOM, see
`http://www.w3.org/DOM/`

Java Object Support

The DIS expands the JCX Common Client Interface (CCI) support for retrieving objects from and passing objects to connected systems. In standard JCX, an application must obtain schema information through APIs proprietary to the external system. Even with this knowledge, there is no standard mechanism to enable application navigation through complex objects. The returned object passed to the EIS is a record, that is, a map, list, or a JDBC result set. Standard JCX provides no further support for the objects contained within these "top-level containers". By contrast, the DIS provides uniform access to complex Java objects using a JavaBean-style API. This access is built on the schema support.

Unified Address Support

The DIS defines a mechanism that enables clients to address resources in a uniform way. It offers a simple unified addressing and querying mechanism based on an extended URI syntax. This is built on support for XPath.

For more information about XPath, see
`http://www.w3.org/TR/xpath`

Mapping support

The DIS offers support to client applications to enhance JCX with three complex mapping functions:

- User-Defined Name Mappings
- Object Structure Mappings
- Mapped Object Structure Queries

User-Defined Name Mappings

The DIS offers support for user-defined name mappings. It creates a logical schema that can use different names as identifiers; it can also form different types based on other types.

Object Structure Mapping

The DIS offers support for application-defined object structure mappings. Instead of relying on the external storage's underlying representation, the DIS creates a logical object that can use different structures for object properties and different types for the values of the properties. The logical schema drives the created object structure.

Mapped Object Structure Queries

The DIS offers support for querying mapped object structures. These involve resolving type and name mappings.

Federation

A *federated* view is also called a logical view, in contrast with a physical or external view controlled entirely by a single EIS. Physical views are like isolated silos that stand side by side. In JCX, the application developer assumes responsibility for their integration. The DIS provides three federation functions to create transparent integration:

- Logical-to-External Schema Mapping
- Heterogeneous Composition of Objects
- Logical Query Decomposition

Logical-to-External Schema Mapping

The DIS provides a number of mappings between physical and logical schemas. It also enables the mapping of particles from multiple physical schemas onto a single particle in the integrated logical schema.

Heterogeneous Composition of Objects

The DIS offers support for object composition using particles from multiple external systems. It enables the composition of objects with properties from multiple, heterogeneous external systems.

Logical Query Decomposition

The DIS offers support for the mapping of queries to multiple sub-queries, where the results are then composed to form a logical query result. The DIS component, the Query and Transformation Engine (DISQE) is a powerful query engine and can accomplish this.

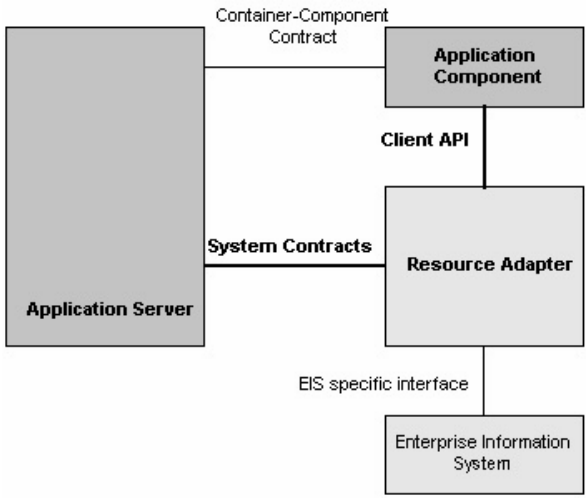
General Architecture

The primary rationale for the JCX architecture is that the application server and the various EISs collaborate to keep system-level services transparent to the application components. This frees the application developer to concentrate on business and presentation logic.

As described in *Roles and Connections on page 37*, Resource Adaptors (RAs) specific to each EIS broker the information flows between the EIS and the application components, and between the EIS and the application server. The system contract encapsulates three main management functions: connections, transactions, and security.

EIS access for the application components comes through a a client API called the Common Client Interface (CCI). Additionally, client APIs may be specific to a RA and its underlying DIS. Java Database Connectivity 2.0 (JDBC) is an example of such an API.

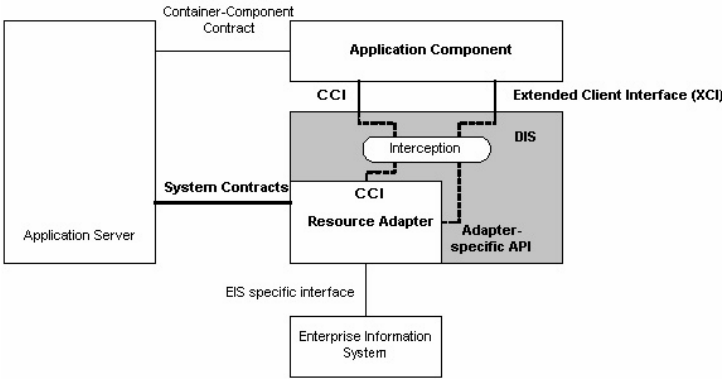
Figure 3-1. Overview of the Connector Architecture



Extended Client API

Using the JCX Extended Client Interface (XCI) standard to add functionality, the DIS does not replace existing resource adapters but rather wraps them and provides additional APIs. To the application component, the DIS appears to be a standard connector, but a connector to internal as opposed to external systems. The DIS delegates all functionality regarding connection, transaction and security management to the EIS-specific connector. This indirection enables mapping and federation support through intercepting the application component's CCI calls to the physical RAs. Figure 3-2 illustrates the role of the DIS regarding the connector architecture.

Figure 3-2. DIS Architecture



The DIS's extended API enables the management functionality described in Table 3-1.

Table 3-1. Management Functionality

Management Functionality	Description
Schema	Applications require metadata access concerning the data flowing to and from the EIS. The Schema Management API is based on the XML Schema standard. This W3C standard defines a rich metadata model that can convert all Java data structures into XML format.
Data Structure	Although CCI does support uniform access to the parameters and results of an EIS interaction, it does not prescribe how to navigate complex structures, such as trees. The DIS APIs for data structure access support two application styles: Java Beans-centric and XML document-centric.
Query	The number of items managed by an EIS is potentially huge. Without query language support, application components require adapter-specific query operation encodings. The DIS provides a uniform access mechanism based on a combination of URI and XPath.
Event	Not all EISs play the role of a passive information resource. Many external systems emit events that application components consume. For some external systems this is the primary usage model, such as a stock ticker where components register interest specific price change events. The optimal model here is a publish/subscribe pattern. The DIS's event management APIs enable a component or an agent to register event listeners.

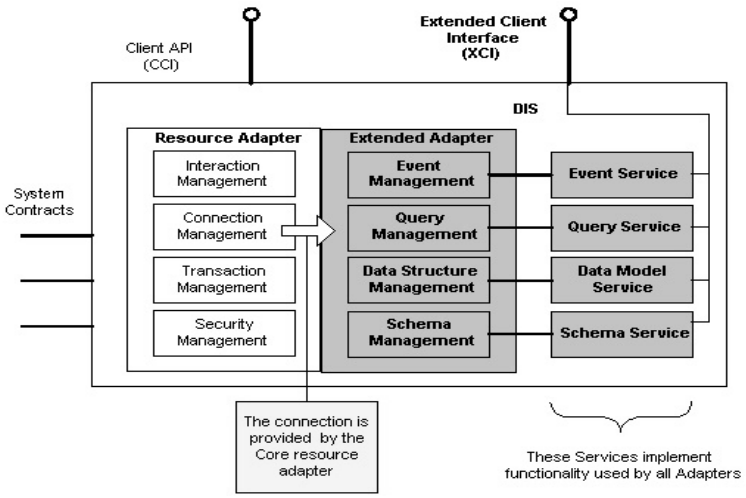
For more information about XML Schema, see
`http://www.w3.org/XML/Schema`

For more information about XPath, see
`http://www.w3.org/TR/xpath`

For more information about URI, see
`http://www.w3.org/Addressing/`

Each of the DIS APIs in the RA corresponds with a generic JCX service that contains the functionality, with defined contracts between the RA and each of the services Figure 3-3 illustrates the DIS architecture for the XCI:

Figure 3-3. Extended Client Interface



Roles and Connections

The J2EE JCX architecture uses roles to describe the responsibilities of different actors during solution development.

Connector Providers supply a resource adaptor specific to an EIS.

Container Providers supply a container implementation for a specific application component type with the Enterprise JavaBeans (EJB) specification serving as the component contract. Application Server Vendors (ASVs) are usually also container providers that provide a J2EE-compliant application server that supports component-based enterprise applications. Container providers usually also supply resource adaptor and application component deployment tools, as well as management utilities.

Application Component Providers (ACPs) create components that access one or more EIS to perform business tasks. ACPs typically use functions provided by the CCI and any ECX APIs; they are not expected to require systems-level knowledge. ACPs specify external dependencies and structural information of components. They package their components in Java Archives (JARs) for deployment.

Enterprise Tools Vendors provide utilities to simplify application development and EIS integration.

Application Assemblers combine application components JARs into more complex JARs with deployment descriptors. Often domain experts, their assembled output can be deployed as an enterprise application.

Deployers take the deployable components (that is, JARs with deployment descriptors) and integrate them into an operation enterprise environment using deployment tools.

System Administrators configure and administer the completed enterprise system. They often work closely with Deployers.

Black Pearl's role within JCX is that of an Extended Connector Provider (ECP). An application component accesses connections instances using a connection factory; these connection instances communicate with the EIS. The connection factory supports the connection management contract that provides for connection pooling.

Connection Factories and JDBC

Using a standard architected contract for connections between the application servers and the resource adaptors ensures that the connector architecture is efficient, scalable, and extensible. The JCX connection factories usually implement the `javax.resource.cci.ConnectionFactory` interface, while JCX connections usually implement the `javax.resource.Connection` interface. In fact, any class can be used as a factory provided it implements a method compatible with this signature:

```
java.lang.Object getConnection() throws Exception
```

This method returns the connection object. By convention, this object implements a method:

```
void close() throws Exception
```

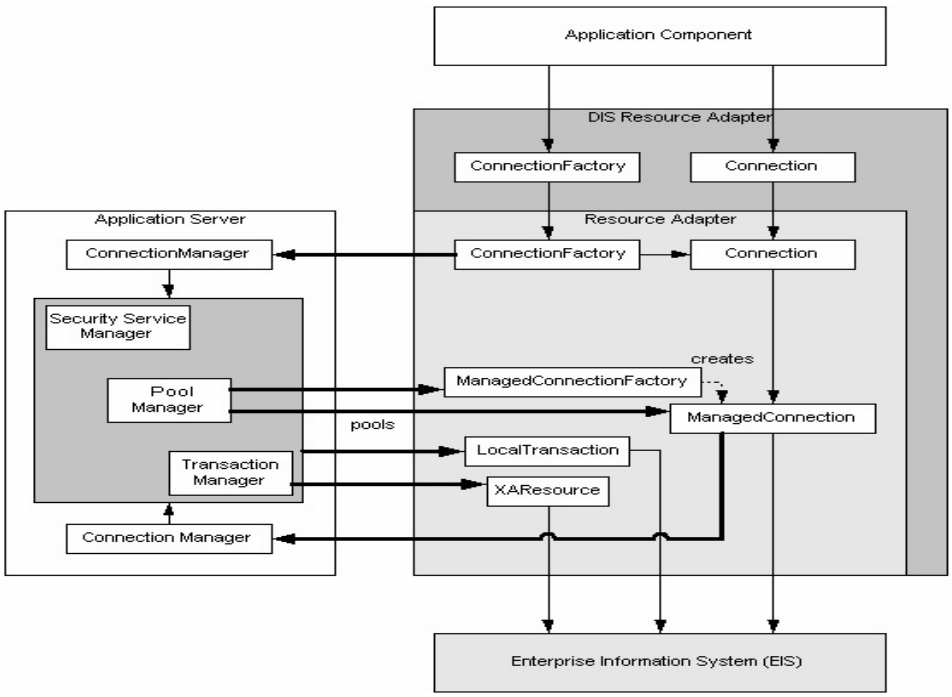
This convention allows standard JDBC connections to be treated as JCX-compliant connections.

