WebSphere Business Integration Server
Express and Express Plus

Business Object Development Guide

IBM

14May2004

This edition of this document applies to IBM WebSphere Business Integration Server Express version 4.3, IBM Business Integration Server Express Plus, version 4.3, and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, e-mail doc-comments@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Chapter 11. BusObjAttrType interface 197

Chapter 12. BusObjDef class. 199

Chapter 13. BusObjVerb class 211

Chapter 14. CompleteCondition class 215

Chapter 15. ContentMetaData class 219

About this document

The products IBM ®WebSphere Business Integration Server Express and IBM ®WebSphere Business Integration Server Express Plus are made up of the following components: InterChange Server Express, the associated Toolset Express, CollaborationFoundation, and a set of software integration adapters. The tools in the Toolset help you to create, modify, and manage business processes. You can choose from among the prepackaged adapters for your business processes that span applications. The standard processes template--CollaborationFoundation-- allows you to quickly crate customized processes.

This document describes how to use Business Object Designer Express to create business object definitions. Before you start, you should be familiar with all the concepts explained in the System Implementation Guide.

Except where noted, all the information in this guide applies to both IBM WebSphere Business Integration Server Express and IBM WebSphere Business Integration Server Express Plus. The term WebSphere Business Integration Server Express and its variants refer to both products.

Audience

This document is for IBM customers, consultants, or resellers who create or modify business objects. Before you start, you should be familiar with all the concepts explained in the System Implementation Guide.

Related documents

The complete set of documentation available with this product describes the features and components common to all WebSphere Business Integration Server Express installations, and includes reference material on specific components.

You can download, install, and view the documentation at the following site: http://www.ibm.com/websphere/wbiserverexpress/infocenter

Note: Important information about this product may be available in Technical Support Technotes and Flashes issued after this document was published. These can be found on the WebSphere Business Integration Support Web site, http://www.ibm.com/software/integration/websphere/support/. Select the component area of interest and browse the Technotes and Flashes sections.

Typographic conventions

This document uses the following conventions:

courier font	Indicates a literal value, such as a command name, file name, information that you type, or information that the system prints on the screen.
italic, italic	Indicates a new term the first time that it appears, a variable name, or a cross-reference.
bold	Indicates a GUI element.

blue outline	A blue outline, which is visible only when you view the manual online, indicates a cross-reference hyperlink. Click inside the outline to jump to the object of the reference.
{ }	In a syntax line, curly braces surround a set of options from which you must choose one and only one.
[ ]	In a syntax line, square brackets surround an optional parameter.
...	In a syntax line, ellipses indicate a repetition of the previous parameter. For example, `option[,...]` means that you can enter multiple, comma-separated options.
< >	In a naming convention, angle brackets surround individual elements of a name to distinguish them from each other, as in `<server_name><connector_name>tmp.log`.
ProductDir	Represents the directory where the product is installed. For the IBM WebSphere InterChange Server Express environment, the default product directory is \WebSphereServer for the IBM WebSphere Business Integration Adapters environment.
`%text%` and `$text`	Text within percent (%) signs indicates the value of the Windows `text` system variable or user variable.

New in this release

This is the first release of this guide.

New in Business Object Designer Express

This guide documents the following new features of Business Object Designer Express:

- Improved steps for opening a business object definition from a project and deleting a business object definition from a project.
- Enhanced Business Object Wizard screens for steps 3 through 6.
- Process Designer Express can be started from the **Tools** menu of Business Object Designer Express.

Part 1. Designing and developing business objects

Chapter 1. Business objects

A business integration system uses business objects to carry data and processing instructions between an integration broker and connectors or an access client. InterChange Server Express is the integration broker in WebSphere Business Integration Server Express. Business objects represent a request from InterChange Server Express, an event in an application or Web server, or a call from an external site. This manual presents information on developing and designing business objects, as well as developing your own object discovery agent. The main topics in this chapter are:

- "Business objects in the WebSphere business integration system"
- "Business object structure" on page 11
- "Overview of the development process" on page 13

This chapter assumes that you have a basic understanding of InterChange Server Express in your environment. Table 1 references documentation for InterChange Server Express:

Table 1. Prerequisite documents

Integration broker	Prerequisite documents
InterChange Server Express	- *WebSphere InterChange Server Express Installation Guide* - *System Implementation Guide*

Business objects in the WebSphere business integration system

The WebSphere business integration system consists of the following components:

- A set of adapters

  An *adapter* is a set of software modules that communicate with InterChange Server Express and with applications or technologies to perform tasks such as running application logic and exchanging data.

- An integration broker

  InterChange Server Express is the integration broker for the WebSphere business integration system. The task of InterChange Server Express is to integrate data among heterogeneous applications.

In the WebSphere business integration system, information sent or received between components is packaged in the form of a *business object*, as follows:

- For data that is transferred between an adapter and InterChange Server Express, you design *application-specific business objects* that model the appropriate application entities.

- For data that is processed within the business logic of InterChange Server Express collaboration object, you design *generic business objects* that contain a superset of information for the application entities that need to communicate. Maps transform data between generic business objects and application-specific business objects so that adapters can communicate with their applications using application-specific entities, while collaboration objects can apply business logic in an application-independent way.

Both application-specific business objects and generic objects are modeled during design-time as *business object definitions*, which are stored in the business integration system. At run time, data is populated in a *business object instance* (often called a "business object"), which is based on the appropriate definition. The business object moves through the business integration system as dictated by its routing and business logic rules.

Business object definitions

A *business object definition* represents a template for data that can be treated as a collective unit. It contains a business object header, which specifies the name and version of the business object definition. In addition, the business object definition contains the following information:

- "Business object attributes and attribute properties"
- "Business object verbs" on page 7
- "Business object application-specific information" on page 7

Figure 1 shows the parts of a business object definition.

```
Business object name
Version
AppSpecificInfo                    ─── Business object
                                        header

Attribute1
   Type
   AppSpecificInfo=
   ...
Attribute2
   Type
   AppSpecificInfo=
   ...
      o
      o
Attributen                          ─── Business object
   Type                                 attributes
   AppSpecificInfo=
   ...

Verb1
   AppSpecificInfo=
      o
      o                              ─── Supported
Verbn                                    verbs
   AppSpecificInfo=
```
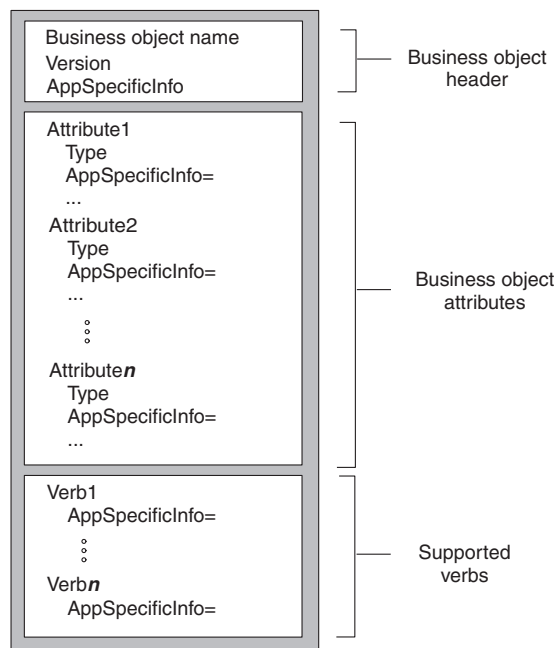
Figure 1. Business object definition parts

Business object attributes and attribute properties

A business object contains attributes, each *attribute* representing one entity of data. In the business object definition, you define the name of each attribute as well as other *attribute properties*. The business object instance holds a value for each attribute (or indicates that the attribute does not have a value).

Business object definitions include various properties that apply to attributes. These properties provide the connector, data handler, and other components with information on the types, sizes, and default values of attributes. The attribute properties are discussed in the sections that follow.

Name property: Each business object attribute must have a unique name within the business object definition. The name should describe the data that the attribute contains. The name can be up to 80 characters; it can contain alphanumeric characters and underscores but cannot contain spaces, punctuation, or special characters.

Notes:

1. When designing an application-specific business object, check its adapter user guide or the *Data Handler Guide* for specific naming requirements and recommendations.

2. This attribute name must use *only* characters defined in the code set associated with the U.S. English locale (`en_US`).

Type property: The Type property defines the data type of the attribute:

- For a simple attribute, the supported types are `Boolean`, `Integer`, `Float`, `Double`, `String`, `Date`, and `LongText`.
- For a complex attribute, the type is a business object definition:
  - If the attribute represents a child business object, specify its type as the name of the child business object definition and specify the cardinality as 1 (single cardinality).
  - If the attribute represents an array of child business objects, specify the type as the name of the child business object definition and specify the cardinality as `n` (multiple cardinality).

Note: All attributes that represent child business objects also have a `ContainedObjectVersion` property (which specifies the version number of the child object's business object definition) and a `Relationship` property (which specifies the value `Containment`).

Cardinality property: Each simple attribute has single cardinality (cardinality 1). Each complex attribute, which represents a child business object or array of child business objects, has single cardinality or multiple cardinality (cardinality n), respectively. For more information on cardinality, see "Hierarchical business objects" on page 11.

Note: When specified for a required attribute, single cardinality indicates that a child business object *must* exist, and multiple cardinality indicates zero to many instances of a child business object.

Key property: At least one attribute in each business object must be specified as the key. The key attribute contains a value that uniquely identifies the business object. To define an attribute as a key, set its Key property to `true`.

Note: A key value in a business object is often referred to as its *primary key*.

When you specify as key a complex attribute:

- If the attribute represents a child business object, the key is the concatenation of the keys in the child business object.
- If the attribute represents an array of child business objects, the key is the concatenation of the keys in the child business object at location 0 in the array.

Foreign key property: The Foreign Key property is typically used in application-specific business objects to specify that the value of an attribute holds the primary key of another business object, which links business objects. The

attribute that holds the primary key of another business object is called a *foreign key*. Define the Foreign Key property as `true` for each attribute that represents a foreign key.

You can also use the Foreign Key property for other processing instructions. For example, this property can be used to specify what kind of foreign key lookup the connector performs. In this case, you might set Foreign Key to `true` to instruct the connector to check for the existence of the entity in the database and create the relationship only if the record for the entity exists.

Required property: The Required property specifies whether an attribute must contain a value. If a particular attribute in the business object must contain a value to be able to process the business object data, set the Required property for the attribute to `true`.

AppSpecificInfo: The AppSpecificInfo property can contain a `String` of up to 1000 characters that is specified primarily for an application-specific business object. For information on this property, see "Business object application-specific information" on page 7.

Note: Application-specific information is *not* available in the mapping process.

Max Length property: The Max Length property is set to the number of bytes that a `String`-type attribute can contain. Although this value is not enforced by the WebSphere business integration system, specific connectors or data handlers might use this value. Check the guide for the specific adapter or the guide for the data handler that processes the business object to determine minimum and maximum allowed lengths.

Important: The Max Length property is very important when you use a fixed-width data handler.

Note: Attribute length is *not* available in the mapping process.

Default value property: The Default Value property can specify a default value for an attribute.

If this property is specified for an application-specific business object, and the `UseDefaults` connector configuration property is set to `true`, the connector can use the default values specified in the business object definition to provide values for attributes that have no values at run time.

Notes:
1. The attribute's default value can use any characters defined in the code set associated with the current locale.
2. For an attribute whose type is String, you can specify a blank character as a default value.

Comments property: The Comments property allows you to specify a comment for an attribute. Unlike the AppSpecificInfo property, which is used to process a business object, the Comments property provides only documentation information, which may assist other developers in understanding your design decisions.

Note: The attribute's comments can use any characters defined in the code set associated with the current locale. However, the newline character is invalid.

ObjectEventId attribute: The `ObjectEventId` attribute is not only required, but it must be the last attribute in every business object. The WebSphere business integration system uses this attribute to identify and track an event flow in the system.

The `ObjectEventId` attribute stores a unique value that identifies each event in the WebSphere business integration system.The connector framework generates values for this attribute in the parent business object and in each child.

Important: Do *not* map the `ObjectEventId` attribute or have a connector or data handler populate it. The business integration system handles the value of this attribute.

Business object verbs

The business object definition includes a list of the verbs that the business object can support. These verbs correspond to operations that are valid on the data within the business object. At run time, a business object contains one active verb, which describes the operation to perform on the data in that particular business object.

Table 2 lists the basic verbs that a business object definition can support.

Table 2. Basic verbs

Verb	Function
Create	Make a new entity in the application.
Retrieve	Using key values, return a complete business object.
Update	Change the value in one or more fields in the application entity.
Delete	Remove the entity from the application. This operation must be a true physical delete.

In addition to the basic verbs in Table 2, a business object definition might also need to support one or more of the following verbs:

- RetrieveByContent—Using non-key values, return a complete business object.
- Exist—Check for the existence of a specified entity but do not retrieve it.
- Custom—Perform an application-specific operation.

Business object application-specific information

A business object definition can provide *application-specific information*, whose content provides *metadata* to the component that processes the business object. A common use of application-specific information is to provide a connector or data handler with application-dependent instructions on how to process the business object. The application-specific information is a string that is entered during business object design and read at run time by a connector or data handler.

Note: Connectors that are designed to use the application-specific information in definitions of their application-specific business objects are called *metadata-driven connectors*. Because the processing information is configurable, rather than hard-coded, a metadata-driven connector is more flexible and easier to maintain than one that is not metadata-driven.

Within a business object definition, you can provide application-specific information at one of three levels:

- The business object definition

- An attribute within the business object definition
- The business object verb

Application-specific information is stored in a field in the business object definition called the AppSpecificInfo property. The value of the AppSpecificInfo property is a text string that can include any information about the business object or application. Figure 2 illustrates the major elements of a business object definition and the application-specific property for each element.
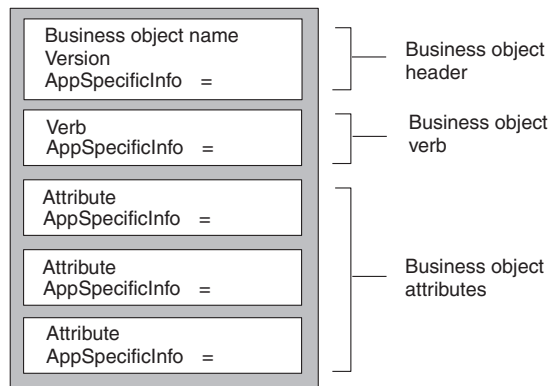
Figure 2. Business object definition showing the application-specific property for each element

This section covers the following topics:
- Application-specific information for a business object
- Application-specific information for an attribute
- Application-specific information for a verb

Application-specific information for a business object: The application-specific information at the business object level provides information that the connector or data handler uses to process the data. Business object-level application-specific information is used whenever processing instructions are relevant for an entire business object hierarchy. For example, the object-level application-specific information might do one or more of the following:
- Define the scope of business object transaction processing
- For applications that require object processing in an application extension, contain the name of the function to call to handle the business object
- Specify the name of the table or form in which the record belongs
- Specify the name of an attribute within the business object that represents a logical or "soft" delete

Figure 3 illustrates application-specific information that identifies a form or table name in an application. The connector can get the table or form name from the AppSpecificInfo property and use it in an API call to retrieve data from the application.
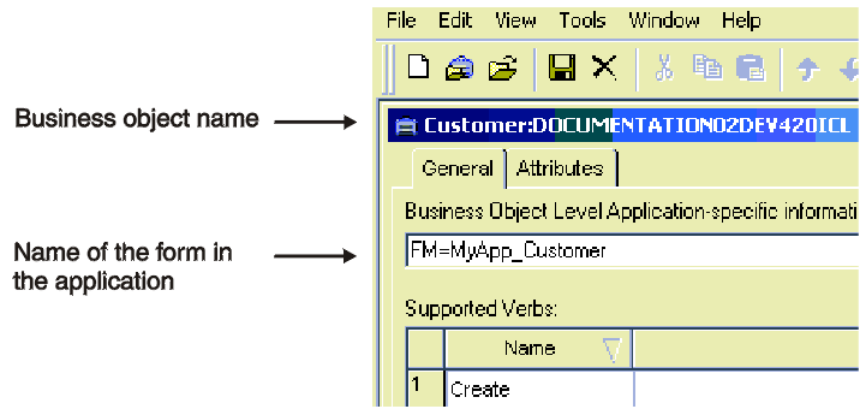
Figure 3. Application-specific information for a business object

Application-specific information for an attribute: Each attribute of a business object definition can have application-specific information associated with it. Attribute-level application-specific information is used whenever processing instructions are relevant for the single attribute. For example, this information can specify a field on a form, a column in a table, or whatever the connector needs to locate or work with the attribute. If certain attributes of a business object are located on a particular subform in the application, the AppSpecificInfo property is a good candidate for a place to encode this information.

Figure 4 illustrates the AppSpecificInfo property for an attribute. In this example, the application-specific information specifies the name of a subform and field.
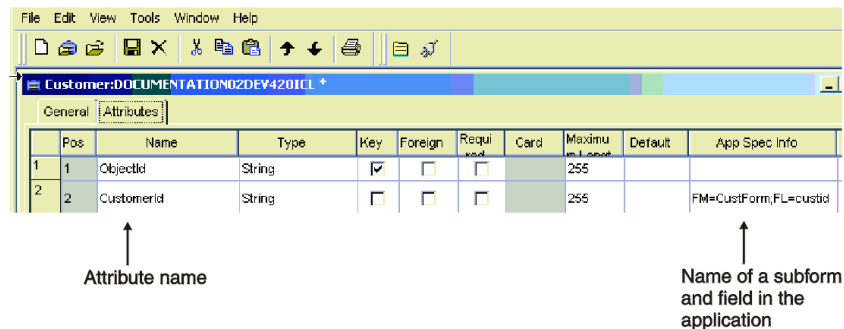
Figure 4. Application-specific information for an attribute.

Figure 5 illustrates the relationship of form, subform, and field name as provided in the object-level and attribute-level application-specific information. This example assumes that a billing application is based on forms, and that the way to interact with invoices in this application is through the Invoice form, which is a subform of a main CustAccount form. The Invoice subform has the following fields: CustName, CustAddr, InvNum, DollarAmount, and Terms.
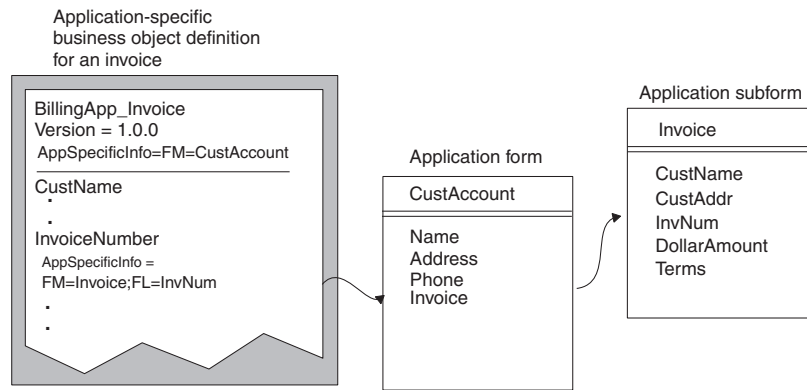
Figure 5. Using business object definition application-specific information

Figure 5 uses the attribute-level AppSpecificInfo property to store the name of the Invoice subform and the attribute's corresponding field name. The example uses name-value pairs to specify the information.