The DIS architecture diagram differs slightly from the JCX 1.0 specification because the application components interact with a DIS-implemented `ConnectionFactory`. To the application component, the DIS `ConnectionFactory` provides the `Connection`. The DIS implements the `Connection` object returned to the application component. The DIS accomplishes this by providing an API that implements both the CCI and the XCI.

Both the DIS `ConnectionFactory` and the DIS `Connection` intercept application component requests and delegate them in a modified form to the standard resource adapter's `ConnectionFactory` and `Connection`. The DIS makes no further alterations to the JCX 1.0 architecture. However, this interception technique enables implementation for all EISs of the `javax.resource.cci.ConnectionFactory` and `javax.resource.Connection` interfaces. Application component code does not have to account for alternative method signatures and becomes simplified.

In an application server-managed scenario, the DIS-enhanced connection architecture looks like that in Figure 3-4.

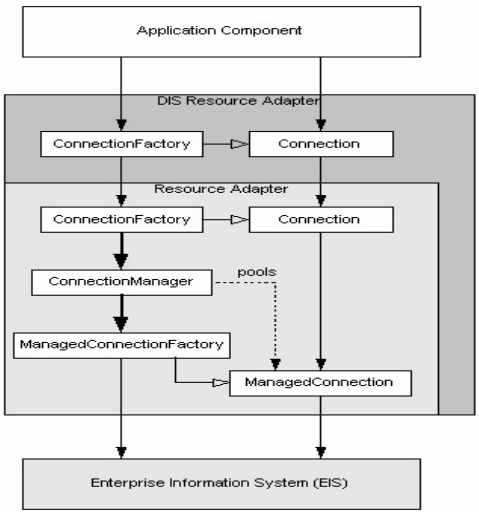
Figure 3-4. Connection Architecture (Managed Scenario)



The non-managed scenario connection architecture uses the same delegation mechanisms. The DIS does not know whether the resource adapter `ConnectionFactory` uses its own `ConnectionManager` or the application server-

provided object. In a non-application server-managed scenario, the DIS-enhanced connection architecture looks like Figure 3-5.

Figure 3-5. Connection Architecture (Non-Managed Scenario)



You can find out more about JCX connection scenarios in section 5.8 of the JCX 1.0 specification:
http://java.sun.com/j2ee/download.html#connectorspec

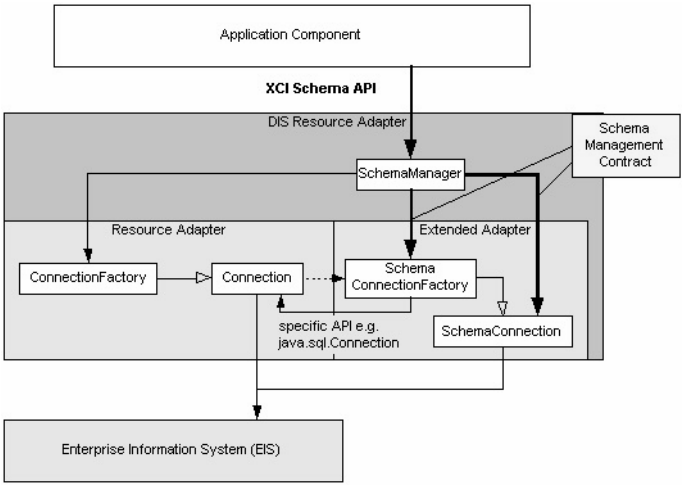
Schema Management

Schema management is one of the key DIS functionality enhancements to the baseline JCX specification. The XML Schema conceptual model specification forms the core of the Knowledge Broker's Schema Management system.

The DIS implements a unified client interface for schema access by using a delegation mechanism. The `SchemaConnection` interface (and some schema component interfaces) provide the XCI's schema "glue". The DIS-returned `Connection` (that is, the logical connection) itself implements the `SchemaConnection`. The resource adapter, however, factors out the `SchemaConnection` interface from the `Connection` interface. The extended resource adapter implements the `SchemaConnection` while the standard resource adapter implements the `Connection`. This enables schema management

support to be implemented on any connection provided by a different vendor that does not implement the XCI. Figure 3-6 outlines the schema management architecture:

Figure 3-6. Schema Management Architecture

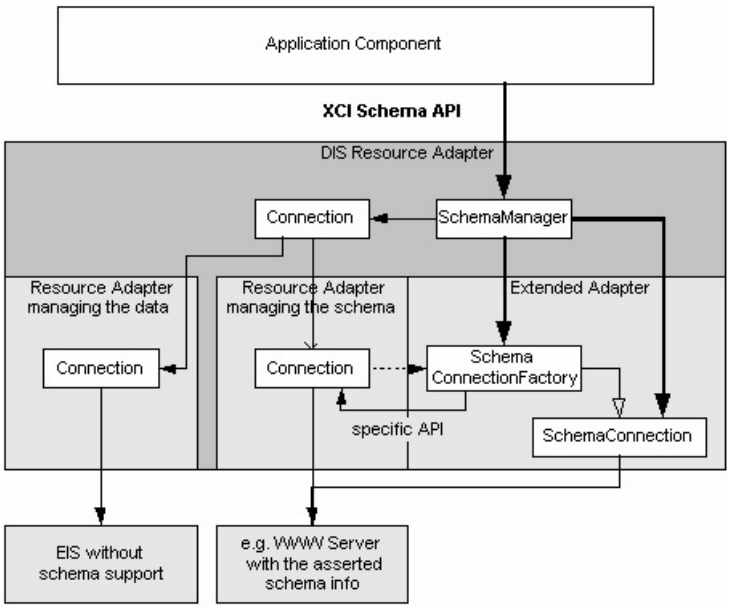


A resource adapter reads and writes the schema. This enables a flexible storage configuration using either the file system, a relational database or any other general purpose storage provider.

Schemas are information items themselves and have a URI. The storage system managing the schema does not have to be the storage that managing the schema-compliant primary data items. They can be managed by different resource adapters. This is common when

the connector is unable to access internal schema information but, instead, the schema is asserted from outside. Figure 3-7 illustrates this scenario:

Figure 3-7. Schema Management Architecture - Asserted Schema

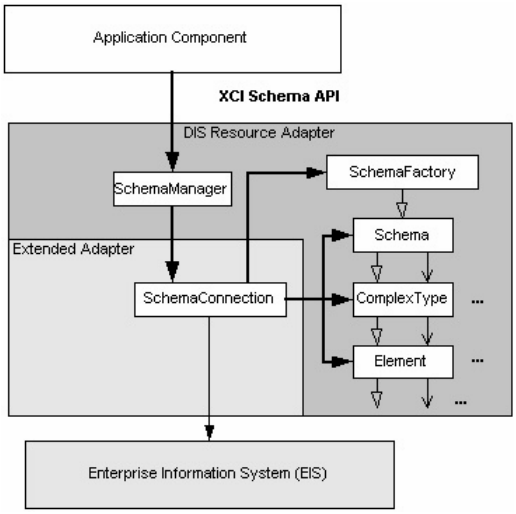


The right-hand-side (RHS) connection supplies the schema retrieved by the left-hand-side (LHS) connection. The RHS connection functions as the schema storage mechanism. The schema URI is only meaningful for the schema-managing connection. As part of the extended adapter, the `SchemaConnection` can call any proprietary API accessible through the resource adapter `Connection` object. The `SchemaConnection` can use either the `SchemaConnectionFactory` or the resource adapter's `Connection` to retrieve the schema and construct the schema components.

Schema Components

A schema encapsulates a strict hierarchy, defined by the XML Schema specification. Figure 3-8 illustrates this hierarchy:

Figure 3-8. Schema Management Hierarchy - Asserted Schema



Schema Roles

The DIS Schema Management Contract (SMC) uses a roles-based approach to functionality. It defines and uses the following entities:

- External Schema
- `SchemaManager`
- `SchemaConnectionFactory`
- `SchemaConnection` interface
- `SchemaFactory`
- Schema Components
- `SchemaEventListener`

External Schema

External schema are blueprints of the EIS schema expressed using the XML Schema conceptual model.

SchemaManager

The `SchemaManager` provides application components with the DIS entry point. This interface enables applications to lookup, create, validate, and change a schema. After the `SchemaManager` collects additional parameters, the DIS delegates future calls to the `SchemaConnectionFactory`.

SchemaConnectionFactory

The `SchemaConnectionFactory` provides the schema management entry point into the extended resource adapter. Using the schema management `Connection` (and a URI that identifies this Connection schema), it creates `SchemaConnection` objects. The Java Naming and Directory Interface (JNDI) supplies the lookup for the `SchemaConnectionFactory`. There is one `SchemaConnectionFactory` instance for each extended resource adapter, configured from an XML file. This file contains `SchemaConnectionFactory`-specific initialization parameters. The DIS configuration strategy is based on JavaBeans; for each *`value`* initialization parameter (in the XML file), the initialization code attempts to call a `setParam(`*`value`*`)`.

SchemaConnection interface

The `SchemaConnection` interface represents a pair of `Schema` and `Connection` objects. At any given time, multiple connections can be associated with a single `Schema` object, and multiple `Schema` objects associated with a single `Connection` object. The `SchemaConnection` object represents these associations. The `Connection` must guarantee that the same schema name (that is, URI) will result in the return of the same `SchemaConnection` object. The `SchemaConnection` cannot modify the schema name.

There is always a `Schema` object associated with a `SchemaConnection` returned from the `SchemaConnectionFactory`. A `SchemaConnection` object is in one of two states (these are, however, invisible to application components using the Schema Adapter). The external schema is either *in sync* or *out of sync* with the associated `Schema` object. The out-of-sync state occurs either after the creation of a `SchemaConnection` (by passing in a `Schema` object) or if an application component begins modifies a `Schema`. Application components can either use the `SchemaConnection.syncSchema` to create an external schema, or synchronize an external schema with the associated `Schema` object.

Because either can be modified independently, a `SchemaConnection` cannot guarantee that its referred-to `Schema` is identical with the external schema; however the `SchemaConnection` can detect a `Schema` object modification. The modification of an external schema cannot normally be detected.

SchemaFactory

The `SchemaFactory` constructs an initial `Schema` object, that is, the top-level `Schema` component. The Schema Manager provides the implementation of the `Schema` component interfaces; this is hidden from other modules.

Schema Components

Types, elements, attributes, and other metadata comprise `Schema` components. Containing component methods create all `Schema` components other than the `Schema` root object. Helper interfaces compose the component interfaces; each helper interface expresses a component role. For example, the interface `Schema` derives from the helper interface `TypeOwner`.

SchemaEventListener

The `SchemaEventListener` registers with a `Schema` object to receive read/write events concerning the in-memory schema. This is separate from the external schema modification event notification.

A `SchemaEventListener` enables the "lazy" construction of a very large in-memory schema. The Schema Adapter initially constructs the high-level object. It then watches the access to other schema objects and constructs them as needed.

The `SchemaEventListener` can also notify other modules that depend on the existence of specific schema information, such as the ontology or the DISQE. The listener follows the standard Java event design patterns (that is, veto, constraint, and property change).

If a connection is closed, the Schema Manager automatically removes any listeners still registered by the corresponding Schema Adapter.

Schema Scenarios

There are six key scenarios where an application component calls the `SchemaManager` methods:

- Reading an External Schema
- Sharing a Schema Object between Connections
- Creating a New External Schema
- Creating an External Schema from an Existing Schema Object
- Corroborating an External Schema
- Validating a Schema

In all cases, a `Connection` uniquely identifies each external schema.

Reading an External Schema

The `SchemaConnectionFactory.getSchemaConnection` method creates the `SchemaConnection`. This method receives a URI that denotes the schema name. The second parameter is a `SchemaFactory` object. The `Schema` root object is accessed using the returned `SchemaConnection`. To create an instance of `Schema`, the `SchemaConnection` calls the `SchemaFactory`. All subsequent calls to `SchemaConnection.getSchema` return the same `Schema` instance. After establishing the `SchemaConnection`, calling the `SchemaConnection.getSchema` method accesses particular `Schema` instances.

Sharing a Schema Object between Connections

An application component can assert that multiple `SchemaConnections` share a `Schema` object. The `SchemaConnection` makes no effort to locate a `Schema` object with the same name or structure. If the application component provides no `Schema` object, the `SchemaFactory` constructs a new `Schema` object (see above, "Reading an External Schema"). Two `SchemaConnections` could become associated with different `Schema` objects despite both connections being associated with the same external schema. To share a `Schema` object, it must be explicitly assigned to a `SchemaConnection` using a different `SchemaConnectionFactory.getSchemaConnection` method. This method receives a URI and a `Schema` object as parameters.

Creating a New External Schema

Creating an entirely new external schema follows the same procedure as "Reading an External Schema" above, except for the non-existence of the `Schema` specified by the external schema name URI. The returned `SchemaConnection` accesses the `Schema` root object. Calling the `SchemaConnection.isInSync` method checks whether an external schema exists. The returned `Schema` root object can be modified and the `SchemaConnection.syncSchema(SyncMethod.TO_EXTERNAL)` method creates the external schema.

Creating an External Schema from an Existing Schema Object

To create an external schema from an existing `Schema` object, pass the `Schema` object to the `SchemaConnectionFactory.getSchemaConnection` method. Calling the `SchemaConnection.syncSchema(SyncMethod.TO_EXTERNAL)` method checks whether the external schema exists. If the check fails and there is no external schema, it is created using the `Schema` components.

Corroborating an External Schema

Corroboration is the process of checking for equivalence between the external schema and the `Schema` objects in memory. Corroboration does not take place during the creation of the `SchemaConnection`, and there is no synchronization of the external schema with the `Schema` object. To corroborate, application components must use the `SchemaManager.isInSync` method. The method returns true if the `Schema` components correspond with the external schema; otherwise it throws an exception that details the variance.

Validating a Schema

Validation is the process of checking that a schema is valid, that is, that all schema components can be resolved. Validation must be explicitly called using the `SchemaConnection.isValid` method. Validation should not be confused with well-formedness in XML documents. An invalid Schema prompts the `SchemaManager.isValid` method to throw an exception detailing the problem.

Putting it All Together

Table 3-2 describes the results of calling the methods `getSchemaConnection(URI, SchemaFactory)` or `getSchemaConnection(URI, Schema)` methods, followed by `SchemaConnection.syncSchema(SyncMethod)`:

Table 3-2. getSchemaConnection Methods

Factory Method	`getSchemaConnection (URI, SchemaFactory)`	`getSchemaConnection(URI, Schema)` `SchemaConnection.syncSchema`	
External Schema		Method: To External	Method: From External
Exists	Create a `Schema` object, read the external schema	Update the external schema	Update the `Schema` object
Not Exists	Throw Exception	Create the external schema	Throw Exception

Interaction Management

Interactions enable an external actor to interact with an adapter. Each adapter exposes its interactions through public interfaces. These interfaces do not prescribe what an interaction does but instead define how to initiate interactions, how to pass parameters, and how to process results.

The interaction interfaces enable the following:

- Applications can use a broad range of interactions in a uniform way.
- Synchronous and asynchronous interactions.
- A uniform data transport mechanism.
- Well-defined interaction specifications.

Interaction Architecture

Each adapter provides an interaction factory that enables an application component to retrieve interactions. To retrieve these interactions, the application component must specify both the type and specification of interaction. An interaction specification is a string that defines the required interaction. There are standard interaction specifications, however an adapter can support non-standard interactions. For non-standard interactions, the corresponding interaction specifications must be defined.

There are four types of interactions. These are explained in Table 3-3:

Table 3-3. Types of Interactions

Property	Description
Synchronous request/response	These are similar to method calls or external resource queries. The outgoing part contains the parameters while the incoming part contains the results.
Synchronous one-way	These pass information to an external resource. Storing information in a database or publishing information to a message queue are good examples.
Notifications	These consume information that an external resource has asynchronously produced.
Solicit/Response	These serve external requests that require a response. The Knowledge Broker application offers to serve these requests. An application component invocation initiated by an external application or client (say, a request to invoke a servlet and return a response) is a typical example.

The four interactions use the utility interfaces listed in Table 3-4. These interfaces describe how both discrete and document data can be passed to and from interactions:

Table 3-4. Utility Interfaces

Interface	Description
`DataInput`	The `DataInput` interface reads data from the implementer. The actor implementing the `DataInput` interface plays the role of the data producer, that is, the side from where the input comes. The actor using the `DataInput` interface plays the role of the data consumer, that is, the side to where the input goes. It pulls the data from the data source (Cursor).
`DataOutput`	The `DataOutput` interface writes data to the implementer. The actor implementing the `DataOutput` interface plays the role of the data consumer, that is, the side to where the output goes. It pulls the data from the source (Cursor). The actor using the `DataOutput` interface plays the role of the data producer, the side from where the output comes.
`DocumentInput`	The `DocumentInput` interface reads data from the implementer. The actor implementing the `DocumentInput` interface plays the role of the data producer, that is, the side from where the input comes. It sends the data to the Content Handler. The actor using the `DocumentInput` interface is responsible for connecting the listener actor, also called the data consumer. This is the side to where the output goes. The data consumer (ContentHandler) receives the document data. It can also assume the role of the data producer itself.
`DocumentOutput`	The `DocumentOutput` interface writes data to the implementer. The actor implementing the `DocumentOutput` interface plays the role of the data consumer, that is, the side to where the output goes. It receives the data via the ContentHandler 'sink'. The actor using the `DocumentOutput` interface plays the role of the data producer, that is, the side from where the output comes. It pushes the data into the sink (the ContentHandler).

Request Object

The `Request` object encapsulates a `Connection` and a `SchemaConnection`, and extends the `java.util.Properties` class. The `Properties` class presents a persistent set of properties. The properties can be saved to or loaded from a stream. A string represents each key and its corresponding value in the property list. A `Request` object can be configured once, then stored either in a file or using JNDI. The following code illustrates how to configure a `Request`:

```
Request request = new Request();
    // Set standard properties
    request.setConnectorName("ORA");
    request.setConnectionURL(
        "jdbc:oracle:thin:@JUPITER:1521:MYDB");
    request.setSchemaURL("MYSchema");
    request.setUser(user);
    request.setPassword(password);
    // Set Adapter specific properties
    request.setProperty("Timeout", "10000");
```

The `Request` object also provides methods that simplify interaction and schema creation. It provides a default implementation that hides the usual tasks of Factory location and object creation, and so on:

```
OneWayOperation oneway =
     request.getOneWayOperation(interactionSpec);
  ...
  oneway.execute();
```

Queries - RequestResponse Interactions

Queries are formulated using the Quilt language. This provides a generic mechanism to execute the query against different data sources. A query must be formulated in conjunction with a schema. Queries execute using a `RequestResponse` interaction. Any input parameters are specified using the `DataOutput` interfaces. A programming analogy is JDBC parameter binding.

For more information about Quilt, see
`http://citeseer.nj.nec.com/chamberlin00quilt.html`

The DIS provides an application schema against which it is possible to formulate queries that can result in a partitioning of the query into sub-queries. These sub-queries are then delegated to their corresponding query adapter connector. A tree of sub-queries can thus result from a query; this requires a mechanism to "join" the sub-query results. These are *heterogeneous joins*. The DIS Query Engine (DISQE) manages generic steps such as validation. The adapter manages connector-specific steps.

Interaction Scenarios

There are three key interaction scenarios:

- Query Execution - Synchronous Call
- Query Execution - Asynchronous Call
- Data Change Interaction Execution

The `InteractionSpec` interface assumes the role of the CCI `InteractionSpec` interface.

Query Execution - Synchronous Call

A query can execute synchronously. The following steps are required to execute a synchronous query:

1. Retrieve an `InteractionSpec` from the `InteractionManager`. This requires a `SchemaConnection` and a textual representation of the required interaction
2. Request an instance of the `RequestResponse` interaction from the adapter's `InteractionFactory` (supply the `InteractionSpec` as argument).

Steps 1 and 2 can be accomplished by using a `Request` object.

3. Bind the parameters using the `DataOutput` interface on the `RequestResponse` object.
4. Call the `RequestResponse.execute` method. Catch any exceptions.
5. Retrieve the result from the `RequestResponse` as a `Cursor` object.

The following code illustrates the retrieval of people's names with specific ZIP codes (in this case, California):

```
String queryString =
    "FOR $a IN document(ADDRESS.xml)//address " +
    "WHERE $a/zip EQ $Zip " +
    "RETURN <TargetMarket>$a/NAME, $a/ZIP</TargetMarket>";

RequestResponse query = request.getRequestResponse(queryStr);
query.write("Zip", new String("CA"));
query.execute();
printCursor(query.read());
```

Query Execution - Asynchronous Call

A query can execute asynchronously, where the execution returns immediately and a callback listener delivers the results later. Asynchronous calls are useful when query execution requires a long time period. The `SolicitResponse` interaction executes asynchronous queries. The steps are similar to the synchronous call scenario, except that a data "sink" collects the asynchronous call result. This sink is specified as a `ContentHandler` and uses the `DocumentOutput` method.

```
ContentHandler getDataSink(String documentName);
```

The `documentName` parameter creates an identification (ID) that enables the sink provider to prepare to receive data.

Data Change Interaction Execution

Adapters supporting persistence can expose the Data Change Interaction ability by providing `OneWayInteractions` with a particular `InteractionSpec`. The DIS defines a standard syntax for formulating such a data change interaction. The `new`, `update`, and `delete` functions define data changes. They take as arguments a Quilt query with any restrictions stated. The Quilt query must also provide a projection that identifies which elements or attributes to change. The following code demonstrates how to change a telephone number associated with a name:

```
OneWayOperation oneway =
    request.getOneWayOperation(
    "update (\
    "FOR $a in document(\"ADDRESS.xml\")
    "WHERE $a/NAME = $NAME "
    "RETURN <Update> " +
    "$a/PhoneNumber
    </Update>\")");
oneway.write("NAME", name2Change);   // Bind the parameter $Name
oneway.write("a/PhoneNumber", "123456789"); // bind new phone number
oneway.execute();
```

Interaction Procedures

Application components can use one or more procedures to execute interactions and retrieve results. There are eight key procedures:

- Obtaining an Interaction object
- Executing an Interaction
- Closing an Interaction
- Canceling an Interaction
- Setting an Interaction Timeout
- Explaining an Interaction
- Monitoring Interaction Progress
- Handling Interaction Exceptions

Obtaining an Interaction object

The application component uses a `Request` instance for the particular adapter; calling the corresponding method retrieves the interaction. The `InteractionSpec` is supplied as a string so that the `Request` object can create and initialize the correct interaction.

Executing an Interaction

Calling the `Interaction.execute` method causes an interaction to execute. The values of Parameters specified in the `InteractionSpec` must be supplied before invoking the `execute` method.

Closing an Interaction

During interaction execution, the underlying adapter can elect to retain some resources (useful in case of interaction re-invocation). These resources can be expensive or scarce, and being able to free these resources when desired is valuable. To close an `Interaction` object, use the `Interaction.close` method; this makes the interaction invalid. Any subsequent `Interaction` method calls will throw an exception. Event listeners can be registered with an `Interaction` object and registered listeners are informed during instance closure. Using this mechanism, an adapter can listen for a close event (such as, for example, a JDBC close event) and take action.