Application-specific information for a verb: Each verb definition can include application-specific information that provides the connector or data handler with instructions on how to process the business object when that verb is active.

Note: The business object handler, which is the part of a connector that handles requests sent from InterChange Server Express to the connector, can be designed to use the application-specific information in the verbs of their application-specific business object definitions. Such business object handlers are called *metadata-driven business object handlers*. Because the processing information is configurable, rather than hard-coded, a metadata-driven business object handler is more flexible and easier to maintain than one that is not metadata-driven.

For example, if the connector is using an API to handle updates to the application database, the application-specific information can provide the connector with information to run an API.

The verb application-specific information can also specify the name of a function to call in the application to handle the processing of a business object.

Business object instances

While the business object definition represents the template for a collection of data, a business object instance (often just called a "*business object*") is the run-time entity that contains the actual data. The business object is what is passed between components of the business integration system.

The business object contains the following information:

- Attributes, each of which contains the data for the associated business object. One of the attributes is usually a key attribute, which contains a value that uniquely identifies this business object among all business objects of the same definition.
- An active verb, which should be one of the supported verbs for the business object definition

Figure 6 shows the `Customer` business object definition and a corresponding business object instance for this definition.

Business object definition   Business object

Customer		Customer
CustomerId	→	8776
CustomerName	→	Trievers Inc
CustomerStatus	→	Active
CustomerRegion	→	NE
ObjectEventId	→	1027111552889_1
Create	→	Create
Retrieve		
Update		
Delete		
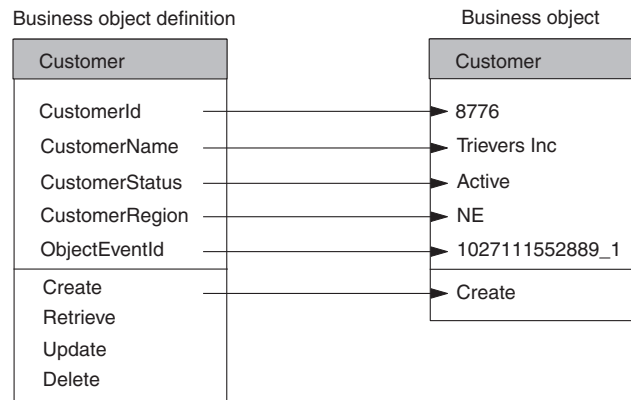
Figure 6. Business object definition and sample business object.

Business object structure

The structure of a business object can be either of the following:

- Flat business objects
- Hierarchical business objects

The following sections show examples of flat and hierarchical business object structures, and provide information on how the business object structure affects connector logic.

Flat business objects

A business object definition for a *flat business object* contains one or more simple attributes and a list of supported verbs. A *simple attribute* represents one value, such as a `String` or `Integer` or `Date`. All simple attributes have single cardinality. The Customer business object in Figure 6 is an example of a flat business object. For more information, see "Business object attributes and attribute properties" on page 4.

Hierarchical business objects

Hierarchical business object definitions define the structure of multiple related entities, encapsulating not only each individual entity but also aspects of the relationship between entities. In addition to containing at least one simple attribute, a hierarchical business object has one or more attributes that are *complex*; that is, the attribute itself contains one or more business objects, called *child business objects*. The business object that contains the complex attribute is called the *parent business object*.

There are two types of relationships between parent and child business objects:

- *Single cardinality*—When an attribute in a parent business object represents a *single* child business object. The type of the attribute is set to the name of the child business object, and the cardinality is set to 1.
- *Multiple cardinality*—When an attribute in the parent business object represents an *array* of child business objects. The type of the attribute is set to the name of the child business object, and the cardinality is set to `n`.

In turn, each child business object can contain attributes that contain a child business object or an array of business objects, and so on. The business object at the top of the hierarchy, which itself does not have a parent, is called the *top-level business object*. Any single business object, independent of its child business objects it might contain (or that might contain it) is called an *individual business object*.

In a typical business object hierarchy, a top-level business object definition contains one or more simple attributes, one or more attributes that represent a child or array of child business objects, and a list of supported verbs. Figure 7 shows a typical hierarchical business object. The top-level business object, Customer, has both single-cardinality attributes and multiple-cardinality attributes with child business objects:

- Its Address attribute is a complex attribute with multiple cardinality. Customer is the parent business object for each of the Address child business objects.
- Its CustProfile attribute is a complex attribute with single cardinality. Customer is the parent business object for the single CustProfile child business object.
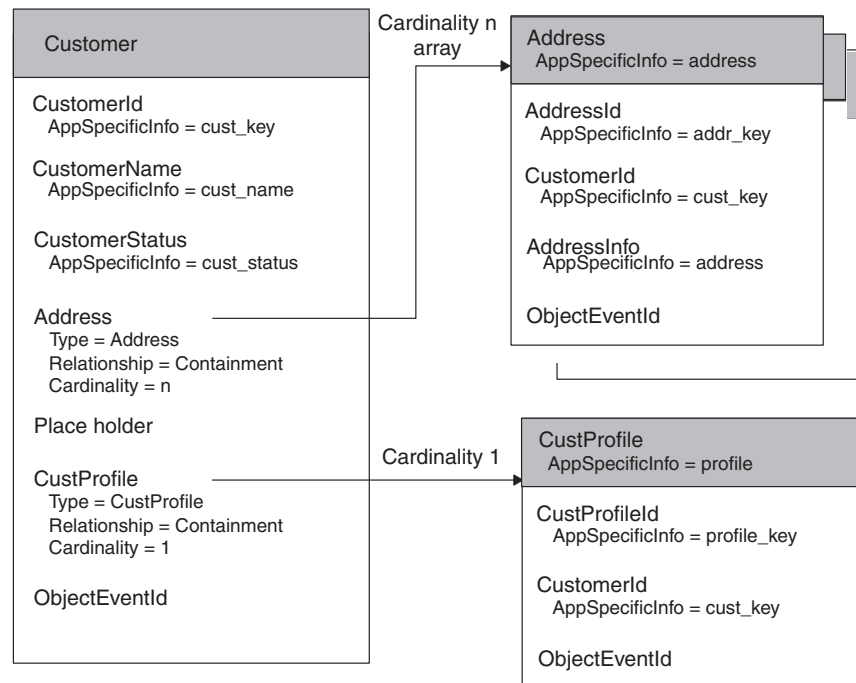
Figure 7. Example of a hierarchical business object definition

In Figure 7, the Customer and CustProfile business objects, as well as each of the Address business objects is an individual business object.

Note: When a top-level business object contains information used to process its child business objects, it is called a wrapper business object. For example, the XML connector requires a wrapper business object to contain information that determines the format of its child data business objects and routes the children.

When designing the structure of a hierarchical application-specific business object, you need to determine:

- How entity data is represented in the business object
- How the primary application entity relates to child entities

- If an application entity includes data from different entities, you must decide:
  - Whether the application-specific business object needs to include related data
  - How to define the relationship between the related data

For more information, see "Design considerations for multiple entities" on page 25.

Overview of the development process

This section provides an overview of the business object development process.

Setting up the development environment

Before you start the development process, the following must be true:

- The WebSphere business integration system is installed on a computer that you can access.

  For information on how to install and start up the WebSphere business integration system, refer to the Installation Guide for WebSphere Business Integration Server Express

- InterChange Server Express and its repository's database server are running. This step is required only when you are ready to save the definition to the repository or to delete a definition from the repository. For development only, you can run Business Object Designer Express locally, without connecting to InterChange Server Express.

- If you plan to generate a business object definition using an Object Discovery Agent, you must have started the Object Activation Daemon (OAD) before you use this ODA to generate business object definitions. For more information, see "Before using an ODA" on page 64.

Stages of business object development

The stages of business object development are as follows:

1. Understand the data requirements that are critical to the business process integration.
   - If creating an application-specific business object, understand the relationship among the connector, the data handler, and the supported application-specific business objects.
   - If creating a generic business object for use with the InterChange Server, understand the relationship between the collaboration object and the business object.

2. Develop the business object definitions in one of two ways:
   a. Generation from a data source—the WebSphere business integration system provides tools that facilitate generation of a business object definition for some connectors. Such tools are Object Discovery Agents or command line tools that are designed to connect to an application and "discover" business object requirements specific to a business entity and to generate definitions from those requirements. Business Object Designer Express presents a graphical user interface to Object Discovery Agents, and helps manage the discovery and definition generation processes. Check the guides for the adapter and data handler you will be using to determine if they have an available tool or utility. You can also check the Connector Feature Checklist, which is available on the main documentation page under the Connectors category. If a custom adapter is being developed to communicate with an application, you can use the Object Discovery Agent Development Kit to create a custom Object Discovery Agent for the adapter.

b. Manual—Business Object Designer Express is a graphical user interface that facilitates the manual creation of business object definitions. This interface is most useful for developing generic business objects to use with InterChange Server Express, as there is no application in which object discovery can be performed.

3. If you used a tool to automatically generate the business object definition from a data source, verify that the generated structure and application-specific information conforms to requirements. Reference the adapter guide for the connector that uses the business object definition determine any special configuring that you must do manually.

4. Test and debug the business object by running it through the system; edit it as necessary.

The following illustration shows a visual overview of the business object development process and provides a quick reference to chapters where you can find information on specific topics.

Task:	Steps:	Refer to:

Designing business objects
→
- Identify data requirements
- Understanding the relationship among connector, data handler, and application-specific business object
- If using InterChange Server Express, understand the relationship between the collaboration and the generic business object

— Chapter 2

Understanding Business Object Designer Express
→
- Launching Business Object Designer Express
- Using Business Object Designer Express
- Working locally or connected

— Chapter 3

Working with business object definitions
→
- Creating a business object definition manually
- Creating a business object definition using an Object Discovery Agent (ODA)
- Deleting a business object definition

— Chapter 4

Chapter 5
Chapter 6
Chapter 7

Creating an Object Discovery Agent
→
- Understanding the application and its requirements
- Learning about the structure of an ODA
- Understanding the relationship between the ODA and Business Object Designer Express
- Learning about ODK classes

Chapter 8
Chapter 9
Chapter 10
Chapter 11
Chapter 12
Chapter 13
Chapter 14
Chapter 15
Chapter 16

Chapter 2. Business object design

The key to the design of business objects is to develop a business object definition that models as closely (and efficiently) as possible the data that needs to be transmitted between components of the business integration system:

- For data that is transferred between a connector and an InterChange Server Express, you design *application-specific business objects* that model the appropriate application entities. These entities might correspond to data structures or technology standards used by a particular application, or to specific technology standards used by a web server.
- For data that is processed within the business logic of an InterChange Server Express collaboration object, you design *generic business objects* that contain a superset of information for the application entities that need to communicate. When the collaboration object exchanges information with an application, maps convert the data between the generic business object and application-specific business object structures.

This chapter presents information on the structure of business objects for the WebSphere business integration system, and makes recommendations for designing both application-specific and generic business objects. The material presented here assumes that you understand the basic object concepts described in the *System Implementation Guide*.

The main topics of this chapter are:

- "Determining business object structure"
- "Designing application-specific business objects" on page 29
- "Designing generic business objects (InterChange Server Express only)" on page 37
- "Determining mapping requirements for business objects (InterChange Server Express only)" on page 41

Determining business object structure

The purpose of a business object is to transport data between components of a business integration system and the applications that it integrates. Therefore, the business object should model the data that needs to be transported. This data is usually associated with an entity in an application or technology that the business integration system integrates. The structure of a business object can be either of the following:

- "Representing one entity"
- "Representing multiple entities" on page 18

In addition, this section provides "Design considerations for multiple entities" on page 25.

Representing one entity

The simplest business object design is a flat business object that represents one entity. All the attributes of a flat business object are simple (that is, each attribute represents one value, such as a `String` or `Integer` or `Date`). For more information, see "Flat business objects" on page 11.

In the case of an application-specific business object, a flat business object can represent one entity in an application or in a technology standard. For example, assume an application has a database table that describes a record. Assume further that this table has five columns named ObjectID, UserName, TimeStamp, Detail, and Status (see Figure 8). The ObjectID is the primary key for each row, and its value is generated by the application. This table has no relationships to other tables.

Application table

ObjectID	UserName	TimeStamp	Detail	Status

Business object definition

Record

ObjectID
UserName
TimeStamp
DetailText
Status
ObjectEventId
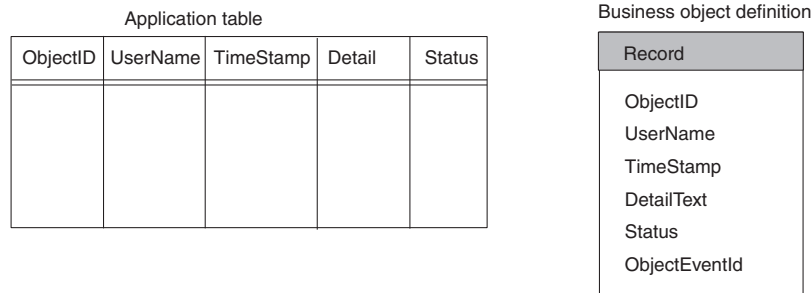
Figure 8. Flat business object representing one entity

As Figure 8 shows, the Record business object you design to represent the table might have five attributes, one for each column, with the key attribute corresponding to the ObjectID column.

Use of flat business objects can simplify corresponding connector design in the following ways:

- On a Create operation, the connector might cycle through the attributes, extracting the non-key attribute values from the business object instance and extracting processing instructions from the business object definition. When it has assembled the information it needs to process the business object, the connector might start an application function call or SQL statement to create a new row for the record in the table. The connector then returns a value for the key to the business integration system.

- On a Retrieve operation, the connector might extract the primary key from the business object request, use the key value to retrieve the current set of data for the row, and return a business object with the complete set of values.

This type of business object is straightforward in its design and in the connector logic required to process it. Typically, however, application entities are more complex and include information that is stored in other objects.

Representing multiple entities

A business object can represent application entities that include data from other entities in one of the ways shown in Table 3.

Table 3. Representing multiple entities.

Structure of business object	Type of data organization	Type of parent/child relationship
Parent business object can have one or more child business objects that represent the other entities.	One-to-one One-to-many	Structural

Table 3. Representing multiple entities. (continued)

Structure of business object	Type of data organization	Type of parent/child relationship
Parent business objects can have one or more foreign-key attributes that reference other top-level business objects that represent the other entities.	One-to-one One-to-many Many-to-many Many-to-one	Semantic
If the application and its interface permit, a flat business object can include attributes that directly reference other entities.	One-to-one	None

When deciding how to structure business objects that represent multiple entities, consider these guidelines:

- If the relationship between entities is a one-to-many relationship, represent the data in the subordinate entities as child business objects. For example:
  - When working with database tables, if an entity row is related to one or more rows in another entity and is the only entity that relates to the subordinate entity, create a separate child business object for each related row.
  - When working with a DTD, if an XML element has an attribute with a cardinality of *, create a separate child business object for each related element attribute.
- If the relationship between entities is a many-to-many relationship, represent the data in the related entities as top-level business objects that are referenced by the parent rather than contained by the parent.
- If a business object definition for an entity includes many attributes from another entity and the attributes from the second entity form a logical grouping, you may want to create a child business object definition for the second entity rather than locate all the attributes for both entities in the same business object definition.
- If an existing business object already contains other child business objects, creating one or more child business objects that represent new entities makes the business object structure consistent.

The following sections describe each of these representations in more detail.

Structural relationships

In a structural relationship, the parent business object physically contains the child business object. Such a business object is a hierarchical business object: at least one of its attributes is complex (that is, it contains either a child business object or an array of child business objects). The Relationship attribute property for this attribute is containment, to indicate a containment relationship. The type of this attribute is the type of the child business object (or objects) it represents. For more information, see "Hierarchical business objects" on page 11.

The following hierarchical business objects represent structural relationships:

- Because an order is composed of line items, an Order business object contains an array of LineItem business objects. The containment relationship has multiple-cardinality because each order can contain multiple line items. This structure represents a one-to-many relationship.
- Because an employee is associated with one home address, an Employee business object contains one Address business object. The containment

relationship has single-cardinality because each employee can be associated with only one home address. This structure represents a one-to-one relationship.

In both cases, because the parent business object contains the child or array of children, the relationship is defined structurally.

A structural relationship assumes that the parent business object owns the data within the child object. Thus, when a new employee is created, a new row is inserted into the address table to hold that employee's address. Similarly, when an employee is deleted, the employee's address is also deleted from the address table.

Semantic relationships

In a semantic relationship, either the parent business object references the child, or the child references the parent. When one business object references another, it stores a value that uniquely identifies the other, but it does not contain the other. In this case, the component that processes the business object derives the relationship semantically.

A semantic relationship is typically defined by a simple attribute that serves as a *foreign key*. The foreign key attribute is located in one business object and contains the unique identifier (called the *primary key*) of the other. In other words, both business objects have a primary key attribute that holds its unique identifier. In addition, one of the business objects also has a foreign key attribute that holds the primary key value of the other. The foreign key establishes the link semantically between parent and child.

Semantic relationships are important when there is a many-to-many or many-to-one relationship between entities, in other words, when more than one parent has a relationship to the same child. Relating the entities semantically rather than structurally isolates the child's data, which is important to maintain data consistency.

Because the parent does not contain the child in a semantically defined relationship, the connector that handles requests for the parent and child receives them in separate operations. In other words, the requests are sent separately to the connector, which handles the parent and child in separate operations. For more information, see "Data ownership in relationships" on page 25 and "Choosing between a semantic and a structural relationship" on page 27.

Consider the design options in Table 4 for specifying a semantic relationship.

Table 4. Design options for semantic relationships.

Design option	Type of relationship
"Storing the foreign key in the parent object"	One-to-one Many-to-one
"Storing the foreign key in the child object" on page 21	One-to-many
"Storing foreign keys in an array of child objects" on page 22	One-to-many Many-to-many
"Storing the foreign key in a business-object tree" on page 23	One-to-one

Storing the foreign key in the parent object: In the simplest use of foreign keys, the foreign key that establishes the relationship is stored in the parent. In this case, a parent can contain a reference to only one child of a given type. The relationship

between parent and child is clearly defined in the parent. Therefore, this structure represents a one-to-one relationship. However, multiple parent business objects can reference the same child business object to implement a many-to-one relationship.

Note: When the foreign key that establishes the relationship is stored in the parent, a parent can contain multiple attributes that each contain a reference to a child, but each of these attributes typically references a different type of child.

In Figure 9, the Customer business object has two attributes (`AddressId` and `CustInfo`) each of which contain a reference to a child business object. The foreign key attributes in Customer readily identify the parent's relationship to the two children.