Canceling an Interaction

Interaction can require considerable lengths of time. During this period, the caller could decide to cancel the call. The `Interaction.cancel` method attempts to cancel the interaction. The results depend on the stage of interaction execution; cancellation can be immediate, or it may require the completion of an external system call.

Setting an Interaction Timeout

Some systems offer interactions with timeout periods. This allows a caller to cancel the interaction if the timeout period has expired. If the interaction has failed to respond within the timeout period, the interaction aborts and throws an exception. The default interaction timeout period is infinite (no timeout).

Explaining an Interaction

The `Interaction.explain` method enables an adapter to explain the details of its intended interaction execution. The method returns a list of steps necessary to complete the interaction as well as detailed information concerning the transformations and states of the interaction at every step. Thus, in the case of a query issued against a JDBC connector, the explanation will contain the following annotated steps:

1. Mapping resolution – name resolving required
2. Validation – validation using the schema
3. Partitioning – whether the query was partitioned and, if so, which partitions are used
4. Delegation to the adapter
5. Adapter-supplied explanation (if applicable), for example, SQL statement

Monitoring Interaction Progress

If an interaction requires a long time period to execute (for example, to query a large XML document), a progress callback can be registered against the interaction. This callback informs any listeners of the interaction progress. The progress report specifies the name of the step as well as the percentage complete.

Handling Interaction Exceptions

Most interaction methods can throw the `InteractionException` class. This contains diagnostic information about the exception cause and can also contain information supplied by the underlying adapter.

Interactions are expensive in terms of resources, time, and memory. Many EISs are sensitive to large numbers of interactions executing simultaneously (or a large quantity of in-memory interactions). You should hold open a minimum of interactions and systematically close them when they are no longer required. Various adapters can also provide varying interaction abilities and restrictions. A query method caller should react to a known set of exceptions. For example, if no timeout can be set then ignore the "Not Supported" exception.

Interacting with XML Documents

The DIS defines these standard interactions for storing and retrieving XML documents:

Table 3-5. Storing and Retrieving XML Documents

Interaction	Description
PUT	Stores an XML document.
GET	Retrieves an XML document.
UPDATE	Overwrites an XML document.
DELETE	Deletes an XML document.

The following code demonstrates how to use an adaptor to store an XML document. XML documents require a unique filename for identification.

```
SAXParser parser = new SAXParser();
OneWayOperation oneway =  request.getOneWayOperation("PUT");
parser.setContentHandler(oneway.getDataSink(xmlFileName));
parser.parse(fileName);
oneway.execute();
```

Event Management

Not all EISs are passive, that is, synchronous requests are sent and external systems reply by returning results. Many systems such as stock quote services send a continuous or discontinuous series of messages for the DIS to consume. Message consumers are normally able to register interest in event subsets. As distinct from database-like synchronous systems, these are called *event sources*.

There are two different event categories. The first covers events that indicate interaction completion in the resource adapter. Applications register listeners to be asynchronously called following successful interaction execution. When the interaction produces data, an optional result is passed to the listener along with the event. A typical example of this type of event listener is a stock quote monitor.

The second event category addresses the arrival of logical document content. These events have a finer granularity than interaction completion events.

Architecting Events

Events are delivered to a callback object. The event consumer registers a listener or handler object with the resource adapter. More specifically, the registration is with an `Interaction` object.

There are two categories of listeners or handlers:

- InteractionListener
- ContentHandler

The `InteractionListener` object acts following interaction completion; an optional result is delivered along with the event. A `Cursor` object encapsulates the result content.

Data Management

This section describes how the DIS manages data structures and their model or schema. The DIS accesses, transforms, and outputs information based on domain-specific data models. The DIS, however, does not enforce its own object model. The fundamental principle underlying the DIS is that it is able to learn about, accommodate to, and work with arbitrary domain object models. This enables connectivity and integration with virtually all EISs. Applications can create their own object models in the same way database systems enable applications to create their own data model.

Once a domain-specific object model has been expressed in a format understandable by the DIS (that is, using a XML schema), a connection can be established to external EIS (using the Connection Management Contract) and an Interaction can be executed that returns the desired information items. These information items are described in such a way that the DIS can explore and use them. The data management mechanism enables access to these complex data structures. This section does not describe the invocation of external system functionality or how to consume external system events.

Working with arbitrary object models requires a strong meta-object model. The meta object model underlying the DIS is XML-based, but it is not equivalent to the Document Object Model (DOM). DOM takes a document-centric approach, whereas the DIS focuses on information entities (or objects). DOM is not a meta-model but a generic object model. Additionally, not all information item types are relevant for the DIS object models. For example, XML processing instructions, entity declarations, and so on are either ignored or do not affect the system data flow. The DIS meta-model enables the definition of items with specific types such as numbers, date, time, and boolean. XML atomic values are strings.

Architecting the Data Management

The Data Management architecture is based on a meta-data architecture, specifically the Schema Management architecture. A schema describes the domain object model's data structure but does not describe the methods or behavior. The Interaction model describes the meta-model's methods, while the Event model describes the meta-model's events.

An accessor interface encapsulates all data structures passed to and returned from external systems. This interface is the same for all external systems and all domain object models. The interface enables uniform access to heterogeneous data sources with proprietary meta- and domain object models. Each adapter is responsible for mapping between the meta-information. For example, an RDBMS accessed via JDBC has an adapter that translates between the Relational and DIS Schema meta models.

The interface allows generic access, which means that instead of using a specific property of a class, the property name will be passed as a string to the accessor interface. The adapter is responsible for mapping between the instance data structures. For example, an RDBMS that is accessed via JDBC has an adapter, which translates between the domain object models (ResultSet<->DIS accessor Interface). Notice that an RDBMS is also a system with a generic data model. With other words it is possible to define domain-specific object models.

The Data Management architecture handles large data structures efficiently. The accessor interface defines Cursor-like semantics that enable the `Accessor` object to point to a fragment of the underlying data structure without revealing the implementation or the full structure. The `Accessor` methods explore all the elements and attributes, but usually without returning the entire object (or structure of objects). To access different parts of the overall data structure, the `Accessor` user moves the `Cursor` to the next element. This moves the `Cursor` away from the previous element and allows the resource adapter to reclaim resources previously allocated during earlier accesses.

The Accessor interface is an alternative to event-based access. Using the Event Management Contract, an application provides a content handler that, when called, passes to the application the fragments of a complex data structure. The application here has no control over the fragment presentation order (unless the order was specified using a query).

The accessor interface serves as the low-level least common denominator. Other content formats are supported by transforming the `Cursor` into these formats. Two content handlers are provided with the DIS: one for JavaBeans, and another for DOM objects.

The Data Management Architecture contracts do not require either the Resource Adapters or the DIS services to instantiate potentially large amounts of small of objects to reconstruct complex data or object structures (these structures might not even fit into a single processing node's memory). Instead, the Cursor-like semantics require the adapters only to make accessible one object at a time. However, the adapter can use its own representation and an appropriate caching/pre-fetching strategy. Layers on top of the DIS can provide a caching layer that caches complex aggregated objects in their instantiated form.

Introducing the Cursor

The `Cursor` interface provides a lightweight mechanism to navigate arbitrary trees. It navigates using the concept of a current position within the tree. A `Cursor` object makes no assumptions about the structure or contents over which it navigates. Different physical representations can be hidden behind a `Cursor`, for example, a DOM tree or a JDBC result set. However, to correctly interpret the `Cursor` return values requires a schema. This schema details the exact nature of the returned `Cursor` values; the adapter that produced the `Cursor` must also have produced a corresponding schema.

The `Cursor` interface is defined here:

```
public interface Cursor {
void close() throws DataException;
void reset() throws DataException

// value, name and type methods --------------------
Object getValue() throws DataException;
String getName() throws DataException;
Type getType() throws DataException;

// Attribute count,  name, value and type method---------
int getAttributeCount() throws DataException;
String getAttributeName(int index) throws DataException;
Object getAttributeValue(int index) throws DataException;
Object getAttributeValue(String attributeName) throws DataException;
Type getAttributeType(int index) throws DataException;
Type getAttributeType(String attributeName) throws DataException;
String getNamespaceURI() throws DataException;

// navigation------------------------------
int next() throws DataException;
boolean nextSibling() throws DataException;
boolean parent() throws DataException;
}
```

The `close()` method tells the cursor that it is no longer required and to deallocate any internal resources. On initial construction, the initial `Cursor` position is before the first element. The `next()` method moves the `Cursor` to a valid position (if there is one). The `Reset()` method sets the `Cursor` state back to the original state, that is, to its state when created initially.

The methods `getValue()`, `getName()`, and `getType()` enable access to the data pointed to by the `Cursor`. The value types are described by a `Type` instance (part of the Schema). If the value is a simple type then various type properties can be queried and displayed, for example, maximum length, minimum value, and so on. Similarly, the current object's attributes can also be retrieved using the `getAttributeName()`, `getAttributeValue()`, and `getAttributeType()` methods.

If the `Cursor` position is invalid, attempted operations throw the `CursorInvalidPosition` exception; this extends the `DataException` standard exception.

The `next()` method moves the `Cursor` to the next position in the tree. The algorithm used to navigate the tree is has 5 steps:

1. Initially, the `next()` attempts to navigate to the first child object.
2. If a next child object does not exist, then `next()` backtracks and attempts to navigate to the next sibling.
3. If a next sibling object does not exist, then `next()` attempts to navigate to the parent's next sibling.
4. If the parent's next sibling does not exist, then `next()` attempts to navigate to the next sibling of the parent's parent.
5. This process continues until `next()` cannot navigate to any more ancestor nodes.

Using Cursors

Assume a query returned a `Cursor`. The following code demonstrates how to output the `Cursor` contents:

```
while((res = cursor.next())!= Cursor.END_OF_CURSOR){
    String name = cursor.getName();
    if (name != null)

        System.out.print("<" + name + ">");
    else
        System.out.print("No name");

    Type type = cursor.getType();

    if (type != null && type instanceof SimpleType){
        Type primitiveType = ((SimpleType)type).getPrimitiveType();

    System.out.println("Primitive type: " +
        primitiveType.getName());
    Object value = cursor.getValue();

    if (value != null)
        System.out.print(value.toString() + " ");
    else
        System.out.print("null ");
    }
}
```

The `next()` method navigates through the `Cursor` contents. The code prints each value encountered, as well as its name and primitive type.

Using DOMProvider

DOMs commonly represent in-memory XML data. Many tools use DOM to perform XML data operations. If an adapter's internal data representation uses DOM, it's useful to be able to pass the DOM tree outside of the adapter. To avoid unnecessary complexity in the `Cursor` interface, the `DOMProvider` interface offers this functionality. If an adapter wants to expose its DOM representation, it implements the `DOMProvider` interface:

```
interface DOMProvider {
    public org.w3c.dom.Node toDOM();
}
```

A `Cursor` consumer can then check the `Cursor` instance to see if it is also a `DOMProvider` instance:

```
if (cursor instanceof DOMProvider){
    Node node = ((DOMProvider)cursor).toDOM();
    // do something with the Node
}
```

The `toDOM()` method returns the DOM tree root node that the `Cursor` is using internally. As long as the returned DOM nodes are not modified, there is no effect on the `Cursor`. If they are modified, then the `Cursor` position may become invalid. It is the `toDOM()` caller's responsibility to manage both the DOM tree as well as the `Cursor` (the adapter no longer has complete control over the DOM nodes).

Distributed Information Service Query Engine (DISQE)

Application components use the DISQE to access data from more than one EIS. The DISQE is a virtual connector that provides a unified view of all available connection descriptors. The DISQE enables heterogeneous joins.