Customer business object

Customer
ObjectId (PK)
CustomerName
AddressId (FK)
CustInfo (FK)

Address business object

Address
ObjectId (PK)
City
State

Customer information business object

CustomerInformation
ObjectId (PK)
CustomerProfileData
CustomerAccountData
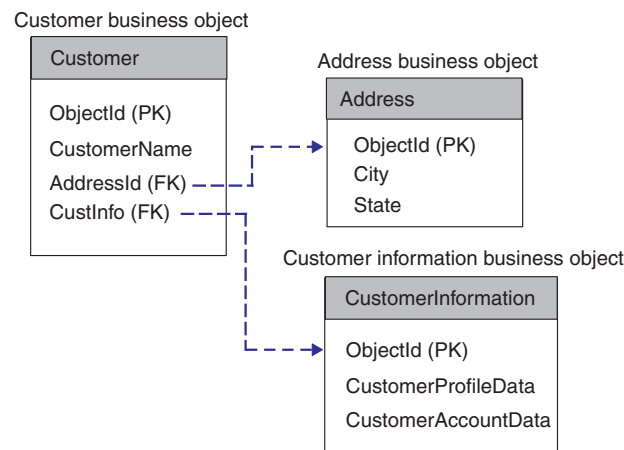
Figure 9. One-to-one: Multiple foreign key attributes stored in the parent business object

Note: In Figure 9, the acronym "PK" is used to indicate a primary key and "FK" is used to indicate a foreign key. In addition, these business objects follow the naming convention for generic business objects by naming their primary key attribute `ObjectId`. In an application-specific business object, it is usually best to name the attribute after the name of its equivalent field or column in the application.

Using InterChange Server Express, you can examine the delivered generic Order business object for an example of a parent object that stores a foreign key reference to another object. It contains the CustomerId attribute, which references the top-level generic Customer business object. See Figure 11 for an illustration of the Order business object.

Storing the foreign key in the child object: Alternatively, the foreign key that establishes the relationship can be stored in the child. This case represents a one-to-many relationship; that is, multiple children can reference the same parent. However, because the relationship between parent and child is defined in the child, the relationship is invisible when you examine only the parent. Therefore, if the parent business object triggers an integration flow, those children cannot be retrieved—there is no reference to them in the parent business object that is traveling through the system.

In Figure 10, the foreign key attribute is stored in each child business object rather than in the parent. This structure allows multiple children to be semantically related to the same parent. In this case, however, because the parent business

object has no attributes that contain a reference to a child business object, there is no way to identify the parent's relationship to its children or, given the parent, to retrieve all of its related children.
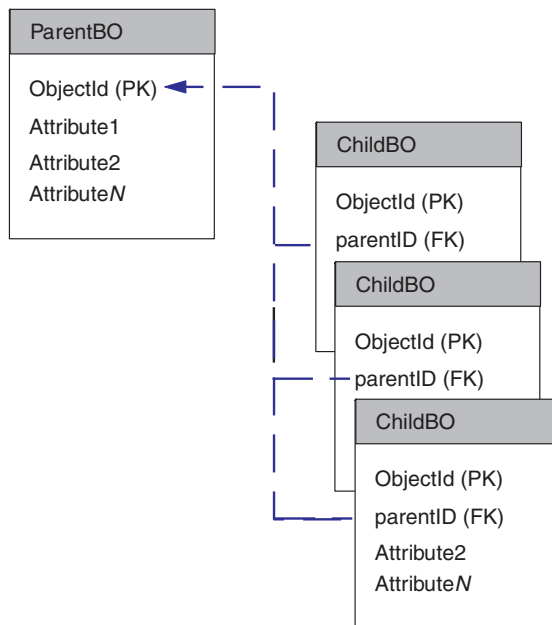
Figure 10. Many-to-one: Foreign key stored in multiple child business objects

Note: In Figure 10, the acronym "PK" is used to indicate a primary key and "FK" is used to indicate a foreign key.

Storing foreign keys in an array of child objects: To represent a one-to-many relationship, the foreign key that actually establishes the relationship is stored in a simple attribute in a child business object. The parent business object structurally contains an array of these children. In other words, the parent contains an array of child business objects, each one of which contains a foreign-key reference to another top-level business object. In addition, multiple parent business objects can reference the same child business object in their child business object arrays to implement a many-to-many relationship.

Note: With InterChange Server Express, there are several business objects you may examine for an example of a parent-child relationship of this type. The generic Order and ContactRef business objects provide an example of this option. Order contains the OrderContactRef attribute, which contains an array of generic ContactRef business objects. Each ContactRef business object contains the ContactId attribute, which holds a reference to the top-level generic Contact business object.

In Figure 11, the Order business object contains a reference to one Customer business object and structurally contains an array of ContactRef business objects. Each ContactRef business object contains a reference to one Contact business object.
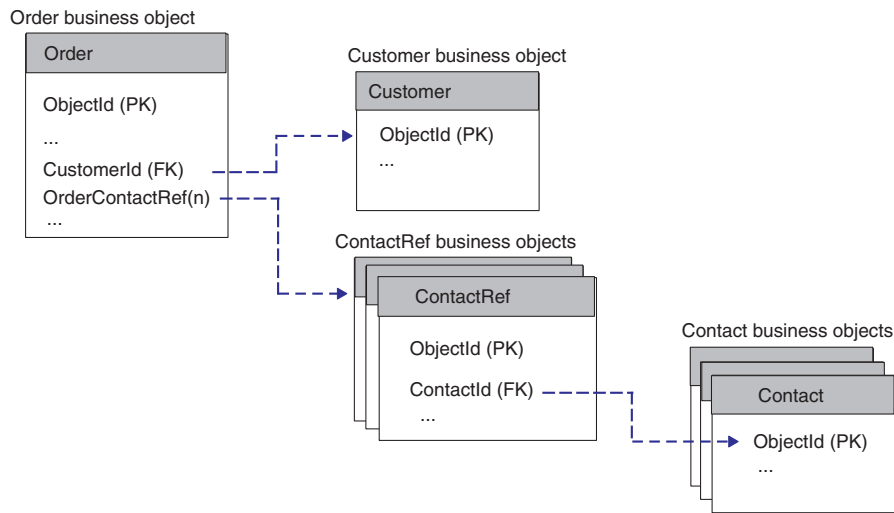
Figure 11. Business object containing a child business object that stores foreign keys

Note: In Figure 11, the acronym "PK" is used to indicate a primary key and "FK" is used to indicate a foreign key.

Storing the foreign key in a business-object tree: In this design, the foreign key that establishes the relationship is stored in a "child" business object whose parent is another business of the same type as itself. With InterChange Server Express, you can examine the generic InstalledProduct business object for an example of this design. This business object contains the ParentId attribute, which can contain a reference to another InstalledProduct business object, which is the direct parent of the current business object.

In Figure 12, the ParentId attribute of one InstalledProduct business object contains a reference to the primary key (ObjectId) attribute of its immediate parent InstalledProduct business object. The head of the hierarchy is the business object whose ParentId attribute does not contain a value.
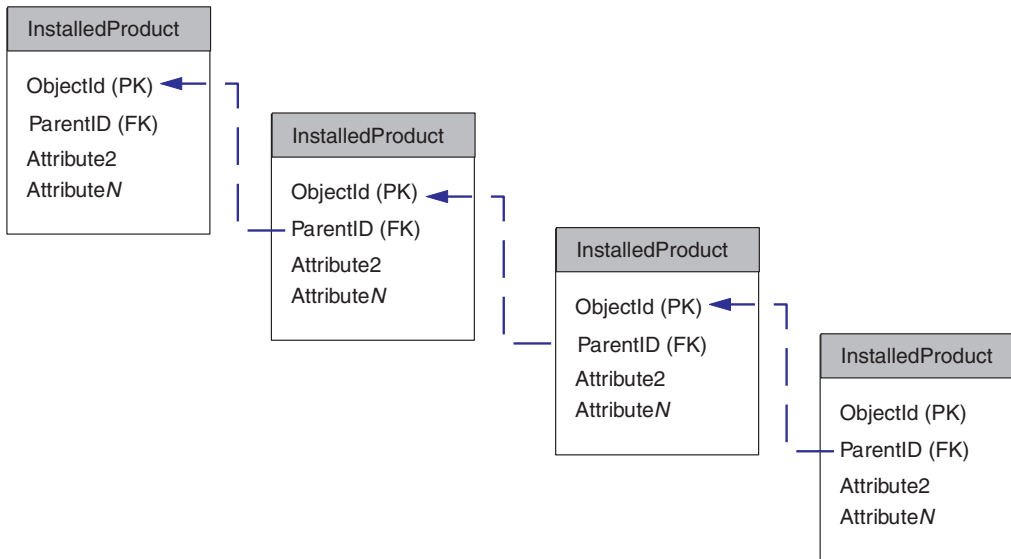
Figure 12. Business object storing a foreign key in its parent of the same type

Note: In Figure 12, the acronym "PK" is used to indicate a primary key and "FK" is used to indicate a foreign key.

Because an InstalledProduct business object can contain a reference to its parent business object, the business integration system can synchronize installed products that are part of a large hierarchy. The business integration system can manage the components of a complex installed product hierarchy as individual InstalledProduct business objects. With InterChange Server Express, you can see the *InstalledProductSync* collaboration template documentation for more information.

Flat business object representing related entities

If the application interface provides the capability of joining multiple application entities in one business object, you may be able to define a flat business object that contains attributes referring to a primary entity and to related entities. If the relationship between the entities is a one-to-one relationship, where one instance of the primary entity can be associated with one instance of each related entity, attributes from multiple entities can be included in one business object.

When designing an application-specific business object of this type, you may need to use application-specific information to specify the location of attribute data in the application so that the connector can find and process the data correctly.

Figure 13 provides an example of a flat WebSphere business integration system business object that represents data in two entities, one a table containing address data and the other a table containing lookup data for state or province and country abbreviations.
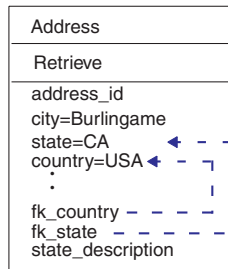
Figure 13. Flat business object that represents two entities

This example uses application-specific information to establish a foreign key relationship between the entities. In this case, the connector performs a lookup from a value in an attribute that represents one table to provide a value for an attribute that represents another table. To retrieve this data, the connector performs two table reads.

Although flat business objects can encapsulate information from or included in multiple application entities, cross-application integration problems often require more complex integration logic and more complicated data structures than flat business objects can represent. To handle more complexity in application entities and integration requirements, the WebSphere business integration system provides hierarchical business objects.

Design considerations for multiple entities

This section provides the following considerations when you design business objects for multiple entities:

- "Data ownership in relationships"
- "Choosing between a semantic and a structural relationship" on page 27

Data ownership in relationships

The way you design your business objects to represent multiple entities has an effect on the ownership of the data:

- A structural relationship assumes that the parent business object owns the child data.
- A semantic relationship does *not* assume that the parent business object owns the data within the child object.

This distinction is significant when considering the data consistency of an entity that is shared by multiple business objects.

For example, assume that a customer and a contact share an address. If the Customer and Contact business objects contain a reference to the Address business object (a semantic relationship) instead of containing the business object (a structural relationship), changes to the Address can be made independently of changes to the Customer or Contact.

However, if the Customer and Contact business objects each contain the Address business object, changes to the Address made by Customer might overwrite changes made by Contact. In this case, two different collaboration objects (CustomerSync and ContactSync) might update the same address data at the same time, causing data inconsistency.

If Customer and Contact have a semantic rather than structural relationship to the Address business object, you can limit modification of Address data to a third interface. For instance, you might have one interface for each of the Contact and Customer business objects. Then both of those interfaces could delegate management of Address business objects to a third interface. In InterChange Server Express this is done by having the CustomerSync and ContactSync collaboration objects call AddressSync through a wrapper collaboration object rather than directly making the changes themselves. For more information on designing business objects to maintain data consistency for InterChange Server Express integration scenarios, see "Designing for Parallel Execution" in the *Collaboration Development Guide*.

Figure 14 illustrates the difference between semantically and structurally defining the relationship to a child business object.
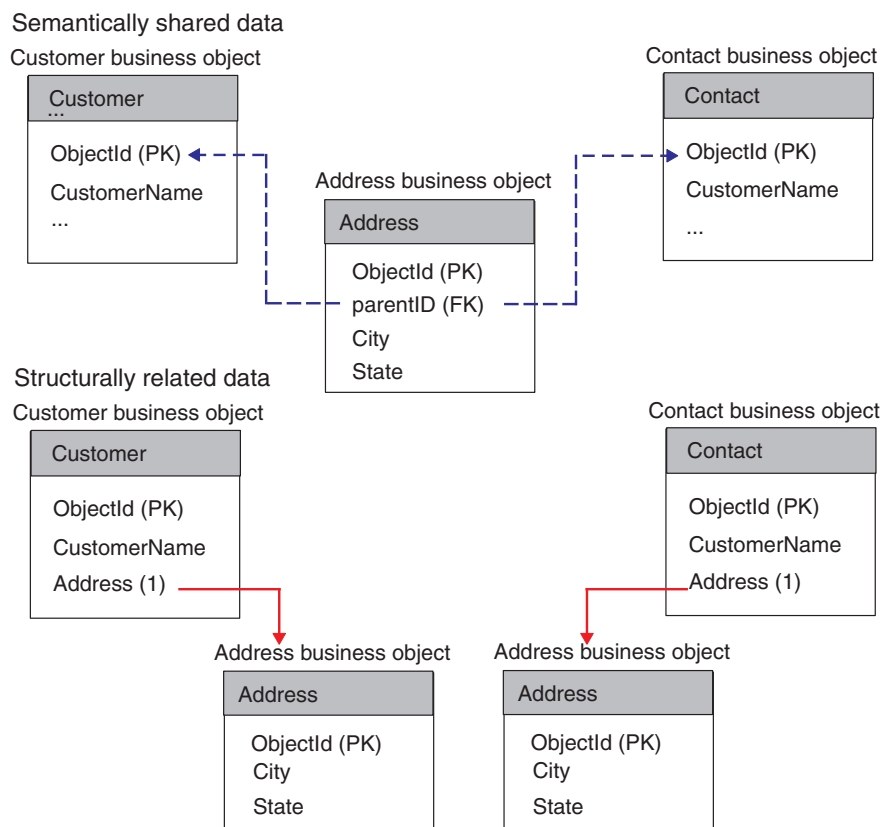
Figure 14. Comparing semantic and structural relationships

The figure above illustrates two kinds of relationships to child data:

- Semantic—The child Address business object, which is semantically linked to both Customer and Contact, contains the value of its parent's primary key in a simple foreign-key attribute. In this case, the name of the primary key attribute in both parents is the same, which simplifies the link from child to parent.
- Structural—The two business objects that structurally link to Address have an attribute that represents an instance of the child. In this case, the data in the child is related only to the parent that contains it and is not shared.

Choosing between a semantic and a structural relationship

As Table 3 on page 18 shows, both the one-to-one and one-to-many relationships can be represented by a structural or semantic relationship. Table 5 summarizes these structural and semantic representations.

Table 5. Representations for one-to-one and one-to-many relationships

Type of relationship	Structural representation	Semantic representation
One-to-one (single cardinality)	An attribute in a parent business object represents one child business object.	An attribute in a parent business object is simple and contains the foreign key to reference one child business object.
One-to-many (multiple-cardinality)	An attribute in a parent business object represents an array of child business objects.	Multiple child business objects each contain a foreign key attribute that stores the parent's primary key.

Figure 9 and Figure 10 illustrate business objects whose single- and multiple-cardinality relationships are defined semantically. The business objects in the example might represent data stored in a database. Relationships between business objects that represent such data can be defined both semantically and structurally. For such data, the relationship between a parent and child can be defined both semantically and structurally in the same two business objects.

Choosing a semantic relationship: To implement a semantic relationship, the underlying application should be able to support foreign keys. For example, when a business object represents database data, it can establish the relationship between entities both semantically and structurally. Such business objects are designed redundantly. In other words, the component that processes them can locate the child through the parent and the parent through each child.

For example, assume an application has a table that represents purchase orders. This table is related by foreign keys to a table that contains line items for a purchase order. Multiple rows in the line items table reference one row in the purchase orders table. Figure 15 illustrates these tables.
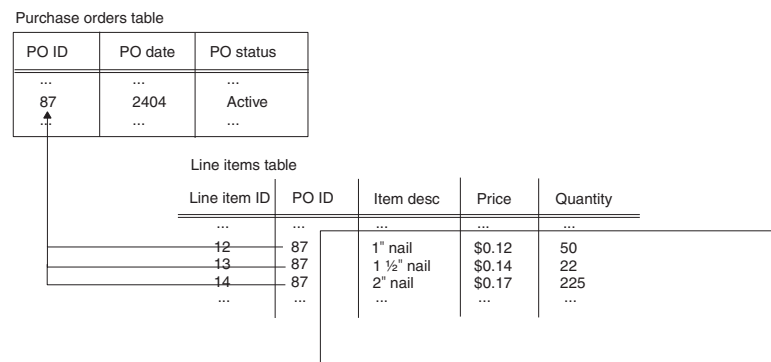
Figure 15. Example application tables with a one-to-many semantic relationship.

Figure 16 illustrates business objects that might correspond to these tables. This figure shows a top-level PurchaseOrder business object and three child LineItem business objects.
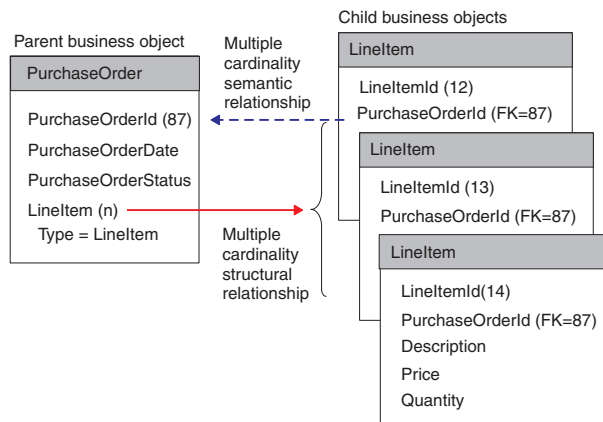
Figure 16. Sample business objects with multiple-cardinality semantic and structural relationships

The illustrated PurchaseOrder business object has both a semantic and structural relationship to its LineItem children. The PurchaseOrderId attribute in each child creates the foreign-key semantic link to the parent from the child. The LineItem attribute in the parent, which is defined with `cardinality n`, creates the structural link to the child from the parent.

Note: IBM does not deliver any business objects that have the foreign key stored in the child. This document presents the above example only to illustrate different ways to link parent and child data.

Choosing a structural relationship: If the underlying application does not support foreign keys, you probably need to implement a structural relationship. For example, a DTD, which represents one XML document does *not* support foreign-key information. Therefore, any one-to-one or one-to-many relationships must be defined structurally. The following Order DTD, which contains elements that correspond to an application `Order` entity, illustrates single- and multiple-cardinality relationships:

```
<!--Order -->
<!-- Element Declarations -->
<!ELEMENT Order (Unit+)>
<!ELEMENT Unit (PartNumber?, Quantity, Price, Accessory*)>
<!ELEMENT PartNumber (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Accessory (Quantity, Type)>
<!ATTLIST Accessory
Name CDATA >
<!ELEMENT Type (#PCDATA)>
```

Figure 17 illustrates a business object that represents the Order DTD. The top-level business object contains the Order business object with one-cardinality relationship, and Order contains the child Unit business objects with a multiple-cardinality

relationship. In turn, Unit contains the Accessory business objects with a multiple-cardinality relationship.
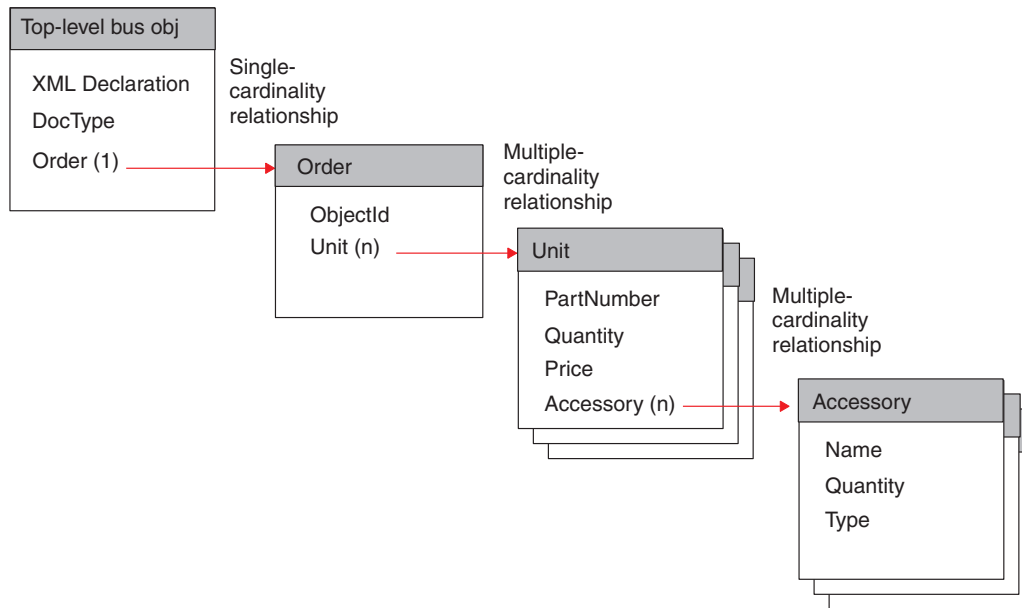
Figure 17. Single- and multiple-cardinality structural relationships

The relationship of business objects illustrated in Figure 17 is defined structurally; that is, each parent business object contains an attribute whose type is the same as the child's and whose relationship is specified as `containment`.

Important: The XML data handler has specific requirements of the top-level business object that represents a DTD. For information about these requirements, see the *Data Handler Guide*.

Designing application-specific business objects

An application-specific business object contains data, actions to be performed on the data (verbs), and information about the data (application-specific information). Many connector methods pass an application-specific business object as an argument. For example:

- When an application event occurs, some connectors invoke a data handler to convert the data's format into a business object, which the connector sends to InterChange Server Express.

- When InterChange Server Express sends a request to a connector, the connector framework sends the business object as an argument to the connector's business object handler. In this case, some connectors invoke the data handler to convert the information in the business object to the format used by the application, which enables the connector to perform operations in the application.

Designing the relationship among the connector, the data handler, and their supported application-specific business objects is one of the tasks in connector and data handler development. Because application-specific business object design can generate requirements for connector and data handler programming logic that must be integrated into the connector development process, the developers of the connector, data handler, and application-specific business objects must work

together to develop specifications for those components. The layout and design of an application-specific business object should be determined by the connector or data handler that processes it.

This section covers the following topics:
- "Contents of application-specific business object definitions"
- "Designing for an existing connector or data handler" on page 36

Contents of application-specific business object definitions

A business object definition includes the following information:

Table 6. Contents of a business object definition.

Contents of business object definition	Description	For more information
Business object structure	The structure of an application-specific business object is typically designed to correspond closely to the application entity (data structure) at the level that the connector or data handler interacts with the application (such as at the table level, the API level, or at different levels within an API).	"Structure of application-specific business objects" on page 30
Attribute properties	Attributes contain individual pieces of data within an application entity. They also have properties that provide information such as the data's type, cardinality, and default value. Attribute properties also specify whether the attribute is required or key.	"Attributes in an application-specific business object" on page 31
Application-specific information	An application-specific business object definition often includes text strings that tell the connector how the business object is represented in the application or how to process it.	"Business object application-specific information" on page 32

Structure of application-specific business objects

The way a connector or data handler processes business objects is determined in part by the structure of the business objects that it supports. As you design the structure of an application-specific business object, you need to determine what structure best represents a particular application entity and how this structure affects the design of connector and data handler logic or how the structure is processed by an existing connector or data handler.

While a goal of connector and data handler design is to code a connector or data handler so that it can handle new and changed business objects without modification, it is difficult to create a connector or data handler that can handle any possible business object.

Typically, a connector or data handler is designed to make assumptions about the structure of its business objects, the relationships between parent and child business objects, and the possible application representation of business objects. If designing for an existing connector or data handler, your task is to understand these assumptions and design business objects accordingly.

A beginning set of questions to consider about the structure of an application-specific business object is:

- What is the organization or database schema for the application entity that will be encapsulated in the business object. Does the application entity represent hierarchical data or one-to-many relationships?

- Does a business object represent one application entity or more than one application entity? In other words, can attribute values in an individual business object be stored in different application entities?

- What kind of relationships between business objects does the connector or data handler handle? How are the relationships modelled in the business objects or how are they processed by the connector or data handler?

For more information about the business object structures that can represent single or multiple application entities, see "Determining business object structure" on page 17.

Note: Certain connectors may require the top-level business object to contain specific information. For example, the XML connector requires its top-level business object to contain simple attributes for a URL, MIME type, and business object prefix, as well as complex attributes to contain a request business object and a response business object. If you are designing a business object for an existing connector, refer to its adapter user guide for specific structure requirements. For more information, see "Designing for an existing connector or data handler" on page 36.

Attributes in an application-specific business object

The attributes hold the individual pieces of data in an application entity. When defining attributes for an application-specific business object, consider these questions:

- What piece of data in the application entity will each attribute represent? For a business object representing a database entity, will each attribute represent a field, such as a table column? For a business object representing an XML document, will each attribute represent an element?

- Is it necessary to create an attribute for every single field in the application entity? Some pieces of data in one application entity may not be significant to the other applications in the integration; by leaving them out of the business object definition you can reduce the complexity of the design and prevent the transfer of unnecessary data from decreasing performance.

- Will the application-specific business object have fewer simple attributes than the application entity? For example, do you need an attribute for every database table column?

- How will the connector operate when the individual business object has more simple attributes than the corresponding database table has columns or the corresponding DTD has tags? In other words, some attributes in the business object are not represented in the database or the DTD. In most cases, these attributes convey information about specific access mechanisms or are used to separate attributes that represent child business objects. The connector or map may employ special logic that requires connector-specific attributes to handle certain application-specific business objects.

As a rule, keep the structure of the business object the same as the structure of the corresponding application entity (such as database tables or DTDs). If the business object is large (contains many attributes), define only the attributes that are used in the business process for which you are designing the business object. However, if

the business object is small, define all of the attributes to be available for future use. The number of attributes you define depends on the size of the business objects and the complexity of the relationships between them.

In addition to identifying which application entities must exist as attributes in the business object, you should also examine the business process to determine if any additional attributes are required. As part of the analysis of the business process, identify the business object's requirements. Stepping through a business process reveals how a business object is handled and how the required attributes are used. Variations in the business process and the handling of exceptions might identify additional attributes that are required to process the business object. These additional attributes might not correspond to data that is retrieved or updated in the application.

For example, you may need attributes that:
- serve as a priority indicator whose value is derived during processing according to the value of an attribute value
- serve as lookup that contains routing information based on one or more attribute values

Business object application-specific information

After you have defined the structure of an application-specific business object definition and defined the set of attributes that the business object definition contains, you can determine whether the connector or data handler needs additional information about how to process the business object to enable it to handle the requests it receives from InterChange Server Express. The business object definition can include this additional information in application-specific information.

Application-specific information provides the connector or data handler with application-dependent instructions on how to process business objects. The recommended approach to designing the relationship between business objects and connectors is to store information in the business object definition that helps a connector interact with an application or data source. Such information, called **metadata**, can be specified in the application-specific information of each business object, business object attribute, and business object verb.

The application-specific information is a string that is entered during business object design and read at run time by a connector or data handler. The connector or data handler uses the metadata in the business object definition to process business object instances. Because the connector or data handler has access to its supported business object definitions at run time, it can dynamically determine how to process a particular business object.

Consider the following advantages and limitations of application-specific information when designing a business object:
- Application-specific information enables a business object to be self-contained with all the information required to process it.

  The application-specific information in the business object definition can include table and column names, processing instructions, names of functions that the connector calls, or other information about how to process the data in the application.

  Because an application-specific business object contains all the information needed to process it, the connector can handle new or modified business objects without requiring modifications to the connector source code. The connector can

be written in a generic manner, with one business object handler that does not contain hard-coded logic for processing specific business objects.

- A metadata-driven connector can build application function calls or SQL statements from the values in a business object instance and the application-specific information in the business object definition.

  The function calls or SQL statements perform the required changes in the application database for the business object and verb the connector is processing.

- The application that a business object represents determines how much application-specific information the business object definition can contain.

  Depending on the application and its programming interface, a connector and its business objects might be designed so that the connector is almost entirely driven by the application-specific information in its business objects. In this case, the connector may require only one business object handler to transform business objects into requests for application operations.

  For some applications, however, the application interface may have constraints that force entirely different processing logic for different business objects and, therefore, the implementation of multiple business object handlers. For these applications, only a partially metadata-driven implementation or no data-driven implementation is possible.

  Depending on the application, business objects vary in how much application-specific information they contain. Most application-specific business objects, however, can be designed to contain some information that assists the connector or data handler with business object processing.

Suggested format of application-specific information: It is recommended that you use name-value pair syntax when you define application-specific information. This syntax specifies the name of the property and its associated value, separated by a equals sign (=), as the following syntax shows:

name1=value1;*name2=value2*

For example, the following name-value pair defines a "table name" property:

TN=*TableName*

Name-value pairs allow values to be specified in random order. The connector evaluates the name of each parameter before interpreting the value. It is recommended that you separate name-value pairs with a delimiter that:

- defaults to a semicolon (**;**)
- is configurable

Note: If you are creating a business object for an existing connector, check its adapter user guide to determine the syntax that it requires. Not all connectors may default to use a semicolon as a delimiter, or be configurable in that respect.

Table 7 provides examples of parameters that can be included in an attribute's application-specific information. These parameters are relevant only to a business object that represents data in a database table.

Table 7. Example name-value parameters for attribute application-specific information.

Parameter	Description
TN=*TableName*	The name of the database table.
CN=*col_name*	The name of the database column for this attribute.

Table 7. Example name-value parameters for attribute application-specific information. (continued)

Parameter	Description
FK=[..]*fk_attributeName*	The value of the Foreign Key property defines a parent/child relationship.
UID=AUTO	This parameter notifies the connector to generate the unique ID for the business object and load the value in this attribute.
CA=*set_attr_name*	The Copy Attribute property instructs the connector to copy the value of one attribute into another. If set_attr_name is set to the name of another attribute within the current individual business object, the connector uses the value of the specified attribute to set the value of this attribute before it adds the business object to the database during a Create operation.
OB=[ASC|DESC]	If a value is specified for the Order By parameter and the attribute is in a child business object, the connector uses the value of the attribute in the ORDER BY clause of retrieval queries to determine whether to retrieve the child business object in ascending order or descending order.
UNVL=*value*	Specifies the value the connector uses to represent a null when it retrieves a business object with null-valued attributes.

One attribute's application-specific information might combine several of the example parameters listed above. This example uses semicolon (;) delimiters to separate the parameters:

```
TN=LineItems;CN=POid;FK=..PO_ID
```

The application-specific information in this example specifies the name of the table, the name of the column, and that the current attribute is a foreign key that links the child business object to its parent.

Content of application-specific information: The content of application-specific information can vary considerably in complexity. Some examples are:

- The application-specific information in a business object definition can encode the name of the table that the business object corresponds to, and, for each attribute, it can encode the name of the column that the attribute corresponds to. This is a relatively simple implementation of application-specific information, but it may be all that a connector needs.

- A more complex implementation of application-specific information might contain a set of parameters that specify how the connector handles various business object operations.

- At its most complex, application-specific information might include conditions, direct connector transaction processing, specify methods of data retrieval, and provide preprocessing capabilities.

If the business object definition includes application-specific information and the connector has been designed to make use of it, the connector can extract the content of the application-specific information from the business object definition and use it for processing.

Example: How a connector processes application-specific information: As an example of how a connector processes application-specific information, assume

that your application is based on tables and that you want to work with an application table called CURRENTCUST, which stores information on customers. This table has two columns: CSTName and CSTCity.

In the AppSpecificInfo property of the business object header, you can store the table name. In the AppSpecificInfo property of each attribute, you can store the column names. In addition, because the connector for this application uses SQL statements to interact with the database, you can design the verb application-specific information to hold SQL verbs and keywords. Figure 18 illustrates how this Customer business object definition might look.
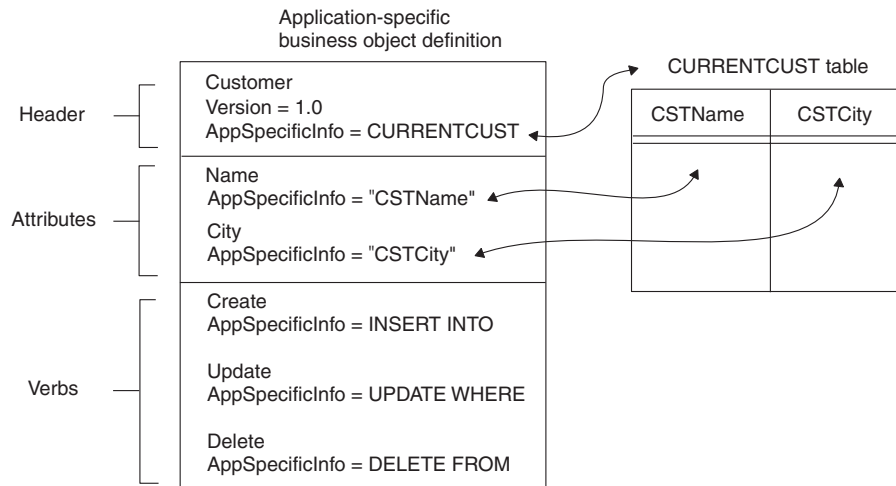
Figure 18. Application-specific information in a business object definition.

When a metadata-driven connector receives an instance of this business object from InterChange Server Express, it extracts the table name and column names from the application-specific information properties in the business object definition, and then extracts the attribute and verb values from the business object instance. Using the table and column names, the attribute values, and the SQL keywords in the verb application-specific information, the connector can build an SQL statement.

Figure 19 shows an example of this type of processing. The connector extracts the verb processing instructions and the table and column names from the business object definition. It then gets the attribute values from the business object instance. Using this information, the connector builds an SQL INSERT statement to update the CURRENTCUST table with the new information.
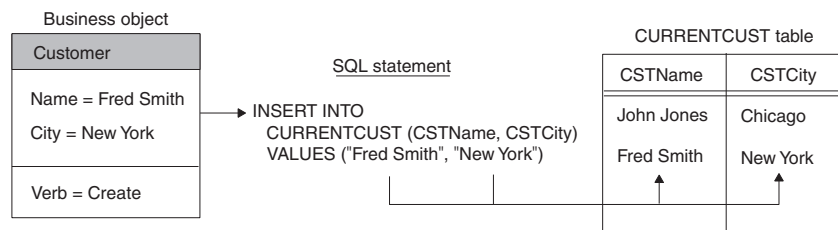
Figure 19. Using application-specific information to build an SQL statement for a Create operation

Chapter 2. Business object design   **35**

As mentioned above, a business object definition can include AppSpecificInfo text for the business object as a whole, and for its attributes and verbs. The following sections provide more information on the use of application-specific information in these components of a business object.

Important

The length of application-specific information is restricted to 1000 characters.

Figure 5 uses the attribute-level AppSpecificInfo property to store the name of the Invoice subform and the attribute's corresponding field name. The example uses name-value pairs to specify the information.

Tips on designing application-specific information: When designing business objects to maximize the metadata-driven behavior of a connector, follow these general recommendations for storing application-specific information in the business object definition:

- Store entity names, such as table or form names, in the business object-level AppSpecificInfo property of a top-level business object. Store subform names or table names in the business object-level AppSpecificInfo property of a child business object.
- Store field names, column names, and other information related to business object attributes in the attribute AppSpecificInfo property.
- Store verb processing instructions in the verb `AppSpecificInfo` property.

The careful use of the `AppSpecificInfo` property enables a connector to handle a variety of business objects in the same way. If an application is consistent in how it handles data operations, and if for all operations the connector performs consistent tasks, the business object can be designed to enable a completely metadata-driven connector.

Designing for an existing connector or data handler

If you are designing an application-specific business object for an existing connector or data handler, your first step is to consult its adapter user guide for its requirements on specifying application-specific information and using business object handlers. Keep the following points in mind when designing business objects for an existing connector or data handler:

- To determine if there is an available Object Discovery Agent, check the adapter guide for the connector and the documentation for the data handler that will process your business object. Using the Object Discovery Agent can greatly facilitate the business object design effort, particularly when the entity involved is large.
- Determine whether there is an existing business object available that models the application entity, such as a sample. Determine whether the effort to customize the existing business object is less than creating an entirely new one and if so then consider using the sample business object.

Important

IBM does not support sample business objects, but they can be very useful as a starting point for business object design.

- If there are no existing business objects for the entities you need to model and an Object Discovery Agent does not already exist for the application, you can develop a new Object Discovery Agent for the application. This might not be an efficient approach if there are very few business objects that are required for the application or if the entities are very small. For more information, see Chapter 5, "Developing an Object Discovery Agent," on page 87.
- Whether you use an Object Discovery Agent or an existing business object, it is still important to examine and confirm all the data definition requirements, such as the object key, foreign keys, child business objects, default values, data types, and size limitations. The following factors result in this requirement:
  - Object Discovery Agents can facilitate the design effort, but cannot discovery all of the requirements surrounding an application entity.
  - With existing business objects, the threat is that applications can be installed and configured different ways to accommodate customer-specific needs. A business object that accurately models an entity in one application installation may not accurately model that entity in another installation of that application.

While designing the application-specific business object, keep in mind that its primary role is to model the entity in the data source. It is also important to identify how its associated connector or data handler handles its processing, and what are the requirements of the business process in which it participates.