In the JCX architecture, there is a one-to-one mapping at the adapter or connector layer between each connector and its particular connecting EIS; a connector reads or writes data only to its connected EIS.

Application components communicate with the DIS and underlying connectors through a set of descriptor components provided by the application container. Just as application containers provide descriptor components for physical adapters, they also provide descriptor components for the DISQE virtual connector.

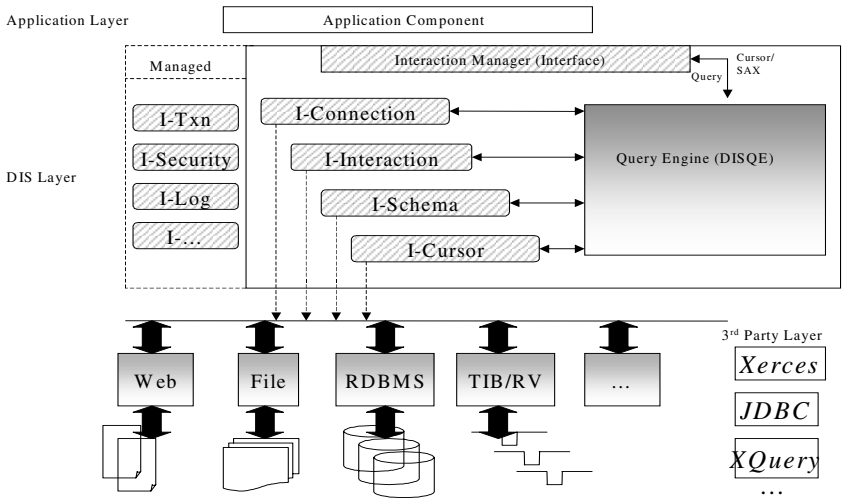
The DISQE initially supports one interaction descriptor, the `RequestResponse`. Designed to support data retrieval, this interaction receives data either synchronously (via `Cursor`) or asynchronously (via `ContentHandler`). The DISQE parses and evaluates queries using a built-in XQuery processor. Queries passed into the DISQE become formulated in XQuery.

DISQE queries using more than one connector descriptor are heterogeneous joins. When the same connector descriptor can retrieve the required data, the query is a homogeneous join. For these, the DISQE may pass the entire query down to the relevant connector.

Implementing the DISQE as a virtual connector leverages off the contract between application components and DIS-compliant connectors, using standard component descriptors. For example, application components querying RDBMS for data use the message descriptor (which identifies the desired interaction descriptor). If the data source changes, the application only needs to identify a different message descriptor (or change the existing descriptor).

The application can be quickly realigned with any EIS changes simply by altering the relevant descriptors. Where an application now begins to query data from multiple data sources presents, the use of descriptors shields the applications from the low-level system details. The DISQE is a process layer that sits over the physical connectors, encapsulating them.

Figure 3-9. DISQE



Analyzing the DISQE

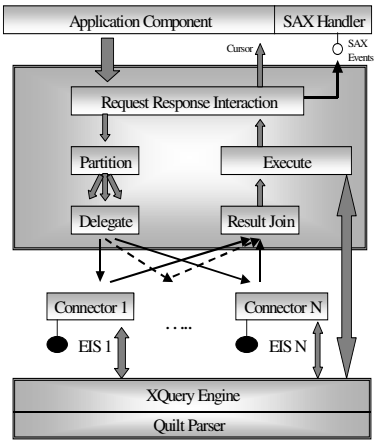
As with other connectors, the relevant descriptor components best describe the DISQE interfaces. A DISQE instance is described by its connector descriptor that encapsulates the connection. There is no requirement for more than one DISQE connection descriptor. There is one interaction descriptor for every query that client applications require to trigger DISQE execution. All interaction descriptors refer to the same connection descriptor, and all DISQE interactions are be `RequestResponse`. For each interaction descriptor, there should be no more than two message descriptors: the first specifies a synchronous data model and the second specifies an asynchronous data model. Applications typically invoke the DISQE by loading and executing a message descriptor that, using its descriptor chain, triggers the DISQE connector.

This sequence of steps illustrates the DISQE operations following an application component request for the DISQE to execute a given XQuery.

1. The DISQE partitions the XQuery input into a list of sub-queries. This is the partitioned query list. Each sub-query corresponds to a dataset that will be provided by the connection. Each dataset will, in turn, be associated with the sub-query.
2. The DISQE delegates each query from the partitioned query list to the associated connection's connector.
3. The connectors use the XQuery engine to execute the queries delegated to them (retrieving data from the underlying EIS), returning their result sets as `Cursor` objects.
4. For each delegated query, the DISQE receives a `Cursor`. It converts each of these `Cursor` objects to DOM trees for further processing. A result joiner process associates these DOM trees with their originator sub-query.
5. The DISQE generates a new XQuery specification based on the original XQuery and the result of the previous step.
6. The DISQE uses the XQuery engine to execute the internally generated query
7. The DISQE passes back to the user the query results as either a `Cursor` or a series of SAX events. This depends on how the application program specified the `RequestResponse` interaction.

Figure 3-10 illustrates these steps:

Figure 3-10. DISQE XQuery Execution



Describing the DISQE

Client applications use the DISQE through its descriptors; the DISQE appears to be subsystem and can be accessed through descriptor interfaces (for further details, see *Descriptors in Detail on page 67*). There are three key descriptors:

- DISQE Connector Descriptor
- DISQE Connection Descriptor
- DISQE Interaction Descriptor

DISQE Connector Descriptor

The JavaBean properties used by the DISQE connector descriptor include but are not limited to those in Table 3-6:

Table 3-6. DISQE Connector Descriptor JavaBean Properties

Property	Description
Name	DISQE
Connection URL	Not Applicable
Required Properties	None

Table 3-6. DISQE Connector Descriptor JavaBean Properties

Implementation Classes	These are provided for DISQE's interaction factory and managed connection factory. Initially, the managed connection factory will just return a DISQE instance.

DISQE Connection Descriptor

The JavaBean properties used by the DISQE connection descriptor include but are not limited to those in Table 3-7:

Table 3-7. DISQE Connection Descriptor JavaBean Properties

Property	Description
Name	DISQEConnection
Connector Name	DISQE
Connection Properties	None
Schema Connection Name	None

DISQE Interaction Descriptor

The JavaBean properties used by the DISQE interaction descriptor include but are not limited to those in Table 3-8:

Table 3-8. DISQE Interaction Descriptor JavaBean Properties

Property	Description
Name	RequestResponse
Connection Name	DISQEConnection
Timeout	0
Interaction Specification	This is the XQuery specification

Chapter

4

Descriptors in Detail

Descriptors enable application developers to create Knowledge Broker applications rapidly and economically. They provide a high-level Application Programming interface (API) that shields developers from the more complex, low-level APIs such as the DIS Connector interface, the DIS Query and Transformation interface (DISQE), and the inference engine interfaces.

Taking a Descriptive Approach

This descriptive approach to application programming is suitable for the construction of most Knowledge Broker applications that use services invoked from external applications (clients) that send and receive messages. In a Message-Oriented Middleware (MOM) infrastructure, these map to "real" delivered messages. But in a broader sense, the invocation of a servlet's service method or an EJB's business method is also message delivery.

Understanding the Descriptive Process

Application code generally continues querying data from the message and using it (together with its state) to assemble more complex data structures. If it requires any data in addition to that contained in the message, the application retrieves it from external data sources or applications. Knowledge Broker applications, therefore, perform a number of interactions with these resources and join the new data with whatever data is already part of the application's state.

The Knowledge Broker makes transformations during the information gathering process. Most external systems, applications, and data sources do not share a common data model. Data, therefore, must be mapped. The Knowledge Broker does not force an application to use specific data models. Instead, it enables data model definition as part of the application development process. These external data models (or schemas) can be easily mapped onto internal, dynamic Schemas. Additionally, the Knowledge Broker can calculate new data elements based on the values of assembled data elements.

After assembling the data, an application generally invokes internal operations on this data. These operations are various kinds of inferences. The gathered data is called evidence. The artificial intelligence (AI) components inference over networks of objects. Most of the application logic (except for the control flow described illustrated in this chapter) resides in knowledge bases. These combine an ontology with a collection of logical statements (a rulebase) suitable for inferencing.

The Knowledge Broker can join the AI components' output to existing data structures, or include it in the request message response, or push it out to another application (not the requestor). This other application could be a database that makes the data available for future requests. Alternatively, the Knowledge Broker could transform the output and map it onto new data structures (similar to the input data).

The process could continue with further inferences or, as a final stage, end here with result delivery.

The dominant application style is request-response. But the Knowledge Broker equally well supports publish-subscribe application style, where the application plays the role of a subscriber or publisher (or both).

All applications follow this control flow and programming the transformation, external interactions, and internal inferences can be tedious and time-consuming. Even though a significant portion of the application logic already resides in knowledge base logic statements, it's efficient to use chained descriptive tools to construct the control flow. With these in place, the application building process largely consists of configuring descriptors that contain information to drive the AI and DIS interfaces.

Role-Playing the Descriptive Process

Three main human actors use the Knowledge Broker's descriptors:

- Descriptor Author
- Descriptor User
- Descriptor Customizer

Descriptor Author

Descriptor Authors (or *Authors*) use a graphical interface to create descriptors and specify their properties. Their role is somewhat similar to that of an application developer, but they require no semantic programming skills. During construction, Authors must understand the application's internal control flow. They require this knowledge to link the descriptors appropriately. The Author leaves the descriptors to be stored in a partially or fully configured state.

Descriptor User

Descriptor Users (or *Users*) develop the application. They call the descriptors' public APIs to activate functions such as receiving request messages, joining and transforming data, and calling inference engines. They find the objects pre-configured: most of the descriptor properties will be already set. The User-developed application code performs the associated activities using relevant descriptors. For example, a connection descriptor opens a connection, a interaction descriptor calls an external resource, and a message descriptor sets interaction parameters and retrieves interaction results.

If the descriptors have been stored in a fully configured state, then no additional properties need to be set by the application code. The code simply retrieves the descriptor objects, which are already linked with each other, and executes the activity.

Descriptor Customizer

Descriptor Customizers (or *Deployers*) use the same tools as Descriptor Authors. However, Deployers set or refine those properties whose values can only be known in a deployment environment, or discovered during installation.

Using this approach, Authors and Users can develop and deliver complete applications. But for execution, all required descriptor properties must be configured by Deployers, who use Knowledge Broker tools to search for missing properties. Deployers do not add descriptors or modify the links between them (change the control logic).

Interacting with the Knowledge Broker Subsystems

Four main Knowledge Broker subsystems use the descriptors:

- Administration Service
- Container
- Connector
- Object Factory

Administration Service

The *Administration Service* (AS) maintains a descriptor repository for one or more applications. All descriptors have a unique, scoped name. The scope is called a *Model*. In fact, the container for all the descriptors is the Model, and each descriptor is contained within exactly one Model. The administration service manages each descriptors' name-to-object mapping. Some descriptor properties are in fact names of other descriptors, and descriptors themselves must resolve links to other descriptors. To support the descriptors here, the administration service maps names to descriptor objects. The AS also manages the persistent storage of all descriptors. The Descriptors' external representation uses an XML-based language.

The administration service reads the XML file(s) that comprise a model, instantiates a Java object for each descriptor, and sets the properties that are defined in the XML document. To use specific Java classes for a descriptor, the administration service delegates instantiation and property setting to the object factory subsystem.

Container

Containers manage application components and dispatch requests to them. Containers use the administration service to iterate over descriptors, then look for interaction descriptors that denote an external application request.

Although applications that run within a specific container (for example, the TIB/RV container) can use any connector for external system interaction, all containers always have a native connector. This connector accepts external service requests from other applications and dispatches these to application object requests without their initiation.