Designing generic business objects (InterChange Server Express only)

A *generic business object* reflects a superset of information that represents entities used by multiple diverse applications or programmatic entities. InterChange Server Express collaboration objects use generic business objects so they can provide information for a variety of diverse applications. Therefore, designing generic business objects is part of the task of collaboration object development. When designing a generic business object, take into account the following:

- Understand the data requirements that are critical to the business process integration.
- Study the business logic that the business object participates in, and all requirements based on that logic.

The following two considerations illustrate the complexity of business logic analysis:

- Processing prerequisite data

  Sometimes the application that triggers the collaboration object does not provide all the data required to process the triggering business object. The additional data may reside in other applications, including the destination application.

  For example, a Sales Force Automation (SFA) application (such as Siebel) may generate a quote that needs to be logged as an Order in an order management system (such as SAP). However, before the Quote can become an Order, it may require additional information not available in the SFA application. For example, an Order may require such additional data as customer credit status (from a financial system), contact information (from a customer support system), or Availability To Promise information (from a warehousing system).

When you design the generic Order business object, you may have to include attributes and design a structure that supports data which is not present in the source application necessarily, but may be present in other applications involved in the interface.

– Cross-referencing between individual application entities

Determine how and whether individual application entities correspond to each other or are cross-referenced to each other generically in the business process.

For example, a customer in Oracle is represented as a Customer whose address is represented as an Address. A customer in SAP is represented as a "SoldTo" entity whose address is represented as a "ship-to" entity. A customer in Clarify is represented as a "Business Organization" whose address is represented as a "Site".

Study the functionality and relationships between an application's entities to determine the business objects and processes involved in integrating data between applications.

- Understand what required data is common or shared across all applications participating in the business process and is critical to the integration of that process. The attributes and their relationships should determine at a minimum the mandatory attributes (the lowest common denominator of attributes) and the transformations that the business integration system must perform between these application-specific business objects.

Consider the following integration possibilities:

– The business process integrates data from an Enterprise Resource Management (ERP) system to a Customer Relationship Management (CRM) system. In this case, the business object probably does not need to be very complex because most CRM systems do not accommodate most of the data stored in an ERP system.

– The business process integrates data between two ERP systems. In this case, it is likely that the business object will be highly complex.

– The business process integrates data from a CRM system to an ERP system. In this case, your design must reflect how much of the data actually originates in the CRM system (and thus must be represented by attributes in the generic business object) and how much of it can be defaulted in the destination application itself (and thus provided as default values in the application-specific business object).

- If they exist, study the application-specific business objects to which the generic business object will map. Analyze the structure and the attributes of all business objects to derive a generic business object that is suitable to all applications.

- Consider whether there is a standard for the type of business object you are designing. For example, there might be an appropriate model for the entity provided by the Electronic Data Interchange (EDI), Open Applications Group (OAG), or Object Management Group (OMG) initiatives.

Generic business object design standards

To be consistent with IBM-delivered generic business objects, use the following standards when designing a generic business object:

- The first attribute of every object should be its key and should be named `ObjectId`.

- If an attribute represents a foreign key, its name should concatenate the name of the foreign business object and Id; for example: `CustomerId`, `ItemId`, and `OrderId`.

- Be consistent. If you use an abbreviation in an attribute name, use the same abbreviation in parent and child business objects. If possible, use the same abbreviation for all relevant attribute names. For example, if you abbreviate `Number` to `Num`, do so consistently.
- Follow the naming guidelines provided in *Naming IBM WebSphere InterChange Server Express Components*.

Designing for event isolation

When designing a generic business object, it is recommended that you consider the needs of event isolation, as explained in the *Collaboration Development Guide* (in the section entitled "Designing for Parallel Execution").

To prevent more than one collaboration object from updating the same data at the same time, each business object should be modified by only one type of collaboration object. In other words, a Customer business object should be modified only by a CustomerSync collaboration object.

If a collaboration object modifies a business object that contains a child business object, and the child business object is also contained by a different top-level business object that has its own modifying collaboration object, design the top-level business objects to contain the child semantically rather than structurally. Develop a third collaboration object to modify the shared child. The collaboration objects that own the two top-level business objects should then delegate processing of the shared child to the third collaboration object.

For example, if both Customer and Contact business objects contain the same address data, design the Address business object as a top-level business object that is referenced by Customer and Contact, but not contained by them. Then develop a separate Address collaboration object to modify address data.

In another example, however, if the Order business object is the only business object that modifies OrderLineItem data, you can design Order to contain the OrderLineItem child business objects rather than merely reference them.

In other words, design the Customer and Contact business objects so that they contain a foreign-key attribute that references the Address business object, that is, that contains only the key value for Address. Do not design them to contain an attribute that represents a full-valued Address business object. But design the Order business object to contain an attribute that represents a full-valued OrderLineItem business object.

Note: Designing shared business objects as referenced rather than contained can simplify business object distribution. If the same child business object is defined in multiple business object definitions, the `repos_copy` utility attempts to load the same business object twice during installation, causing rollback. For information on `repos_copy` flags that change this default behavior, see the *System Administration Guide*.

Attributes in a generic business object

When defining attributes for a generic business object, study the attributes of the application-specific business objects to which the generic business object will map. Consider these guidelines:

- Note the similarities between entities in the application-specific business objects's attributes. Define attributes for the generic business object that most simply match those in the application-specific business objects.
- Note the differences between entities in the application-specific business objects' attributes. If one application-specific business object splits data into multiple fields while another combines the same data into one field, determine which design best simplifies mapping between the two application entities. For more information, see "Designing for an existing connector or data handler" on page 36.
- Consider requirements generated by the processing performed by the collaboration object. For example, if the collaboration object processes prerequisites as described in "Designing generic business objects (InterChange Server Express only)" on page 37, ensure that the generic business object contains all attributes required to store the prerequisite data.
- Develop the generic business object and interface to accommodate the largest number of applications involved in the interface. For instance, if there are four applications involved in an interface and three of them encapsulate data in a child object but the fourth contains that data at the parent-level object, then design the generic business object so that it encapsulates the data in a child object as well—this results in mapping and other related tasks being that much easier.
- Take future development efforts into account; you may want to design a generic business object to accommodate data structures that will be required at a later point to minimize the effort and change impact at that time. Do not, however, significantly increase the scope of development for a future project that may never come to be.

In general, a generic business object definition should include attributes that capture all the data elements that are to be transformed among all the application-specific business objects to which the generic business object will map.

Names of the attributes should be as intuitive as possible. For example, if several applications refer to an entity as a `Customer` and one application refers to the same entity as a `Business Organization`, use the more common terminology to name the generic attributes.

Note: The name of an attribute can contain only alphanumeric characters and underscore (_).

Evaluating existing generic business objects

You may be able to facilitate development of a generic business object by copying and customizing an existing one.

To evaluate a generic business object, examine the data involved in the interface. A guideline is that if 80% or more of the data exists in a delivered generic business object, customize the existing object.

When performing this analysis, it is more important to look at the business object structure than the attributes. Attributes are relatively easy to add and remove, whereas structural or hierarchical changes can require much more effort.

If you decide to customize an existing generic business object, examine the business object definition to determine whether it is missing one or more desired attributes. Missing attributes become more apparent during mapping design. If the

generic business object requires one or more additional attributes, create a child business object that contains the additional attributes. Isolating custom attributes in child business objects facilitates future upgrade of IBM-delivered business objects.

If you embed custom attributes in an IBM-delivered business object, upgrading to a new version of the business object requires re-embedding those attributes in the new business object. Isolating the custom attributes in their own business object allows you to add one attribute to the new IBM business object—the attribute that creates the relationship between the parent and the custom child business object. If you are customizing a hierarchical business object that requires additional attributes in both the parent and the child, create separate child business objects for each.

It is recommended that you name custom attributes and business objects in a way that readily identifies them. A simple convention is to add an _x suffix to each custom name. For example, if you create a custom child business object that adds attributes to the generic Order business object, name the child `Order_x`. Doing so allows alphabetic listing to keep related names together. If it is more important to identify custom business objects or attributes than to alphabetize the custom object with its generic object, add an x_ prefix to each custom name. For more information, see *Naming IBM WebSphere InterChange Server Express Components*.

Determining mapping requirements for business objects (InterChange Server Express only)

When an application-specific business object has been designed to match an application entity, it may not match its corresponding generic business object. Therefore, you must create maps between the application-specific business object and the generic business object so that the application data can be transported across the WebSphere business integration system.

An application-specific business object may not need to include all the fields or columns or elements in an application entity. Use the functional requirements of the application and the business processes in which it participates to identify which attributes belong in the application-specific business object.

You can also examine the correspondence between the generic business object and the application entity. You may choose to include fields in the application-specific business object that correspond those in the generic, which allows these data elements to participate in the business process.

When designing the business object, note the differences between the application entity and the generic business object. These differences define what kind of data transformation needs to take place. You may need to design mapping to:
- Combine multiple fields in the application entity to fill one attribute in the generic business object
- Split a field in the application entity to fill multiple attributes in the generic business object
- Ignore a field that is present in the generic business object but that is not relevant to the application entity
- Handle differences in semantic or structural relationships between an application-specific business object and a generic business object
- Handle foreign key relationships and other types of relationships between application entities

- Establish associations between data, for example:
  - Establish a lookup association between data in non-key attributes, such as an association that transforms code values (for example, marital status or currency code) between applications.
  - Establish an identity association between data in business objects, such as an association that transforms the key attributes (for example, unique identifiers and product codes) between applications.

To assist with mapping and design concepts, the relationship among fields in a table, attributes in an application-specific business object, and attributes in a generic business object is shown in a highly simplified way in Figure 20. The differences between the application-specific business object and the generic business object are handled in mapping. If the business object has attributes that do not have a representation in the database, the connector can provide a default value for the attribute.
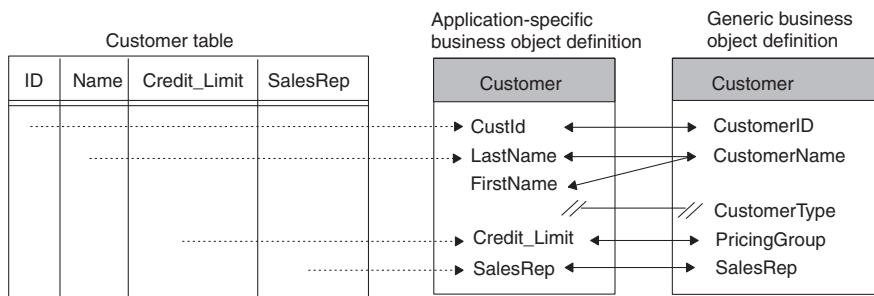
Figure 20. High-level view of field/attribute relationships.

For information on creating maps, see the *Map Development Guide*.

Chapter 3. Using Business Object Designer Express

The Business Object Designer Express tool is used to create, edit, and delete business object definitions. This chapter provides an overview of how to start and use Business Object Designer Express. The main topics of this chapter are:

- "Working with projects"
- "Starting Business Object Designer Express" on page 46
- "Opening a business object definition from Business Object Designer Express" on page 47
- "Working with business object definitions" on page 50
- "Business Object Designer Express functionality" on page 51

Working with projects

Business Object Designer Express uses the concept of a *project* to define a virtual work area in which business object definitions are created, modified, or deleted. Depending on your environment, the "project" to which Business Object Designer Express refers in its dialog boxes can be either of the following:

Table 8. Projects in Business Object Designer Express

Business Object Designer Express environment	Project
You are not running Business Object Designer Express from System Manager.	A virtual work area into which you have imported business object definitions from a local directory to work with during your current Business Object Designer Express session. Also called a *local project*.
You are running Business Object Designer Express from System Manager.	An Integration Component Library (ICL) on the Windows machine where Business Object Designer Express and System Manager are running. Also called an *ICL-based project*.

The use of each type of project is explained in more detail below.

If Business Object Designer Express is running without System Manager

If you are not running Business Object Designer Express from System Manager, Business Object Designer Express uses a local project as "the project". A local project is a virtual work area into which you can import business object definitions you want to work with.

How Business Object Designer Express works with a local project

Listed below is a high-level summary of how Business Object Designer Express functions operate on a local project. More detailed information about performing these tasks is provided in the topics starting with "Starting Business Object Designer Express" on page 46.

- **Editing existing Business object definitions:** To edit an existing business object definition, click **File -> Open From File**. This menu item imports the business object definition from a local directory into your project and optionally opens it for editing.

  To edit an existing business object definition that has already been imported into your project but that is now closed, click **File -> Open**.

- **Creating a new business object definition:** To create a new business object definition, click **File -> New or File -> New Using ODA**.

- **Saving a business object definition:** To save a new or modified business object definition, click **Save** on the menu bar. You are prompted to save it to a local directory. To save a business object definition under a different name or directory, click **Save As**.

- **Deleting business object definitions:** To delete a business object definition from the Windows directory where it resides, use the tools provided by Windows. You cannot use the **Delete** function in Business Object Designer Express to do this. To delete a business object definition from a local project, select **File -> Delete**. You are prompted to select a business object definition to delete from your project.

If Business Object Designer Express is running from System Manager

When you run Business Object Designer Express from System Manager, you have access to additional, more sophisticated, functionality for developing and managing business object definitions. In System Manager, business object definitions, along with other business integration components such as collaborations and maps, are stored in *Integration Component Libraries* (ICLs). ICLs are repositories of business integration components, which you can use as building blocks to construct business integration solutions. Each ICL contains a collection of folders, one for each type of integration component, as shown in Figure 21.
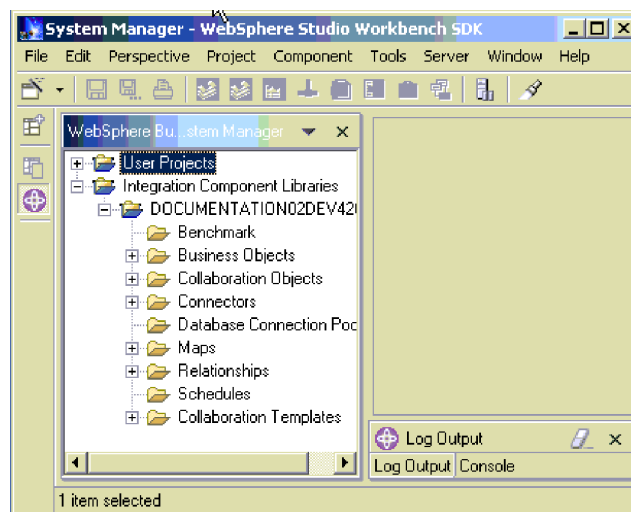
Figure 21. Integration Component Libraries in System Manager.

The methodology for developing and deploying business object definitions is as follows. You develop a business object definition in Business Object Designer Express and save it to the business objects folder in an ICL. When you want to use that business object definition in a business integration solution, you associate the definition with one or more *user projects*. Each user project includes all the business

integration components needed to implement a particular business integration solution. For example, in a user project that contains the components needed for the implementation of the PeopleSoft adapter, the business objects folder contains all the business object definitions needed by that adapter.

Like an ICL, each user project contains a collection of business integration component folders. However, a user project contains only virtual copies of ICL components. When you change a business object definition, you modify the instance in the ICL. The changes you make are automatically propagated to every user project that includes the business object definition. In other words, if a particular business object definition is included in two user projects, and a change is made to that definition in the Integration Component Library, the change is automatically reflected in the virtual copies residing in the user projects. This linkage between business object definitions in an ICL and their virtual copies in the user projects allows you to modify and maintain business object definitions in one central location while deploying them in multiple business integration solutions.
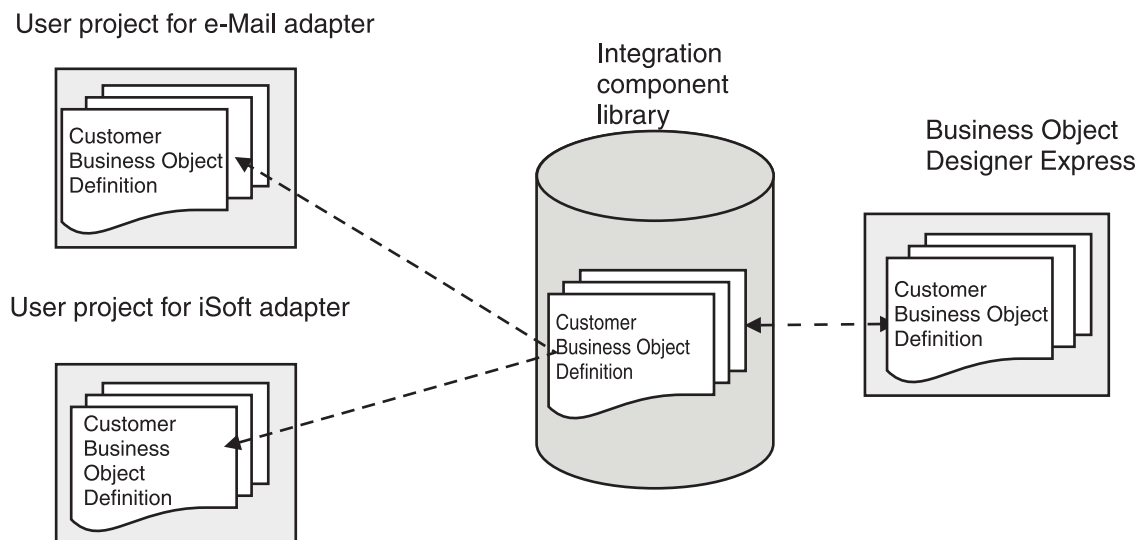
Figure 22. Changes to business object definitions in an ICL propagate automatically to virtual copies in user projects.

For more information about developing business integration components using Integration Component Libraries, see the implementation guide for your system.

How Business Object Designer Express works with an ICL-based project

When you are running Business Object Designer Express from System Manager, it uses as "the project" the Integration Component Library you have selected. Listed below is a high-level summary of how Business Object Designer Express functions operate on an ICL-based project. More detailed information about performing these tasks is provided in the topics starting with "Starting Business Object Designer Express" on page 46

- **Editing existing Business object definitions:** To edit a business object definition stored in a project, click **File -> Open**.
- **Creating new business object definitions:** To create a new business object definition, click **File -> New** or **File -> New Using ODA**.

- **Saving business object definitions:** To save a new or modified business object definition, click **File -> Save**. The business object is saved to the business objects folder in the project. To save a new or modified business object definition using a different name, click **File -> Save As**.

- **Deleting business object definitions:** To delete a business object definition, select **Delete** from the menu bar. You are prompted to select a business object definition to delete from your project.

When you run Business Object Designer Express without System Manager, you do not have access to the Integration Component Libraries. In this environment, Business Object Designer Express uses a local project as described in "If Business Object Designer Express is running without System Manager" on page 43.

Starting Business Object Designer Express

You can open Business Object Designer Express in any of the ways listed in Table 9. After opening Business Object Designer Express, you can create a business object definition manually or use an Object Discovery Agent to generate a definition for an application-specific business object. For more information, see Chapter 4, "Developing business object definitions," on page 55.

Table 9. Ways to open Business Object Designer Express.

From System Manager	• Select the business objects folder in an Integration Component Library, then do either of the following: – Click **Business Object Designer Express** from the **Tools** menu. – Click the **Business Object Designer Express** tool bar icon. • Right-click the business objects folder in an Integration Component Library. • Double-click a business object definition.
Using a Windows shortcut (InterChange Server Express)	Click **Programs -> IBM WebSphere InterChange Server Express -> IBM WebSphere Business Integration Toolset Express-> Development -> Business Object Designer Express**.
From another development tool (InterChange Server Express only)	Do either of the following: • On the **Tools** menu, click **Business Object Designer Express**. • From the tool bar, double-click the Business Object Designer Express icon.

When you open Business Object Designer Express directly from System Manager, without first selecting a business object definition, the New Business Object dialog box opens automatically. If System Manager is not running, Business Object Designer Express opens, but the New Business Object dialog box does not open.
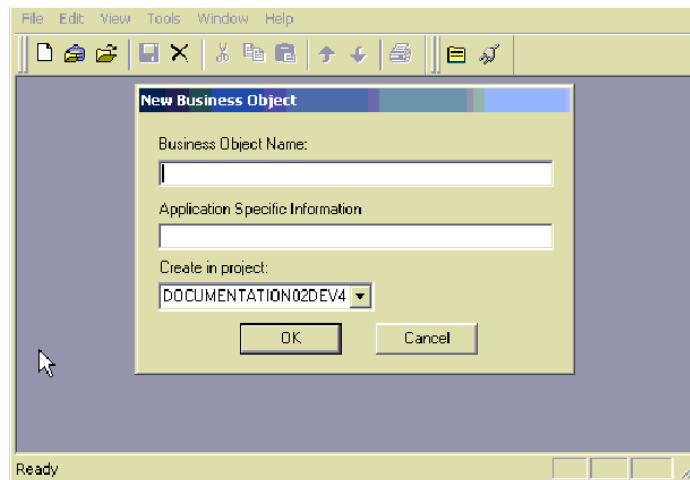
Figure 23. New Business Object dialog box

When you open Business Object Designer Express by double-clicking a business object definition, the selected definition is displayed in the Business Object Designer Express work area.

Opening a business object definition from Business Object Designer Express

Once you open Business Object Designer Express, you can open object definitions stored in a file. If Business Object Designer Express is running from System Manager, you can also open business object definitions stored in an Integration Component Library.

This section describes:
- "Opening a business object definition from a project"
- "Preventing duplicate definition names" on page 48
- "Opening a definition from a file" on page 48

Opening a business object definition from a project

If Business Object Designer Express is already open, you can do the following to open a business object definition from a project.

Note: If Business Object Designer Express is running from System Manager, the project is an ICL-based project. Otherwise, the project is a local project, which contains only business object definitions you have imported into it. See "Working with projects" on page 43 for more information about projects in Business Object Designer Express.

1. From the list of business object definitions in the project, highlight the name of the definition you want to open.
2. To select multiple business object definitions in the project, do one of the following:
   - When selecting consecutive names, select the first name and, while pressing the Shift key, click the last name.
   - When selecting non-consecutive names, press the Ctrl key and click as you select each name.

3. After selecting the definitions to be opened, right-click and then click **Open**.

   Business Object Designer Express displays a window for each selected definition.

Opening a definition from a file

To open a business object definition that is stored in a local directory, do the following:

1. Click **File -> Open From File**.

   The Import dialog box opens. The dialog box defaults to filter files of type XML Schema Definition (with a `.xsd` extension). You can also select a different file type from the **Files of type** list, or you can select all file types.

2. In the Import dialog box, browse until you locate the file, select it, and click **Open**. Figure 24 illustrates this dialog box.

   Note: If Business Object Designer Express is not running from System Manager, the **To Project** list, which lets you specify the ICL-based project to receive the imported business object definition, is omitted from the dialog box. Instead, the business object definition is imported into your local project.
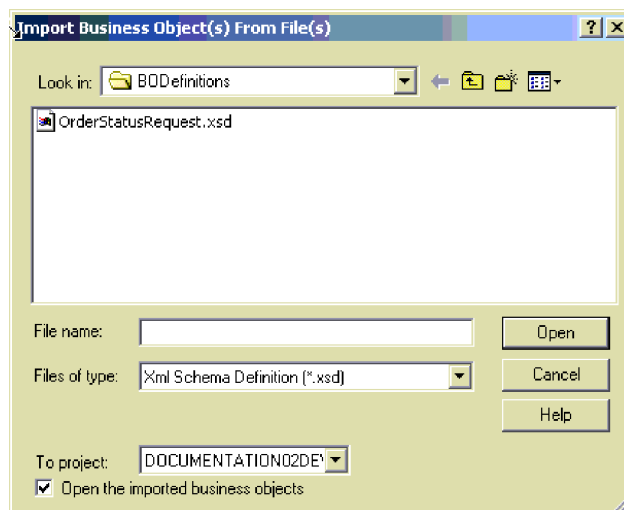
Figure 24. Import Business Objects dialog box

If the **Open the imported business objects check box** is selected then Business Object Designer Express also opens the business object definition for editing. Otherwise, the business object definition is imported into the project but not opened for editing. For more information, see "Working with business object definitions" on page 50.

Preventing duplicate definition names

Business Object Designer Express does not allow you to have two business objects with the same name in the same project, which might occur in either of the following situations:

- You attempt to open a definition from file that is identical to one you already have in your project.
- You attempt to create a new definition that is identical to one that already exists in your project.

In this case, Business Object Designer Express displays an error message with the text: Business object with this name already exists.

If you attempt to open a definition from a file and your local or ICL-based project already contains a definition with the same name, Business Object Designer Express displays the Import Results dialog box illustrated in Figure 25.
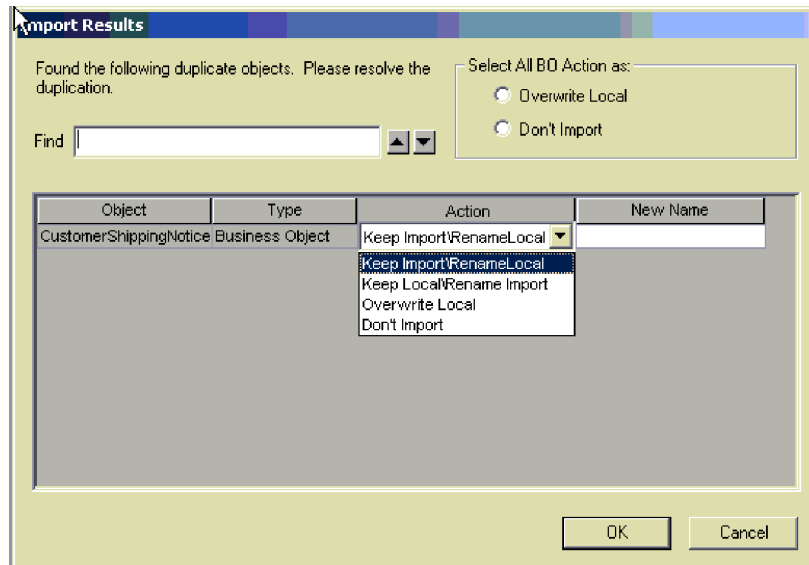
Figure 25. Preventing duplicate names: Keep Local or Import

In the Import Results dialog box, do the following for each business object definition listed as having a duplicate name:

1. Click **Action** and select an action from the list. An explanation of each action is provided below.
2. If you selected **Keep Import/Rename Local** or **Keep Local/Rename Import**, type the new name for the business object definition in the **Name** column as shown in Figure 25.

Alternatively, you can use **Select All BO Action as** to specify either of two actions for every business object definition listed as a duplicate name. To overwrite all the business object definitions in your project with the definitions you are importing, select **Overwrite Local**. To refrain from importing the business object definitions that have duplicate names, select **Don't Import**.

The **Actions** list in the Import Results dialog box provides the following options:

- **Keep Import\Rename Local**—Allows you to change the name of the definition in your project and leave unchanged the name of the definition in the file.

  To make this change, enter the new name in the **New Name** column as shown in Figure 25.

- **Keep Local\Rename Import**—Allows you to change the name of the definition in the file and leave unchanged the name of the definition in your project.

  To make this change, enter the new name in the **New Name** column as shown in Figure 25.

- **Overwrite Local**—Overwrites the definition currently stored your project with the definition stored in the file.

- **Don't Import**—Cancels the action to import the definition stored in the file.

Working with business object definitions

Business Object Designer Express provides a tabbed dialog box with two screens for creating and editing a definition:

- **General** tab — specify or change application-specific information and verbs at the business object-level.
- **Attributes** tab — specify or change attribute properties.

When you first create or open a definition, the **Attributes** tab opens.

Figure 26 illustrates the environment for defining and editing attributes.
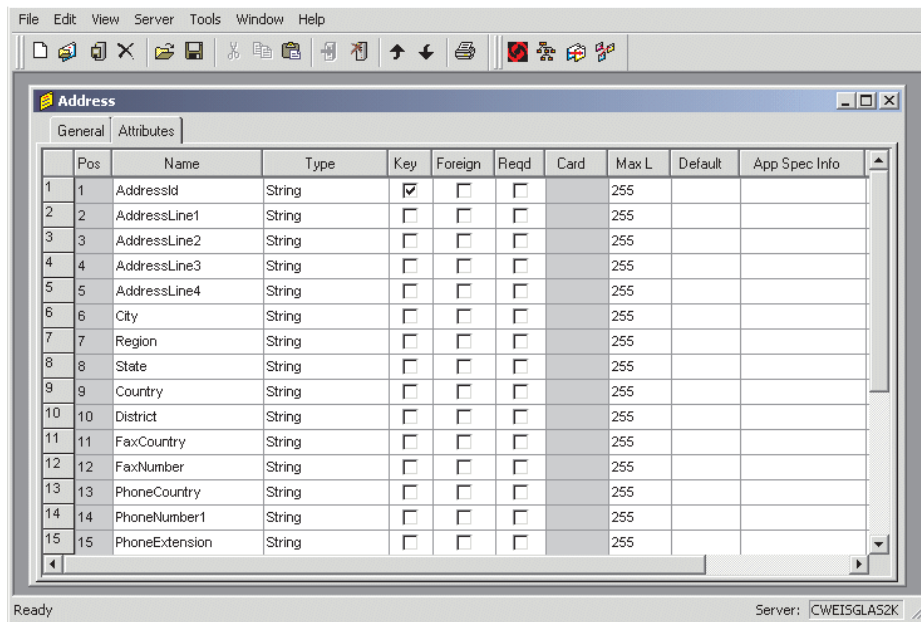
Figure 26. Defining and editing attributes

For information on using the **General** and **Attributes** tabs, see "Creating a business object definition" on page 55.

Opening a business object definition and its contained child

Business Object Designer Express allows you to open separate windows to edit definitions for a parent business object definition and its contained child.

Figure 27 illustrates the separate windows for editing parent and child business objects.

Address business object definition →

Contact business object definition →

Contained Address business object definition →
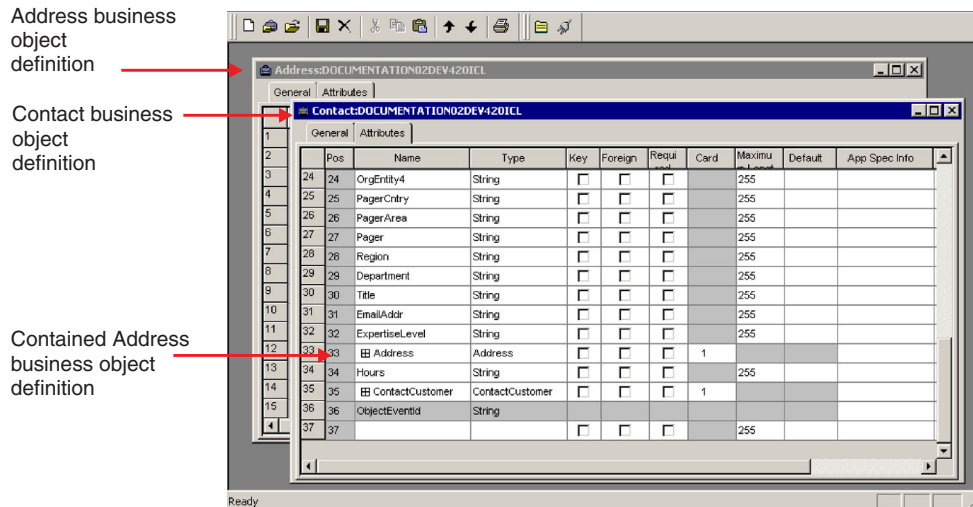
Figure 27. Separate windows for parent and child business objects

Notice that the Address attribute in the Contact business object is collapsed in Figure 27. You can expand the attribute so that the Contact window displays all attributes of the Address business object, which enables you to edit the child directly from the parent. To prevent you from changing the same definition in two places, however, the tool automatically closes a child business object's window whenever you expand a child business object within its parent business object.

Figure 28 illustrates the tool after Contact's Address attribute has been expanded and the Address window has been closed.

Expanded Address attribute →
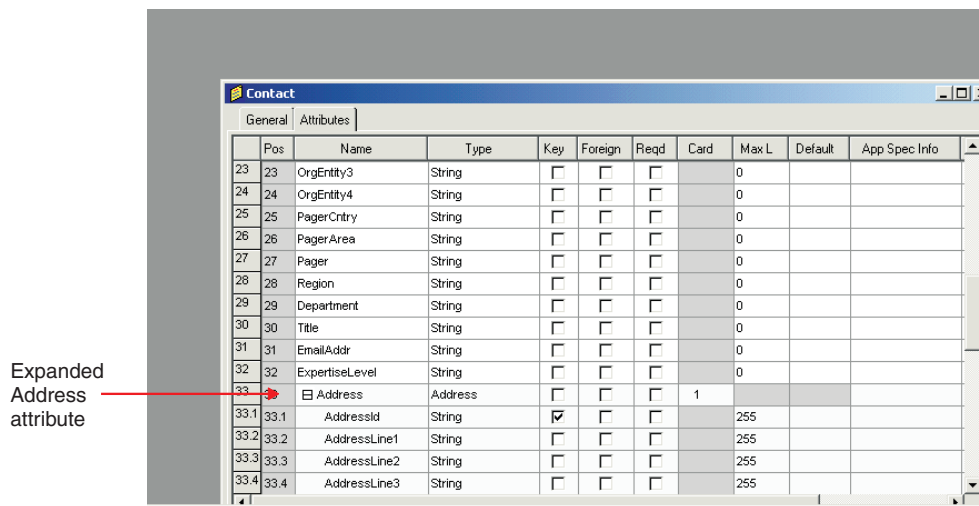
Figure 28. Expanding a parent business object's attribute that represents the child business object

Business Object Designer Express functionality

You can access Business Object Designer Express functions in either of the following ways:

- From the menu bar
- Using toolbar icons.

The following sections provide an overview Business Object Designer Express menus and menu options.

File menu

The **File** menu contains the following items:

- **New BO** — Creates a business object definition manually. For more information, see "Creating a business object definition" on page 55.
- **New Using ODA** — Presents the Business Object Wizard, which allows you to create a business object definition from an Object Discovery Agent. For more information, see "Using an Object Discovery Agent to create a business object definition" on page 64.
- **Open** — Opens a business object definition located in the project. If Business Object Designer Express is running from System Manager, the project is an ICL-based project. Otherwise, the project is a local project. See "Working with projects" on page 43 for more information about projects.
- **Open From File** — Imports and optionally opens a business object definition from a local directory.
- **Save** — Saves the business object definition as follows to the project.

    If you modified an existing business object definition:
    - If the project is ICL-based, the business object definition is saved in the project where it originated.
    - If the project is local, the business object definition is saved to its existing file.

    If you created a new business object definition:
    - If Business Object Designer Express is running from System manager, you are prompted to select the ICL in which you want to save the business object definition.
    - Otherwise, you are prompted to specify the local destination directory and file name for the business object definition.

    The business object definition can be saved as a file of type:

.xsd	XML Schema Definition. This is the default file type.
.in or .text	InterChange Server Express
.sly	Spreadsheet

- **Save As** — Saves the business object definition with a new name. The name of the file containing the business object definition must be unique within the destination project.

    If you modified an existing business object definition:
    - If Business Object Designer Express is running from System Manager, the business object definition is saved to the ICL-based project.
    - If the business object definition was opened from a file, the modified business object definition is saved to that file.

    If you created a new business object definition:
    - If Business Object Designer Express is running from System Manager, you are prompted to select the ICL where the business object definition is to be saved.
    - Otherwise, you are prompted to specify a local destination directory and file name for the business object definition.

    The business object definition can be saved as a file of type:

.xsd	XML Schema Definition. This is the default file type.
.in or.txt	InterChange Server Express

| .xls | Spreadsheet |

- **Save All**—Saves all open business object definitions as described under the **Save** menu item on 52.
- **Save Copy to File**—Exports a copy of the business object definition to a separate file.
- **Copy All to One File**—Exports all business object definitions in a project as one file in repos-copy format.
- **Close**—Closes the selected definition. This menu item is not available if no definitions are open.
- **Close All**—Closes all open definitions. This menu item is not available if no definitions are open.
- **Delete** — Allows you to delete a business object definition from a project.

  Note: Business Object Designer Express only lets you delete business object definitions from the project. If your project is ICL-based, the business object definitions you delete are removed from the specified ICL. If your project is local, the business object definitions you delete are removed from the local project but the files that contain business object definitions in local directories are not affected. To delete local files, use the tools provided by Windows.

- **Print Setup** — Allows you to specify the printer and printing properties.
- **Print Preview** — Displays a preview of the definition to be printed. This menu item is not available if no definitions are open.
- **Print** — Allows you to print the selected definition. This menu item is not available if no definitions are open.
- **Exit** — Allows you to exit Business Object Designer Express.

Edit menu

All options of the **Edit** menu are not available if no definitions are open. The **Edit** menu contains the following items:

- **Cut** — Deletes an attribute from the definition or text from a column. This menu item is not available if no text has been selected in a column or no attribute has been selected (by clicking in the left-most column). See Figure 26 on page 50 for an illustration of the window for editing attributes.
- **Copy** — Copies an attribute in the definition or text in a column. This menu item is not available if no text has been selected in a column or no attribute has been selected (by clicking in the left-most column). See Figure 26 on page 50 for an illustration of the window for editing attributes.
- **Paste** — Pastes a cut or copied attribute into the definition, or cut or copied text into the selected column. By default, the tool pastes a buffered attribute at the bottom of the definition. However, if you insert an empty row at a specific location, you can paste the buffered attribute into the empty row.
- **Delete Row** — Deletes an attribute from the definition. This menu item is not available if no attribute has been selected (by clicking in the left-most column). See Figure 26 on page 50 for an illustration of the screen for editing attributes.
- **Select All** — Selects all attributes in the definition.
- **Insert Above** — Inserts an empty row above the selected attribute.
- **Insert Below** — Inserts an empty row below the selected attribute.
- **Move Up** — Moves the selected attribute up one row. This menu item is not available if no attribute has been selected.