Connector

Connectors hold configuration information and drive other APIs. Connectivity descriptors perform interactions across process boundaries. Querying a database system or sending a message to another application are examples of such interactions.

Connector Descriptors describe a Knowledge Broker-specific extension of a standard J2EE connector. A Knowledge Broker Connector provides a means of accessing an external system, for example, a database or an Enterprise Information System. It also provides a Schema for the underlying data and also Interactions which it can perform, for exmaple, a Query.

Each connector provides exactly one connector descriptor. Currently, the connector API specifies the following connector names:

Table 4-1. Connector Names

Connector	Name
TIB/Rendezvous	`TIBCO`
Oracle	`Oracle`
WWW	`www`
File System	`FILE`

Object Factory

Special Java classes implement all descriptor interfaces. For example, the `com.blackpearl.descriptor.ConnectionDescriptorImpl` class implements the `com.blackpearl.api.descriptor.ConnectionDescriptor` interface. The administration server deserializes the XML data stream into an instance of the implementation object using the object factory subsystem. The descriptor subsystem provides deserialization handlers that instantiate the classes and appropriately set the properties. Driven by the object factory subsystem, the handlers either provide an instance of the `org.xml.sax.ContentHandler` interface or implement standard setter methods that accept the properties.

The Knowledge Broker externalizes all descriptors using an XML language. The bundled SAX-compliant parser is Apache Xerces. Deployers can configure the Knowledge Broker to use another suitable parser, if required.

You can find Apache Xerces information here:
`http://xml.apache.org/xerces-j/`

Organizing the Descriptor Interfaces

The Descriptor interfaces provide the external programming interface to the descriptor subsystem and also to other Knowledge Broker functionality. The Descriptor interfaces are available online as JavaDoc format.

This section organizes the descriptors into four sections, grouped by functionality. The four functional descriptor groups are listed in Table 4-2:

Table 4-2. Descriptor Groups

Function	Descriptor
Connectivity	Connector Connection
Activity	Interaction Message
Data Modeling	Schema Type
Business Logic	TBA

Each descriptor group depends somewhat on the previous group for complete functionality. However, every descriptor can be created and combined with other descriptors at any time. Development can proceed in parallel.

All descriptor interfaces share certain basic features. These are:

- Naming
- Properties
- Linking

Naming Descriptors

Every descriptor has a unique name, scoped within its class. The administration service uses this name to look up the descriptor. Descriptors generally use a consistent naming scheme.

Connector Descriptors

Connector descriptors use a short name of the middleware that provides the connection or the server that accepts the connection. Knowledge Broker users generally will not have to define connector descriptors. A finite number of connectors ship as part of the product.

Connection Descriptors

Connection descriptors hold the configuration information for a connection (as provided by a connector). Usually, the logical resource (being connected to) provides a connection name that identifies it. This could be the name of a database, another application, or a service. For example, an appropriate name for a database connection would be `SALESDB`.

Interaction Descriptors

Contains all the information to drive a spcific interaction with an external system. For example, to drive a Query.

Message Descriptors

Message descriptors denote what they transport. If the message is linked to a request-reply interaction, then one message should end with `Request` and the other with `Reply`.

Schema and Type Descriptors

Schema and type descriptors are free-form and flexible. They denote logical or physical artifacts of the modeled domain.

Map Descriptors

Map descriptors include the target type and start with `map`.

Descriptor Properties

Each descriptor contains distinct properties. These properties drive the underlying functionality and provide it with parameters. Because the descriptor properties can be set and used during deployment, an application does not have to provide these parameters. However, application code can set or overwrite these properties. Various descriptors have standard properties, and each standard property has specific getter and setter methods.

Generally, each property can be set using the method:

```
setProperty(String name, Object value)
```

The descriptor may store properties "non-supported" properties, but these are ignored. This mechanism can be used to provide customized properties for certain applications. Every descriptor implements a method that returns the names of supported properties. The Knowledge Broker descriptor editor GUI presents a variable, contextual number of fields. The on-screen format adjusts to suit whichever descriptor is being edited and to reflect the current user's security permission set. For example, the connection descriptor editor displays different text input fields for different connection types. These fields are familiar to an Author who knows the connector specifics.

Descriptor Users need to know when the Author has supplied all the required properties and, thus, when the descriptor can execute the low-level connection functionality. Three methods support this.

The first method returns the names of the properties:

```
String[] getRequiredProperties()
```

The second method returns a full list of all supported properties:

```
String[] getSupportedProperties()
```

The third method indicates when the descriptor is fully configured:

```
boolean isConfigured()
```

Fully configured does not mandate that all supported properties have been supplied. Some properties may have been set with default values. The set of required properties contains only those needed for functionality execution and that have no reasonable default value.

Linking Descriptors

Descriptors can have symbolic links (that is, a link defined through a name). These links can exist even when the related descriptor does not exist. There are typically two pairs of getter and setter methods. The first pair uses symbolic names, and the second uses the related descriptor object.

The method names are constructed using a scheme. The getter methods have the following signature (where `XYZ` is a placeholder for the descriptor type):

```
String getXYZName()
```

and

```
XYZDescriptor getXYZDescriptor();
```

The setter methods have the following signature:

```
void setXYZName(String name);
void setXYZDescriptor(XYZDescriptor descriptor);
```

Detailing the Descriptors

The descriptor properties are stored as XML 1.0-format files.

Connectivity Descriptors

TIBCO Connector Descriptor

The `tibco.xml` connector connectivity descriptor file specifies both the connector and some default connection properties. The default file looks like this:

```
<?xml version="1.0"?>
<connectorDescriptor>
    <name>TIBCO</name>
    <URLPrefix>TIBCO</URLPrefix>

    <!--Factories-->
    <ManagedConnectionFactory>com.blackpearl.dis2.adapter.
              tibco.ManagedConnectionFactoryImpl</ManagedConnectionFactory>
    <SchemaConnectionFactory>com.blackpearl.dis2.adapter.
                tibco.SchemaConnectionFactoryImpl</SchemaConnectionFactory>
    <InteractionFactory>com.blackpearl.dis2.adapter.
```

```
            tibco.interaction.InteractionFactoryObject</InteractionFactory>

    <!-- Connection Properties-->
    <connectionProperties>
        <service required="true"/>
        <daemon required="true"/>
    </connectionProperties>
</connectorDescriptor>
```

The parameter contents of this file are:

Name

The `name` element uniquely identifies the connector type. References to this name in the `model.xml` by either the `connectionDescriptor.connectorName` or the `schemaDescriptor.connectorName` serve to link together descriptors in a logical flow. In this case, `TIBCO`.

URLPrefix

The `URLPrefix` element contains a short string that prefixes the connector URL and helps to identify the type of resource and protocol required. The `URLPrefix` for the TIBCO connector is `TIBCO`.

ManagedConnectionFactory

The `ManagedConnectionFactory` element identifies the Java class (one of the primary JCX classes) that enables connection pooling with methods for matching and creating a `ManagedConnection` instance. `ConnectionFactories` provide interfaces to get `Connections` to EIS instances (in this case, a TIBCO message source). Clients use `Connections` as application-level handles to access the underlying physical connection, in this case, a connection to a TIBCO resource. A client gets a connection instance by using the `getConnection()` method on a `ConnectionFactory` instance. `ManagedConnection` instances created by the `ManagedConnectionFactory` represent the physical connections associated with a TIBCO `Connection` instance.

The `ManagedConnectionFactory` class for the TIBCO connector is `com.blackpearl.dis2.adapter.tibco.ManagedConnectionFactoryImpl`.

SchemaConnectionFactory

The `SchemaConnectionFactory` element provides similar functionality as the `ManagedConnectionFactory` but expands the JCX architecture by enabling Schema support.

The `SchemaConnectionFactory` class for the TIBCO connector is `com.blackpearl.dis2.adapter.file.SchemaConnectionFactoryImpl`.

InteractionFactory

The `InteractionFactory` element identifies the Java class (one of the primary JCX classes) that enables connection pooling with methods for matching and creating a `Interaction` instance. `Interactions` provide interfaces to invoke associated functions in the underlying EIS (in this case, a TIBCO message source). Clients use `Interactions` as application-level handles to access the interactions. A client gets an interaction instance by using the `getInteraction()` method on an `Interaction` instance.

The `InteractionFactory` class for the TIBCO connector is `com.blackpearl.dis2.adapter.tibco.interaction.InteractionFactoryObject`.

The `tibco.xml` connector descriptor also details some default and required connection properties:

service

This is a Rendezvous API (Version 6) Network Transport Parameter. Different values of `service` can isolate independent distributed applications running on the same network from one another.

daemon

This is a Rendezvous API Network Transport Parameter. You can specify a particular remote TIB/Rendezvous daemon using the `daemon` parameter.

TIBCO Connection Descriptor

The TIBCO connection descriptor includes extra parameters. Here is a typical connection descriptor (contained in the `model.xml` file):

```
<?xml version="1.0"?>
<model>
...
        <connectionDescriptor>
                <name>tibcoCxn</name>
                <url>10.0.100.100</url>
                <service>null</service>
                <daemon>null</daemon>
                <refreshTimeout>-1</refreshTimeout>
                <connectorName>TIBCO</connectorName>
                <schema>fileSchemaCxn</schema>
        </connectionDescriptor>
</model>
```

The extra descriptor parameters defined here are:

url

This is Black Pearl's name for the Rendezvous API Network Transport Parameter called "network." You can use the url/network parameter to control multicast addressing or to address a specific outbound network interface on systems with multiple network interfaces. Selecting the `null` value specifies the default NIC.

refreshTimeout

The `refreshTimeout` specifies the length of time in milliseconds the connection waits to receive a TIBCO message before timing out. A setting of `-1` specifies an infinite wait time.

connectorName

The `connectorName` identifies which connector descriptor this connection references. In this case, the value is `TIBCO`.

schema

The `schema` identifies the named schema descriptor that points to an XML Schema file that describes how to convert between the TIBCO/Rv message format and the Knowledge Broker's internal in-memory representations.

Oracle Connector Descriptor

The `oracle.xml` connector connectivity descriptor file specifies both the connector and some default connection properties. The default file looks like this:

```
<?xml version="1.0"?>
<connectorDescriptor>
    <name>Oracle</name>
    <URLPrefix>ORACLE</URLPrefix>
    <ManagedConnectionFactory>com.blackpearl.dis2.adapter.database.
                    ManagedConnectionFactoryImpl</ManagedConnectionFactory>
    <SchemaConnectionFactory>com.blackpearl.dis2.adapter.database.
                      SchemaConnectionFactoryImpl</SchemaConnectionFactory>
    <InteractionFactory>com.blackpearl.dis2.adapter.database.
```

```
                    interaction.InteractionFactoryObject</InteractionFactory>

    <connectionProperties>
        <urlPrefix required="true">Oracle</urlPrefix>
        <serverName required="true"/>
        <portNumber required="true">1521</portNumber>
        <userName required="true"/>
        <password required="true"/>
        <driver required="true">oracle.jdbc.driver.OracleDriver</driver>
        <protocol required="true">oracle:thin</protocol>
        <databaseName required="true"/>
        <timeout/>
    </connectionProperties>
</connectorDescriptor>
```

The parameter contents of this file are:

Name

The `name` element uniquely identifies the connector type; in this case, `oracle`.

URLPrefix

The `URLPrefix` element contains a short string that prefixes the connector URL and helps to identify the type of resource and protocol required. The `URLPrefix` for the Oracle connector is `ORACLE`.

ManagedConnectionFactory

The `ManagedConnectionFactory` element identifies the Java class (one of the primary JCX classes) that enables connection pooling with methods for matching and creating a `ManagedConnection` instance. `ConnectionFactories` provide interfaces to get `Connections` to EIS instances (in this case, an Oracle RDBMS). Clients use `Connections` as application-level handles to access the underlying physical connection, in this case, the connection to the Oracle database. A client gets a connection instance by using the `getConnection()` method on a `ConnectionFactory` instance. `ManagedConnection` instances created by the `ManagedConnectionFactory` represent the physical connections associated with an Oracle `Connection` instance.