- **Move Down** — Moves the selected attribute down one row. This menu item is not available if no attribute has been selected.

Note: You can access the **Insert Above, Insert Below, Cut, Copy, Paste**, and **Delete** menu items by right-clicking in the left-most column of an attribute.

View menu

The **View** menu operations are valid when Business Object Designer Express first opens and when the working area pertains to the visual appearance of activity diagrams. Many of these operations can be toggled on or off.

The **View** menu displays the following options:
- **Expand All** — Displays all attributes in all child business objects. This menu item is not available if no definitions are open.
- **Collapse All** — Closes display of all attributes in all child business objects. This menu item is not available if no definitions are open.
- **Preferences** — Opens the Business Object Preferences dialog box, which allows you to turn off confirmation of business object deletion.
- **Toolbars** — Contains a submenu with items that control display of the two toolbars of the Business Object Designer Express. Menu options include:
  - **Standard** — When you click this menu item, Business Object Designer Express displays the buttons for the Standard toolbar.
  - **Programs** — When you click this menu item, Business Object Designer Express displays the buttons for accessing other WebSphere Business Integration Toolset Express programs.
- **Status Bar** — When you click this menu item, Business Object Designer Express displays a one-line status message at the bottom of its main window.

Note: You can access the **Expand**, **Collapse**, and **Open In Window** items when right-clicking in the left-most column of an attribute that represents a child business object or an array of child business objects.

Tools menu

The **Tools** menu contains the following items:
- **Log Viewer** — Opens Log Viewer.
- **Connector Configurator Express** — Opens Connector Configurator Express.
- **System Manager** — Opens System Manager.

Window menu

The **Window** menu operates as it does in a standard Windows environment. Use the menu options to control display features such as tiling, cascading, and activating open windows.

Chapter 4. Developing business object definitions

This chapter walks you through the basic steps for creating and deleting a business object definition. After you complete this chapter, you will be familiar with the steps for creating a definition both manually and by using an Object Discovery Agent (ODA). Each ODA generates definitions for a specific application.

Although this chapter presents the mechanics for creating business object definitions, you should understand the design concepts before you actually create one. For more information, see Chapter 2, "Business object design," on page 17. For information on how to create an Object Discovery Agent, see Chapter 5, "Developing an Object Discovery Agent," on page 87.

The main topics of this chapter are:
- "Creating a business object definition"
- "Deleting a business object definition" on page 61
- "Using an Object Discovery Agent to create a business object definition" on page 64

Creating a business object definition

There are two ways to create a business object definition:
- Manually—Useful when creating a generic business object or a simple business object, or when modifying a definition generated by an Object Discovery Agent. Business Object Designer Express provides a graphic interface for the manual creation of a business object definition. This section provides a tutorial that explains:
    - "Creating a flat business object definition manually"
    - "Creating a hierarchical business object definition manually" on page 61
- Using an Object Discovery Agent—Useful when creating an application-specific business object. The *Object Discovery Agent* examines specified entities in the application, "discovers" the elements of those objects that correspond to business object attributes and the properties of each attribute, and generates the business object definition. For more information, see "Using an Object Discovery Agent to create a business object definition" on page 64.

Creating a flat business object definition manually

This section describes the manual creation of a business object definition named Hello. With InterChange Server Express, this business object is used by the SampleHello collaboration, whose creation is described in the tutorial chapter of the *Collaboration Development Guide*.

Figure 29 illustrates the Hello business object definition that you can create and shows the values that InterChange Server Express might expect from its triggering-event business object.
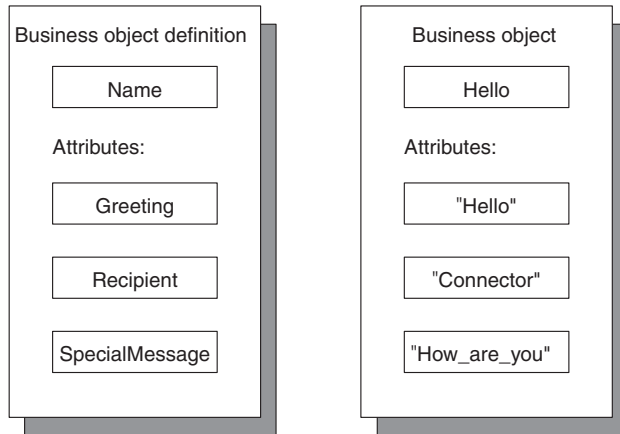
Figure 29. Hello business object

To create a business object definition manually:

1. Start Business Object Designer Express; for more information, see "Starting Business Object Designer Express" on page 46.

2. Click **File -> New**.

   Business Object Designer Express displays the New Business Object dialog box. Figure 30 shows the version of the New Business Object dialog box you see if you are running Business Object Designer Express from System Manager. If you are not running Business Object Designer Express from System Manager, the **Create in Project** list is omitted from the dialog box.
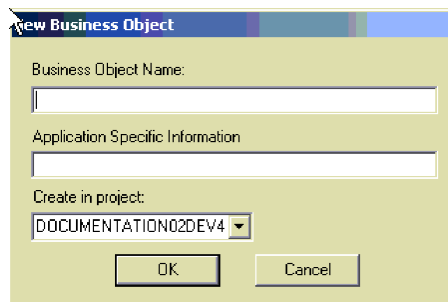
Figure 30. New Business Object dialog box

3. Enter the name Hello for the new business object definition.

   Names are generally case-sensitive, so type the name exactly as shown here.

   Note: The name of a business object definition can contain only alphanumeric characters and underscore (_). This name must use *only* characters defined in the code set associated with the U.S. English locale (en_US).

4. Leave the **Application Specific Information** box empty and click OK.

Business Object Designer Express displays the business object definition dialog box, as illustrated in Figure 31..
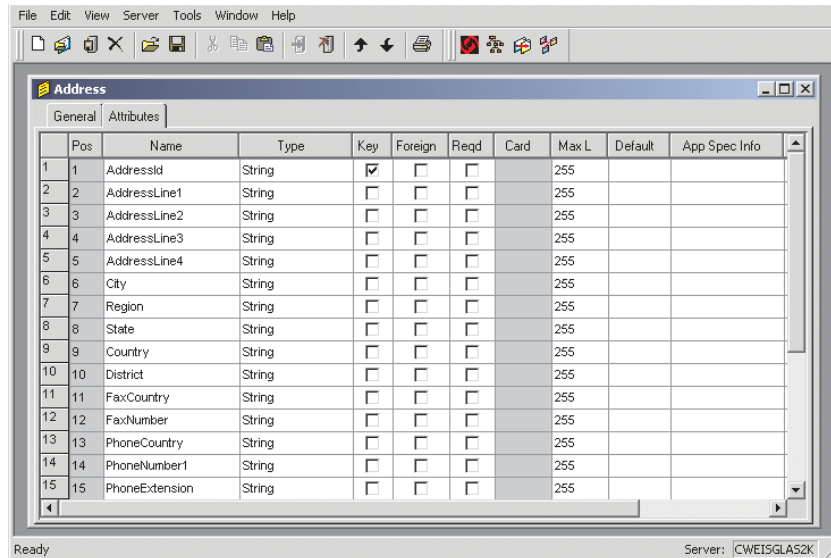
Figure 31. Initial display of a new business object definition

> **Note:** There may be minor differences in the Business Object Designer Express interface. However. the basic functionality of the tool is the same.

Adding attributes

Each piece of information in the business object is represented by an *attribute* in the `Hello` business object definition. You must provide the attribute definitions for the `Hello` business object. As illustrated in Figure 31,, Business Object Designer Express automatically adds an entry for the required end-of-object marker, `ObjectEventId`.

> **Important**
>
> Do not delete, change, or move the `ObjectEventId` attribute. This attribute is reserved for the WebSphere business integration system's internal use. Business Object Designer Express automatically moves this attribute when you save the definition.

The row for each attribute defines the attribute's properties. For information on the attribute properties, see "Business object attributes and attribute properties" on page 4.

As Figure 29 on page 56 shows, the `Hello` business object definition has the following attributes: `Greeting`, `Recipient`, and `SpecialMessage`. Define the attributes and their properties, one at a time.

Adding the Greeting attribute: To add the `Greeting` attribute:

1. Type the attribute name `Greeting` in the **Name** column of the first available empty row, which is 2 for the first attribute.

   > **Note:** This attribute name must use *only* characters defined in the code set associated with the U.S. English locale (`en_US`).

2. Click the **Type** column and select **String** for the attribute type. The type of an attribute is its data type.

Tip:

> If you have other business objects opened in Business Object Designer Express, their names appear in the **Type** list. Displaying existing business objects among the choices for **Type** allows you to create a hierarchical business object with an attribute whose type is another business object.
>
> When InterChange Server Express and System Manager are running, then every business object definition in the Integration Component Library you are working from is automatically displayed in this list.

3. Skip the **Key**, **Foreign**, **Reqd** (or **Required**), and **Card** columns.

   These columns specify whether the current attribute is the business object's primary or foreign key, whether the attribute's value is required, and whether the attribute represents a child business object or objects. For an explanation of these properties, see Chapter 2, "Business object design," on page 17.

4. In the **Max Length** box, leave the default value of 255.

   This box specifies the maximum number of bytes available for this attribute's value.

5. In the **Default** box, type `Hello`.

   This specifies the value to use if no other value is supplied for the attribute at run time.

   You have now defined the following properties for the `Greeting` attribute:

Name:	Greeting
Type:	String
Maximum length:	255
Default value:	Hello

6. Ignore all other columns and click the **Name** column of the third row.

Adding the Recipient attribute: The second attribute, `Recipient`, is a string.

In InterChange Server Express, the `SampleHello` collaboration object uses this attribute as follows:

- The connector sets the value to `Collaboration` when it sends a message to the collaboration.
- The collaboration sets the value to `Connector` when it sends a message to the connector.

At least one attribute in each business object definition must be a *key attribute*. A key attribute contains a value by which the WebSphere business integration system uniquely identifies instances of the business object. Make the `Recipient` attribute the key attribute.

To add the `Recipient` attribute, type the text `Recipient` in the **Name** column, and follow the steps for adding the `Greeting` attribute, using the following properties:

Name:	Recipient
Type:	String
Maximum length:	255
Default value:	Collaboration

Key:	Yes (A check mark appears in the Key column)

Leave the other columns blank and click the**Name** column of the fourth row.

Adding the SpecialMessage attribute: The third attribute, `SpecialMessage`, is a string.

In InterChange Server Express, the `SampleHello` collaboration expects the value of this attribute to be entered by the system administrator or another person with access to the collaboration configuration properties after the collaboration object has been created. The collaboration dynamically obtains the value of the configuration property and appends it to the message.

To add the `SpecialMessage` attribute, type the text `SpecialMessage` in the **Name** column, and follow the steps for adding the `Greeting` attribute, using the following properties

Name:	SpecialMessage
Type:	String
Maximum length:	255

Leave the other columns blank.

The **Attributes** tab now displays three user-defined attributes: `Greeting`, `Recipient`, and `SpecialMessage`. Figure 32 illustrates the `Hello` business object's attributes.

Figure 32. New business object definition with attributes

Changing attribute order

You can graphically change the sequence order of attributes in the business object definition. For example, to place the key attribute, `Recipient`, above the `Greeting` attribute, click the first (leftmost) column and drag the cursor up one row.

Specifying the supported verbs

You must now specify the verbs that this Hello business object supports. These
verbs represent the triggering events that the business object sends to InterChange
Server Express. Click the **General** tab of the Hello business object definition dialog
box to display the window in which you specify the verbs. Figure 33 illustrates this
tab.
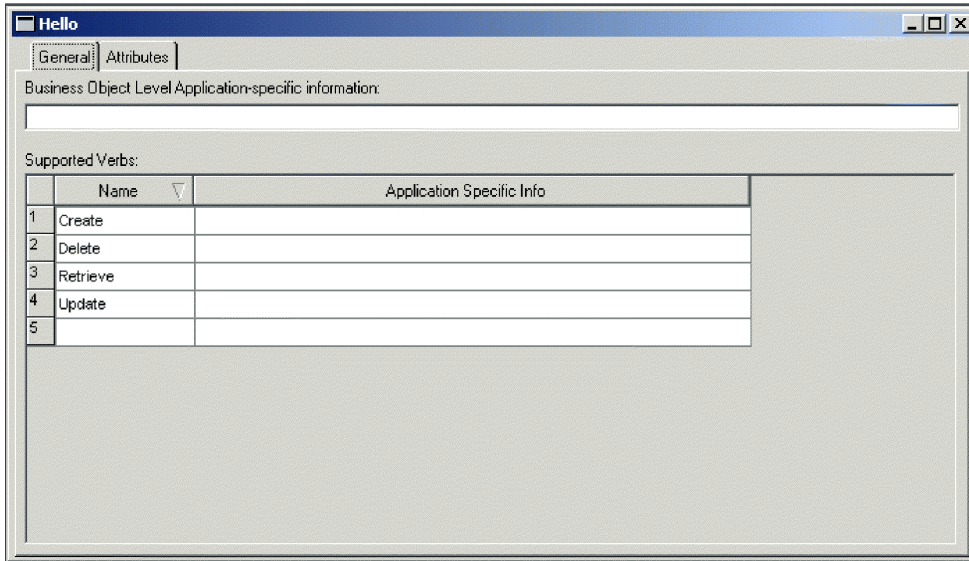
Figure 33. General editing tab

The business object supports the four default verbs—Create, Delete, Retrieve, and
Update; they appear on the **General** tab by default. For the purposes of this
tutorial, only one triggering event is supported: Create. Therefore, change the
business object definition to support only this verb.

Important: You must specify at least one verb for each business object definition.

Note: The name of a verb can contain only alphanumeric characters and
underscore (_). This name must use *only* characters defined in the code set
associated with the U.S. English locale (en_US).

To indicate that the Hello business object supports only the Create verb, you can
either delete the remaining verbs simultaneously or individually.

Deleting multiple verbs: To delete the Delete, Retrieve, and Update verbs:
1. Select the Delete verb and, while pressing the Shift key, click the Update verb.
2. Press the Delete key.

Deleting individual verbs: To delete each verb individually:
1. Click the number to the left of the Delete line in the **Supported Verbs** table.
   The row is selected.
2. Press the Delete key.
3. Repeat steps 1 and 2 for the Retrieve and Updateverbs in the **Supported Verbs**
   table.
4. Leave the **Application Specific Info** box blank for the Create verb.

You have finished the definition for the `Hello` business object. This is a good time to save your changes by clicking **File -> Save**. If you are using an ICL-based project, the definition is saved to the ICL. If you are using a local project, you will be prompted to specify a file name and local directory in which to save the definition.

Creating a hierarchical business object definition manually

This section describes how to create a hierarchical business object definition by defining an attribute that represents a child business object or an array of child business objects.

Because the previous section explains how to define a simple attribute and supported verbs, this section explains only the definition of an attribute that represents a child business object. This example creates a business object named `HierarchicalBO` that has two attributes:

- An attribute named `Key` that serves as the required business object key.
- An attribute named `Addr` that represents the `Address` business object with cardinality 1.

To manually create a hierarchical business object definition:

1. Open Business Object Designer Express.
2. Click **File -> New**.

   Business Object Designer Express displays the New Business Object dialog box, as illustrated in Figure 30 on page 56.
3. Type the name `HierarchicalBO` for the new business object definition.
4. Leave the **Application Specific Information** column empty and click **OK**.

   Business Object Designer Express displays the business object definition dialog box, as illustrated in Figure 31 on page 57.
5. Create a key attribute in the first available empty row, which is 2 for the first attribute. Name it `Key`, specify any simple data type, and click the **Key** column.
6. Create the next attribute in the next available empty row, which is 3. Name it `Addr`.
7. Click the **Type** list and select **Address** for the attribute type.

   Note: If the child business object does not exist in the list, you can create it now by selecting **New business object** in the **Type** list. You must save the new child business object before you can complete this step.
8. Skip the **Key**, **Foreign**, and **Reqd** (or **Required**) columns. Click the **Card** list and select **1**.
9. Ignore all other columns. Define supported verbs, and save the definition.

Deleting a business object definition

You can delete a business object definition using Business Object Designer Express or System Manager, in InterChange Server Express. This section describes:

- "Deleting a definition using Business Object Designer Express" on page 62
- "Deleting a definition using System Manager" on page 63

> **Important**
>
> You can delete business object definitions from an integration component library through System Manager (if you are using InterChange Server Express and System Manager is running) or from a project in Business Object Designer Express. You cannot use the Delete function in Business Object Designer Express or in System Manager to delete local files that contain business object definitions. To delete local files, use the tools provided by Windows.

Deleting a definition using Business Object Designer Express

To delete a business object definition from a project using Business Object Designer Express, do the following:

1. Open Business Object Designer Express.

2. From the list of business object definitions in the project, select the name of the definition you want to delete.

3. To select multiple names, do one of the following:

   - To select consecutive names, click the first name and, while pressing the Shift key, click the last name.

   - To select non-consecutive names, press the Ctrl key and click each name.

4. After selecting the definitions to be deleted, right-click and then click **Delete**.

   - Business Object Designer Express displays the **Deleting business object** confirmation message. Click **Yes** to delete the business object definition you selected in Step 2 or to delete all the business object definitions you selected in Step 3.

   - If the business object definition has dependencies with other business objects, Business Object Designer Express displays a collapse delete confirmation message.
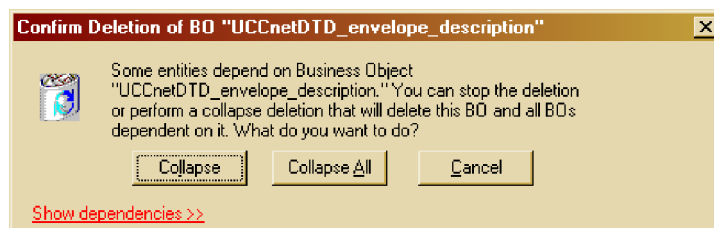
5. If dependencies exist, select the **Show dependencies** link. All dependencies with other business objects are listed for the business object definition that you

want to delete.
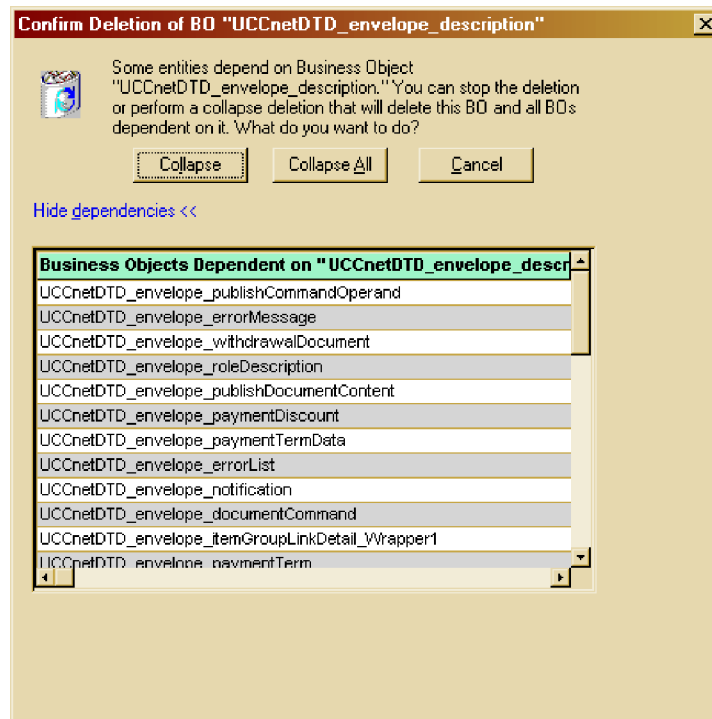
6. Do one of the following:
   - Click **Collapse** to delete the business object definition that you selected in Step 2 and all of the business objects that depend on it.
   - If you selected multiple business objects in Step 3, click **Collapse All** to delete the business objects that you selected and all of the business objects that depend on each of those business objects.
   - Click **Cancel** to cancel deleting the business object definition and its dependencies.