The `ManagedConnectionFactory` class for the Oracle connector is `com.blackpearl.dis2.adapter.database.ManagedConnectionFactoryImpl`.

SchemaConnectionFactory

The `SchemaConnectionFactory` element provides similar functionality as the `ManagedConnectionFactory` but expands the JCX architecture by enabling Schema support.

The `SchemaConnectionFactory` class for the Oracle connector is `com.blackpearl.dis2.adapter.database.SchemaConnectionFactoryImpl`.

InteractionFactory

The `InteractionFactory` element identifies the Java class (one of the primary JCX classes) that enables connection pooling with methods for matching and creating a `Interaction` instance. `Interactions` provide interfaces to invoke associated functions in the underlying EIS (in this case, an Oracle RDBMS). Clients use `Interactions` as application-level handles to access the interactions. A client gets an interaction instance by using the `getInteraction()` method on an `Interaction` instance.

The `InteractionFactory` class for the Oracle connector is `com.blackpearl.dis2.adapter.database.interaction.InteractionFactoryObject`

The `oracle.xml` connector descriptor also details some default and required connection properties:

urlPrefix

The `urlPrefix` element contains a short string that prefixes the connector URL and helps to identify the type of resource and protocol required. The default `URLPrefix` for the Oracle connector is `ORACLE`. This element is required.

serverName

The `serverName` element identifies the RDBMS server host using standard IP addressing. This element is required.

portNumber

The `portNumber` element identifies the port used for communication with the Oracle RDBMS. The default `portNumber` for the Oracle connector is `1521`. This element is required.

userName

The `userName` element supplies the username credentials of a principal authorized to access the Oracle RDBMS. This element is required.

password

The `password` element supplies the password credentials of a principal authorized to access the Oracle RDBMS. This element is required.

driver

The `driver` element identifies the Java class that contains a driver suitable for communication with the Oracle RDBMS. The default `driver` for the Oracle connector

is `oracle.jdbc.driver.OracleDriver`. This class file must be accessible through the Java `CLASSPATH` environment setting. This element is required.

protocol

The `protocol` element identifies the protocol used for communication with the Oracle RDBMS. The default `protocol` for the Oracle connector is `oracle:thin`. This element is required.

databaseName

The `databaseName` element identifies the name of the Oracle RDBMS. This element is required.

timeout

The `timeout` element specifies how long in milliseconds to wait for a response from the Oracle RDBMS.

Oracle Connection Descriptor

The Oracle connection descriptor includes some extra parameters and provides values for others defined initially in the `oracle.xml` file. Here is a typical connection descriptor (contained in the `model.xml` file):

```
<?xml version="1.0"?>
<model>
...
        <connectionDescriptor>
                <name>oraLocalCxn</name>
                <serverName>10.0.2.84</serverName>
                <portNumber>1521</portNumber>
                <userName>SYSTEM</userName>
                <password>MANAGER</password>
                <connectorName>oracle</connectorName>
                <schemaName>oraLocalCxn</schemaName>
                <driver>oracle.jdbc.driver.OracleDriver</driver>
                <protocol>oracle:thin</protocol>
                <databaseName>ORCL</databaseName>
        </connectionDescriptor>
</model>
```

The extra descriptor parameters defined here are:

connectorName

The `connectorName` identifies to which connector descriptor this connection refers. In this case, the value is `oracle`.

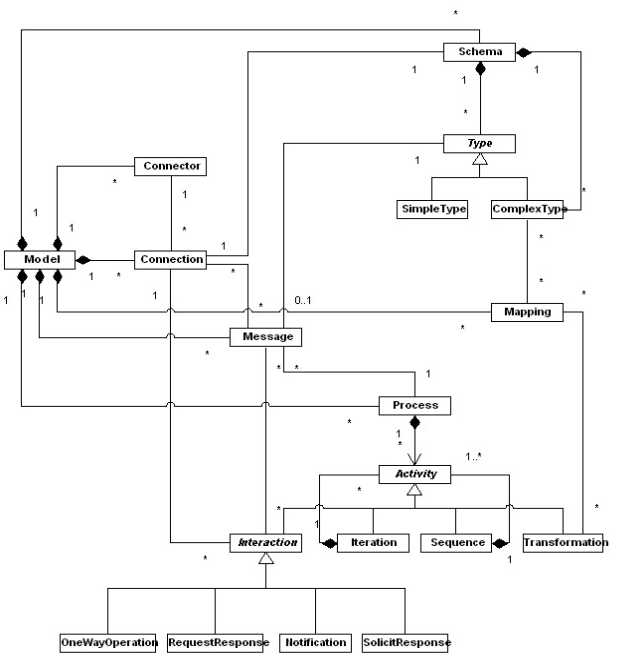
schemaName

The `schema` identifies the named schema descriptor that points to an XML Schema file that describes how to convert between the Oracle data format and the Knowledge Broker's internal in-memory representations.

Visualizing the Descriptors Graphically

The relations between the descriptors can be seen from Figure 4-1:

Figure 4-1. Analyzing Descriptors



Chapter 5

Building an Application

Creating a call list for financial brokers is a classic demonstration of how to integrate the Knowledge Broker's reasoning capabilities with external data sources. This chapter explains how to code such an application.

Building an Application Using Descriptors

The best way to illustrate the use of descriptors is to work through a comprehensive application example that will demonstrate how descriptors simplify development by factoring out a large portion of the application code. This example will detail the descriptor properties, demonstrate how Authors work with descriptors, how Users write applications using the public API, and finally how Deployers configure the application for a particular execution environment (in this case, TIB/Rv).

Analyzing the Application

The application goal is to listen for a specific message ("NASDAQ Tanks"). This message triggers an inference that creates a client list. This list could be used, for example, to issue limit sell order recommendations for specific stocks.

There are three interactions, in sequence:

1. Event Trigger
2. Reason to Call (RTC) List
3. Customer Recommendations

Event Trigger

The event trigger is a specific TIBCO-format message that arrives "on the wire," advertising that the NASDAQ composite index has "tanked." This will have an immediate impact on most clients who are heavily invested in technology stocks.

Reason to Call (RTC) List

Following the event trigger, the system presents the Financial Consultant (FC) with a list of clients to call. The RTC list omits clients with no pending advice. For this demo application, a simple message ("NASDAQ Tanks") triggers a response from an inference ("IF customer highly-weighted in technology sector AND NASDAQ tanks THEN recommend limit sell") and displays a list of clients to call.

Customer Recommendations

Selecting a client to call will trigger a subsequent screen that describes in detail the recommended limit sell orders for each specific customer based on their stock purchase prices, investing history, and risk preferences. This control logic for this subsequent screen is outside the scope of this simple demo.

Limit Sell Call List

This interaction is pushed to the FC and contains a list of clients with the appropriate "reason-to-call" (RTC) associated with them (in this case, "limit sell").

Using the External Descriptor API

This section outlines the descriptive methodology's approach to application development. The GUI development procedure requires three stages:

1. Create interaction descriptors for all the features (described in the previous section).
2. Create message descriptors for the interactions.
3. Create type descriptors and put them into the schema descriptor.

Using the Interaction Descriptors

The Knowledge Broker stores its configuration information as a chain of linked descriptors in the `model.xml` file.

To create the RTC Application, the Author specifies interactions using interaction descriptors. At this point, the interactions do not have to be tied to a particular connection. Table 5-1 details the interaction descriptors that are understood by this version of the Knowledge Broker and the associated `RTCApplication` class (see `RTCApplication.java` at the end of the chapter for details). Service styles are called from outside, while Consume styles (not used here) require no response:

Table 5-1. Interaction Descriptors

Name	Style	Function Name	Interaction Verb	Input Message	Output Message
RTCRequest	service	Listen	SYNC_RECEIVE	-	-
RTCReply	-	Send	SYNC_SEND	-	-

After entering the interaction descriptor information, the `model.xml` file contains the following RTCApplication-specific interaction descriptors:

```
<?xml version="1.0"?>
<model>
        ...
<!-- APPLICATION:  RTC Message Interaction Descriptors -->
    <interactionDescriptor>
        <name>RTCRequest</name>
        <subject>RTCRequest</subject>
        <connectionName>tibcoCxn</connectionName>
        <functionName>Listen</functionName>
        <interactionVerb>SYNC_RECEIVE</interactionVerb>
        <style>service</style>
        <applicationService>RTCApplication</applicationService>
    </interactionDescriptor>

        <interactionDescriptor>
        <name>RTCReply</name>
        <subject>RTCReply</subject>
        <connectionName>tibcoCxn</connectionName>
        <functionName>Send</functionName>
        <interactionVerb>SYNC_SEND</interactionVerb>
    </interactionDescriptor>
    ...
 <interactionDescriptor>
        <name>SendTestMessage</name>
        <expression/>
        <connectionName>testMessageFile</connectionName>
        <functionName>Get</functionName>
        <interactionVerb>SYNC_RECEIVE</interactionVerb>
        <outputMessageName>sendTestMessage</outputMessageName>
    </interactionDescriptor>
    ...
    <interactionDescriptor>
        <name>SendTestMessageNow</name>
        <subject>RTCRequest</subject>
        <connectionName>tibcoCxn</connectionName>
        <functionName>Send</functionName>
        <interactionVerb>SYNC_SEND</interactionVerb>
        <inputMessageName>sendTestMessage</inputMessageName>
    </interactionDescriptor>
</model>
```

Using the Message Descriptors

In a complex application, message descriptors facilitate complex message handling. In this simple application, they are not required. A non-functional, demo message descriptor is presented in Table 5-2 .

Table 5-2. Message Descriptors

Name	Type
sendTestMessage	RTCRequest

After entering this message descriptor information, the following content is added to the model file:

```
<?xml version="1.0"?>
<model>
    ...
<messageDescriptor>
        <name>sendTestMessage</name>
        <type>RTCRequest</type>
    <interactionDescriptorName>SendTestMessageNow
                                                </interactionDescriptorName>
    </messageDescriptor>
    ...
</model>
```

Using the Type Descriptors

The final GUI task for the Author is to define the types for the message content. These definitions are contained within an external schema file (`.xsd`) referenced by a schema descriptor. One schema file contains the Concept definitions, the other the message type definitions. The following content is added to the model file:

```
<?xml version="1.0"?>
<model>
    ...
    <schemaDescriptor>
        <name>ConceptSchema</name>
        <connectorName>FILE</connectorName>
        <url>e:/demos/rtc/container/schemas/RTCConcepts.xsd</url>
        <schema/>
    </schemaDescriptor>
    ...
    <schemaDescriptor>
        <name>mesgSchemaCxn</name>
        <connectorName>FILE</connectorName>
        <url>E:/demos/rtc/container/schemas/RTCMessageSchemas.xsd</url>
    </schemaDescriptor>
    ...
</model>
```

Customizing the Schema Descriptors

The Author defines the "outward" application interface using the GUI. The focus now shifts to the internal functionality description through the control of the type and composition of the messages.

For XML Schema editing, use the TIBCO Extensibility editor (XML Authority), or a similar tool.

See this link for further information about TIBCO Extensibility:
`http://www.extensibility.com/`

The `RTCMessageSchemas.xsd` file contains a list of `complexType` definitions of message contents, as well as a series of `simpleType` definitions using the the XML Schema `restriction` property to perform simple type mappings.

The Author could stop creating descriptors at this point and hand over to the Descriptor User, who could now begin coding Java objects to use the descriptors. But there is still some work to be done.

Creating the Descriptor User's Control Flow

So far, the application has no control flow - data types have been created but no specifications for the data routing or transformations have been defined and only a static data schema exists. The Descriptor User's responsibility is to add control flow logic to create a dynamic, responsive application. This requires the use of externalk Java programming tools.

The User accesses the descriptor API using programmatic Java code. This section outlines illustrative (but not necessarily complete) code examples.

Every application must implement the `ApplicationService` interface. In this case, the demo uses the `AbstractApplicationService`, a sub-class of `ApplicationService`. This class encapsulates methods that provide parameter preparation and message hanbdling and dispatch.

```
public class RTCApplication extends AbstractApplicationService {
  private static final String ROOT_NAME = "RTCRequest";
  private static final String INTERACTION_NAME = "RTCReply";
  private static final String PCT_TECH_STOCK_NAME =
    "PercentageTechnologyStocks";
  private static final String EVENT_NAME = "Event";
  private Container m_container = null;
  private AdvisorWrapper m_advisor = new AdvisorWrapper("MarketAdvisor");
...
```

```
}
```

The `private static final String ROOT_NAME = "RTCRequest";` code line is necessary because TIBCO messages do not have a single root. To ensure XML compatibility, a single root must be added when TIBCO messages are received by or stripped when TIBCO messages are sent from the Knowledge Broker.

Demonstrating Message Processing

The code example in this section demonstrates how an application processes an incoming message. In this case, the Knowledge Broker's inferencing mechanisms are listening for a message: "NASDAQ Tanks". This key piece of "evidence" will trigger an inferred response from the rules contained in the Knowledge Broker's knowledge base that will produce an RTC alert. In a more advanced scenario, this list would comprise Clients with heavy technology weightings where a "Limit Sell" order makes good financial sense. In this simple demo, the customers remain unidentified.

Inferring Using the Evidence

The `NewOntology.xml` file encodes the business intelligence that associates concepts with messages and desired rules-based responses. Information concerning this file's location and parameters are written automatically within the `model.xml` file using descriptors:

```
<?xml version="1.0"?>
<model>
    ...
   <!-- KB: Ontology SubSystem -->
    <interactionDescriptor>
        <name>getOntologyData</name>
        <expression/>
        <connectionName>ontologyFile</connectionName>
        <functionName>Get</functionName>
        <interactionVerb>SYNC_RECEIVE</interactionVerb>
    </interactionDescriptor>

    <interactionDescriptor>
        <name>putOntologyData</name>
        <expression/>
        <connectionName>ontologyFile</connectionName>
        <functionName>Put</functionName>
        <interactionVerb>SYNC_RECEIVE</interactionVerb>
    </interactionDescriptor>

    <connectionDescriptor>
        <name>ontologyFile</name>
        <connectorName>FILE</connectorName>
        <url>e:/demos/rtc/ontology/NewOntology.xml</url>
```

```
        <schema>ontologySchema</schema>
    </connectionDescriptor>

    <schemaDescriptor>
        <name>ontologySchema</name>
        <connectorName>FILE</connectorName>
        <url>e:/demos/rtc/container/schemas/ontology.xsd</url>
        <schema/>
    </schemaDescriptor>

    <ontologyDescriptor>
        <name>BlackPearlOntology</name>
        <BusinessConceptBaseURL>e:/demos/rtc/ontology/NewOntology.xml
                                                    </BusinessConceptBaseURL>
        <RuleBaseURL>e:/demos/rtc/ontology/BPRuleBase.xml</RuleBaseURL>
        <ActionsBaseURL></ActionsBaseURL>
        <CalculationsBaseURL></CalculationsBaseURL>
        <MappingsBaseURL></MappingsBaseURL>
        <RuleBaseURL_Temp>e:/demos/rtc/ontology/BPRuleBase.xml
                                                        </RuleBaseURL_Temp>
    </ontologyDescriptor>
    ...
</model>
```

The final `ontologyDescriptor` is required for future functionality enhancements.

Communicating With the Knowledge Broker

For the Knowledge Broker's inference engines to accept and process information, the messages must be converted into an ordered list of attributes and values, the "evidence". This is an `ArrayListAttributeList` object. This object is passed to and from the Knowledge Broker using various `Cursor` objects. The `Cursor` objects can use utility classes that provide DOM Parsing and other XML handling and formatting methods.

Analyzing the Application Code

The entire `RTCApplication.java` code is reproduced here. Explanatory code comments have been highlighted.

```
package rtcapp;

import java.io.*;
import java.util.*;

// DOM parser is XERCES
import org.apache.xerces.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;

import com.blackpearl.container.*;
```

```
import com.blackpearl.api.connector.Cursor;
import com.blackpearl.api.connector.DataException;
import com.blackpearl.api.descriptor.DescriptorException;
import com.blackpearl.api.descriptor.InteractionDescriptor;
import com.blackpearl.api.descriptor.Message;
import com.blackpearl.dis2.common.DOMCursor;
import com.blackpearl.api.container.Container;
import com.blackpearl.api.container.ServiceException;

import com.blackpearl.standalone.AdvisorWrapper;
import com.blackpearl.application.utilities.*;

/**
 * Reason To Call Application.
 *
 * This application listens for a request
 * which contains the fields Account Number,
 * Event and PercentageTechnologyStocks. This application then generates a
 * recommendation and broadcasts under the subject "RTCReply".
 */

public class RTCApplication extends AbstractApplicationService {
  private static final String ROOT_NAME = "RTCRequest";
  private static final String INTERACTION_NAME = "RTCReply";
  private static final String PCT_TECH_STOCK_NAME =
                                              "PercentageTechnologyStocks";
  private static final String EVENT_NAME = "Event";
  private Container m_container = null;
  private AdvisorWrapper m_advisor = new AdvisorWrapper("MarketAdvisor");

  /**
   * Constructs the RTCApplication.
   */

  public RTCApplication() {}

  /**
   * Call back from the container when an incoming message arrives.
   * This IS the application.
   *
   * The processMessage method is from the AbstractApplicationService.
   * The underlying service routine is specified in
   * com.blackpearl.api.Container.
   *
   * General Flow:
   * 1.  Listens for RTCRequests
   * 2.  Gathers evidence for the Knowledge Broker
   * 3.  Asks the Knowledge Broker for recommendations
   * 4.  Processes the recommendations
   * 5.  Sends a broadcast message out with the recommendation
   */

  public void processMessage(MessageHdl incoming) {
```

```
// 1.  Listens to incoming requests

// This call back occurs when an incoming message arrives
Cursor data = incoming.input.getCursor();

// 2.  Gathers evidence for the Knowledge Broker
// By extracting the data from the incoming message

String event = null;
String pct_tech_stock = null;
try {
  while (data.next() != Cursor.END_OF_CURSOR) {
    if (EVENT_NAME.equals(data.getName())) {
      event = data.getValue().toString();
    }
    else if (PCT_TECH_STOCK_NAME.equals(data.getName())) {
      pct_tech_stock = data.getValue().toString();
    }
  }
}
catch (DataException ex) {
  // log the exception
  System.err.println("Error processing incoming message.
                                                  Got exception" + ex);
  return;
}

// Make sure that we have all the necessary information

if (event == null || pct_tech_stock == null) {
  System.err.println("Error processing incoming message.
     Message does not contain all required fields.  Ignoring message.");
  return;
}

// Transform the data into something KnowledgeBroker understands

ArrayListAttributeList attributes = new ArrayListAttributeList();
attributes.add("Event", event);
attributes.add("PercentageTechnologyStocks", pct_tech_stock);
Map evidenceMap = new HashMap();
evidenceMap.put(attributes, "RTCRequest");
com.blackpearl.dis.Cursor evidence = new MapCursor(evidenceMap);

// 3.  Asks the KnowledgeBroker for recommendations

com.blackpearl.dis.Cursor result = null;
try {
  result = m_advisor.getInferences(evidence);
}
catch (Exception ex) {

  // log it
```

```
  System.err.println("Error inferring: " + ex);
  return;
}

// 4.  Process the recommendations

String recommendation = null;
try {
  while (result.next() != Cursor.END_OF_CURSOR) {
    System.out.println(result.getLocalName()
                                          + ": " + result.getObject());
    if (result.getLocalName().equals("explanation")) {
      recommendation = result.getObject().toString();
    }
  }
}
catch (Exception ex) {

  // log it

  System.err.println("While processing results got: " + ex);
  return;
}

// 5.  Sends a broadcast message out with the recommendation

// Construct a cursor with the advice embedded within it

String xml = "<RTCReply>" + recommendation + "</RTCReply>";
DOMParser parser = new DOMParser();
Cursor outCursor = null;

try {
  parser.parse(new InputSource(new StringReader(xml)));
}
catch (Exception ioe) {
  System.out.println("FATAL: could not parse message!");
  System.out.println(xml);
  ioe.printStackTrace();
  System.exit(1);
}
try {
  outCursor =
    new DOMCursor(parser.getDocument(),
                  m_container.getAdminService()
                    .getSchemaDescriptor("mesgSchemaCxn").getSchema());
}
catch (DescriptorException de) {
  de.getNestedException().printStackTrace();
}
catch (DataException de) {
  de.getNestedException().printStackTrace();
}
```

```
    // Now use the RTCReply interaction and
    // send out the cursor we constructed above.


    InteractionDescriptor sendInteractionDescriptor =
      m_container.getAdminService()
        .getInteractionDescriptor(INTERACTION_NAME);

    Message outMsg = new Message(null, outCursor);
    try {
      sendInteractionDescriptor.execute(outMsg);
    }
    catch (DescriptorException de) {
      de.getNestedException().printStackTrace();
    }

    // Done
  }

  /**
   * Returns the name of the application.
   */

  public String getName() {
    return "RTCApplication";
  }

  /**
   * Configures the application.  Just keep a reference to the container
around.
   */

  public void configure(Container container,
                        Properties properties) throws ServiceException {
    super.configure(container, properties);
    m_container = container;
  }
}
```

Writing the Deployer Descriptor

Following the packaging and coding, the application is ready for deployment. The descriptors do not yet contain parameters specific to the execution environment, how the requests arrive, or where the client, portfolio, and market data resides. The dynamic application must be "triggered" by external events or internal timers or options. Code must be created to start and then run the application. To execute the application, the data sources must be defined during deployment.

Setting Connections and Starting the Application

Several interaction descriptors communicate with external processes and require Connection descriptors. For this example, assume that all requests come over a TIB/RV bus and a database supplies the data.

Define the following connection using the Knowledge Broker GUI:

A TIB/RV connection with the name `tibcoCxn`. TIB/RV requires no connection properties but supports a relatively large set of connection properties. All these properties can be initialized with their default values. This connection descriptor is added to the model file. The Connection descriptor here defines a named `tibcoCxn` connection on the localhost at port 7890.

The Connection descriptors are also stored in the `model.xml` file:

```
<?xml version="1.0"?>
<model>
    ...
    <!-- APPLICATION: RTC TIB Connection -->
        <connectionDescriptor>
        <name>tibcoCxn</name>
        <url>127.0.0.1</url>
        <service>7890</service>
        <daemon>null</daemon>
        <refreshTimeout>-1</refreshTimeout>
        <connectorName>TIBCO</connectorName>
        <schema>mesgSchemaCxn</schema>
    </connectionDescriptor>

    <!-- APPLICATION: Sends out a test message -->
    <connectionDescriptor>
        <name>testMessageFile</name>
        <connectorName>FILE</connectorName>
        <url>e:/demos/rtc/rtcapp/testMessage.txt</url>
        <schema>mesgSchemaCxn</schema>
    </connectionDescriptor>
    ...
</model>
```

Next, assign the newly created connection descriptor to the interaction descriptors. Again, use the Knowledge Broker GUI to update the interaction descriptors.

Index

A

B

C

D

E

F

G

H

I

J

K

M

N

O

P

Q

R

S

T

U

X