Deleting a definition using System Manager

To delete a business object definition using System Manager, do the following:

1. Start System Manager.
2. Expand **Integration Component Libraries** and then expand the integration component library from which you want to delete a business object definition.
3. Open the business objects folder and select the name of the business object definition to delete.
4. Delete the business object definition by doing either of the following:
   - Click the **Delete** toolbar icon.
   - Right-click the business object definition and select **Delete**.
5. When prompted whether you want to delete, click **Yes**.
6. If the business object definition has dependencies with other business objects, System Manager notifies you with an error message. You must use Business Object Designer Express to remove these dependencies before you can delete the business object definition with System Manager.

Using an Object Discovery Agent to create a business object definition

This section describes how to use an Object Discovery Agent (ODA) to generate business object definitions for application-specific business objects. An ODA is an optional component of an adapter. When you install a pre-defined adapter that has an ODA, its ODA is installed automatically. If you are developing a custom adapter and you want to use an ODA to create business object definitions, you can use the Object Discovery Agent Development Kit (ODK) to develop one. For more information about developing a custom ODA, see Chapter 5, "Developing an Object Discovery Agent," on page 87.

To configure and run the ODA, use the Business Object Wizard in Business Object Designer. Business Object Wizard is a graphical user interface to ODAs that manages the discovery and content-generation process. This section provides the following information:

- "Before using an ODA"
- "Using the ODA to create business object definitions" on page 67
- "Entering values and saving a profile" on page 75
- "Setting up logging and tracing" on page 75
- "Moving through the source-node hierarchy" on page 78
- "Providing additional information" on page 82
- "Using multiple ODAs simultaneously" on page 83

Before using an ODA

Before you run an ODA, verify that the following steps have occurred:

- System startup files are available and correct.
- The ODA has been started.
- Business Object Designer Express has been started.

System startup files

For the ODA to start, you need to verify that your system has the required files for the ODA. When you install a pre-defined adapter that has an ODA, these ODA system startup files should be installed automatically. If you are developing a custom adapter with a custom ODA, these ODA system startup files should be created as part of the ODA development process. However, IBM recommends that you confirm that the startup script exists and is correct for your ODA:

Each ODA requires a *startup script*, which begins execution of the ODA. Before you start an ODA for the first time, you must make sure that the variables are correctly set within the startup script. Open for editing the shell (start_*ODAname*.sh) or batch (start_*ODAname*.bat) file and confirm that the values described in Table 10 are correct.

Table 10. ODA shell and batch file configuration variables

Variable	Explanation	Example
set AGENTNAME	Name of the ODA	set AGENTNAME=*ODAname*
set AGENT	Name of the ODA's jar file	**WINDOWS:** set AGENT = %*ProductDir*%\ODA*srcDataName**ODAname*.jar
set AGENTCLASS	Name of the ODA's Java class	set AGENTCLASS=com.ibm.oda.*srcDataName*.*ODAname*

For information on the ODA name (*ODAname*) and its source-data name (*srcDataName*), see "Naming the ODA" on page 159.

Starting the ODA

You can start ODA with the following startup script for the windows operating system.

> **Windows**
>
> ```
> start_srcDataNameODA.bat
> ```

You configure and run the ODA using the Business Object Wizard in Business Object Designer Express. Business Object Wizard locates each ODA by the name specified in the AGENTNAME variable of each script or batch file.

Note: For information on how to start multiple instances of the ODA, see "Using multiple ODAs simultaneously" on page 83.

Starting Business Object Designer Express

Once you start the ODA, you must open Business Object Designer Express to configure and run it. For information on the ways to open Business Object Designer Express, see "Starting Business Object Designer Express" on page 46. To run an ODA, Business Object Designer Express provides Business Object Wizard, which guides you through each step.

To start Business Object Wizard, do the following:

1. Open Business Object Designer Express using one of the methods listed in Table 9 on page 46.
2. Click **File -> New Using ODA**.

Business Object Wizard begins displays the first dialog box in the wizard, Select Agent. Table 11 summarizes the steps of Business Object Wizard.

Table 11. Steps of Business Object Wizard

Task	Step in Business Object Wizard
1. Select the desired ODA	Step 1: Select Agent
2. Obtain the configuration properties, including those that describe the data source to open.	Step 2: Configure Agent
3. Obtain the source data for which the ODA generates the content.	Step 3: Select Source
4. Confirm that the selected source nodes are those desired for content generation.	Step 4: Confirm Source Nodes
5. Initiate the content-generation process.	Step 5: Generating Business Objects, Business Object Properties
6. Save the business object definitions in a user-specified format.	Step 6: Save Business Objects

For an example of how Business Object Wizard runs an ODA, see "Using the sample ODA" on page 66.

Using the sample ODA

IBM provides a sample Object Discovery Agent that converts Roman-army soldiers (in XML format) to business object definitions. To familiarize you with using an ODA, the following step-by-step description of generating business object definitions uses this sample ODA.

Note: For information on the location and files of this sample ODA, see "Development support for ODAs" on page 97.

This section includes the following tasks:
- "Starting the sample ODA"
- "Using the ODA to create business object definitions" on page 67

Starting the sample ODA

If you have installed the Adapter Development Kit (ADK), the sample ODA and the file to run it are located in the `DevelopmentKits\Odk\Samples` directory in the product directory. The file to run the sample ODA depends on your operating-system environment, as Table 12 shows.

Table 12. Startup script for a sample Roman Army ODA

Operating system	Startup script
Windows	`start_Agent4.bat`

Note: The sample Roman Army ODA provides five versions to illustrate various features of an ODA. This section runs the fourth version of this sample ODA, which uses the `start_Agent4` startup script and the `ArmyAgent4` class file.

Because the sample Roman Army ODA provides five versions of itself, all startup scripts call one common startup script called `start_AgentX`, passing it the name of the ODA class (which is assigned to the `AGENTCLASS` configuration variable in `start_AgentX`). Therefore, the `start_Agent4` startup script should contain a call to `start_AgentX`, passing it the following path as the name of the ODA class:

`com.ibm.btools.ODK2.RomanArmy.ArmyAgent4`

To verify configuration variables for this sample ODA, check the `start_AgentX` batch or script file to confirm that your confirmation variables match those in Table 13. If you move any of the files that version 4 of the sample Roman Army ODA uses, make sure you change the corresponding configuration variable.

Table 13. Configuration variables for the sample Roman Army ODA

Variable	Value for sample Roman Army ODA
AGENTNAME	`set AGENTNAME=Roman`
AGENT	**WINDOWS:** `set AGENT =` `%ProductDir%\DevelopmentKits\Odk\Samples\RomanArmy\ArmyODA.jar`
FILE_LOCATION	**WINDOWS:** `set FILE_LOCATION =` `%ProductDir%\DevelopmentKits\Samples\Odk\RomanArmy\RomanArmy.xml`

> **Important**
>
> You must start the sample ODA before you try to connect to it through Business Object Wizard. Business Object Wizard can only locate those ODAs that have been started.

Using the ODA to create business object definitions

To start Business Object Wizard, do the following:

1. Open Business Object Designer Express using a method listed in Table 9 on page 46.

2. Click **File -> New Using ODA**.

   Business Object Wizard displays the first dialog box, Select Agent, shown in Figure 34..
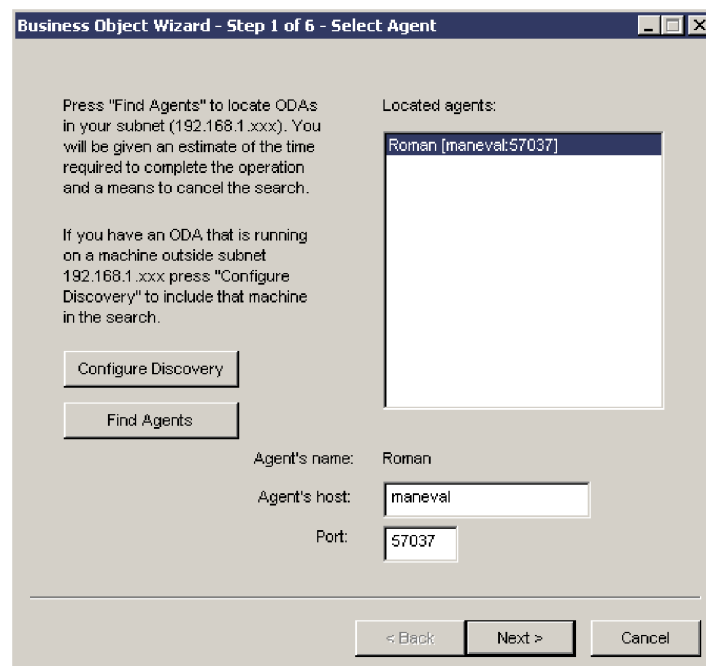
Figure 34. Select Agent dialog box

3. To select the ODA to which Business Object Wizard connects:

   a. Click **Find Agents** to display ODAs that are currently running (those that have been started with their startup scripts) in the **Located agents** list.

      Note: If Business Object Wizard does *not* locate your desired ODA, check the startup of the ODA.

      Business Object Wizard identifies each running ODA by the name specified for the AGENTNAME variable of its startup script or batch file. This sample ODA is named Roman.

   b. Select the desired ODA from the **Located agents** list. Business Object Wizard displays your selection as **Agent's name**. Alternatively, you can find the ODA by specifying its host name and port number.

4. Click **Next**. Business Object Wizard attempts to connect to the specified ODA. If the ODA has been started, Business Object Wizard displays a status window as it connects to the ODA, as Figure 35 shows.
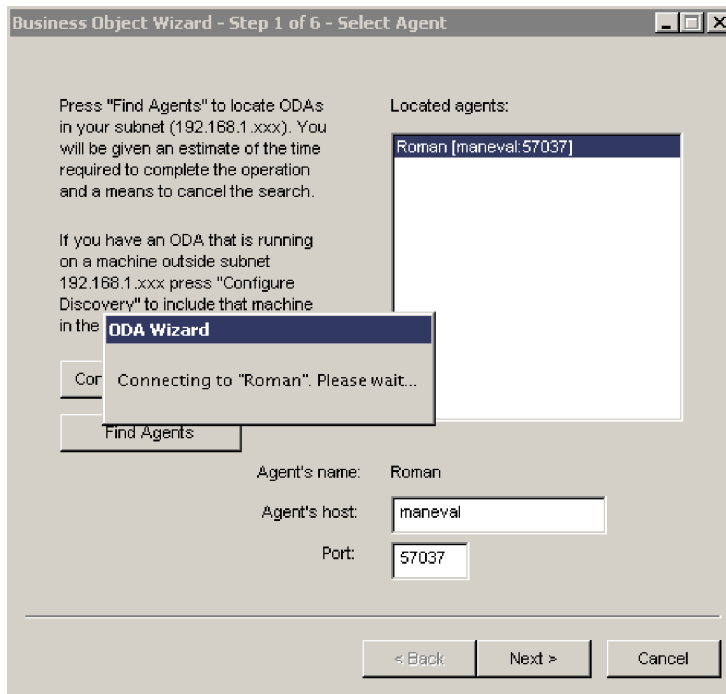
Figure 35. Connecting to an ODA.

5. After Business Object Wizard is connected to the ODA, it displays the second wizard dialog box, Configure Agent, which is shown in Figure 36. This dialog box displays the ODA configuration properties required to access the data source and initialize the ODA.
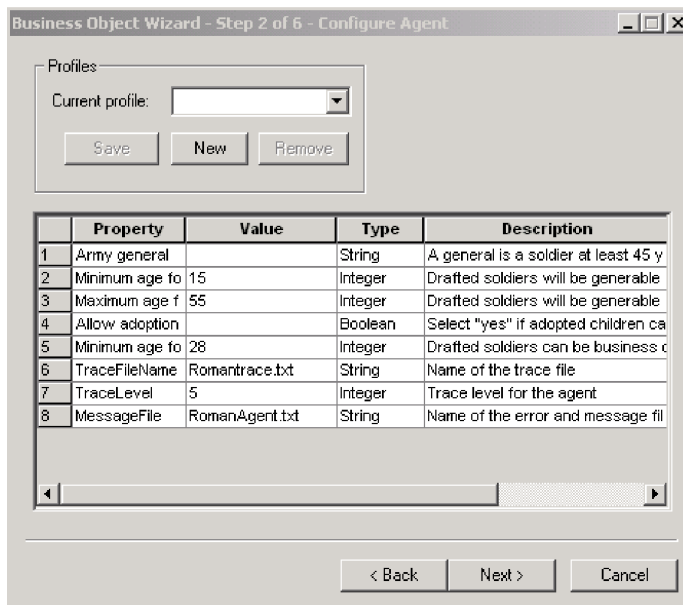
Figure 36. Configure Agent dialog box

6. Specify ODA configuration values or select a profile to display previously saved values. One of the required configuration areas for the ODA is to set up the logging and tracing. For more information, see "Setting up logging and tracing" on page 75.

   The first time you use a particular ODA, you specify values for each of its configuration properties. After doing so, you can save the property values in a named profile by clicking **Save**. The next time you use the same ODA, you can select the saved profile from the **Select profile** box. For more information, see "Entering values and saving a profile" on page 75.

7. Click **Next**. Business Object Wizard displays the third wizard dialog box, Select Source, which is shown in Figure 37. The Select Source dialog box displays the *source-node hierarchy*, which is a tree structure with the top-level objects at the top and child objects underneath. In the initial display, the Select Source dialog box usually displays only the top-level source nodes.

Important

If the ODA is unable to proceed when you click Next, verify that the ODA message file you have specified for the `MessageFile` configuration property exists in the `ProgramDir\ODA\messages` directory. For this sample ODA, the default name of this message file is `RomanAgent.txt`. For more information, see "Specifying the ODA message file" on page 77.
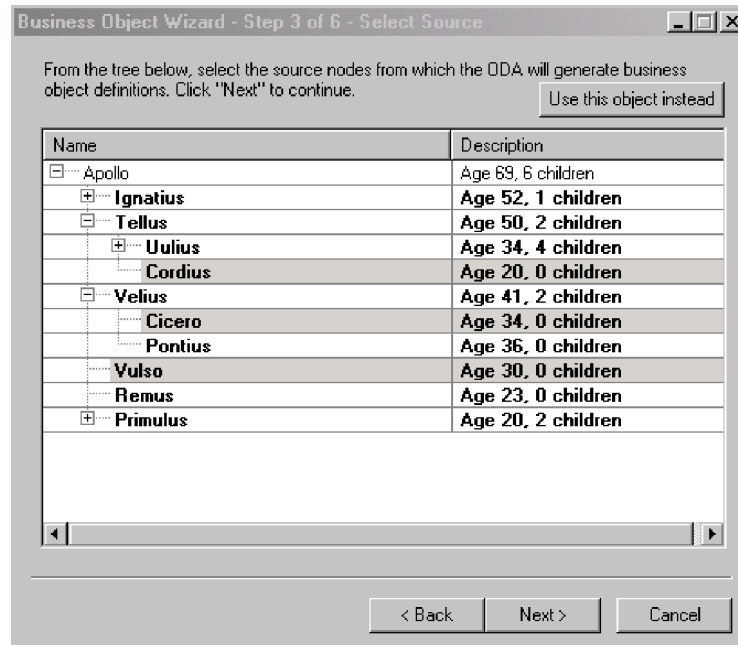
Figure 37. Initial Select Source dialog box

The nodes of the source-node hierarchy can be table names, business object names, schema, or functions, depending on the ODA's data source. This sample ODA generates nodes from objects within an XML file called `RomanArmy.xml`. Figure 37 shows the single top-level source node for the Roman general specified for the `Army general` configuration property (see Figure 36 on page 68).

8. Select objects in the source-code hierarchy for which you want the ODA to generate business object definitions. To select one source node, click on the node name. To select additional nodes, use the `Ctrl` key. In Figure 38, several source nodes have been expanded and three source nodes (which correspond to XML objects) have been selected.
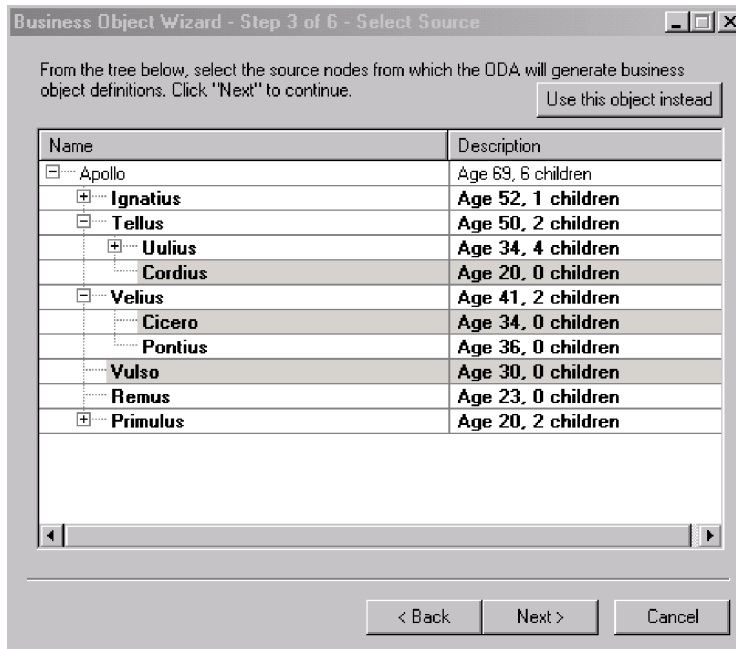
Figure 38. Select Source dialog box with source nodes expanded and selected

To expand a source node to display its child nodes, do either of the following:

- Click the + symbol to the left of the node name.
- Right-click the node name. Business Object Wizard displays the pop-up menu shown in Figure 39.. To expand the selected node, click **Retrieve all items**. Business Object Wizard displays the next level of source nodes: the child nodes for the expanded parent node. To open lower levels, repeat this process.
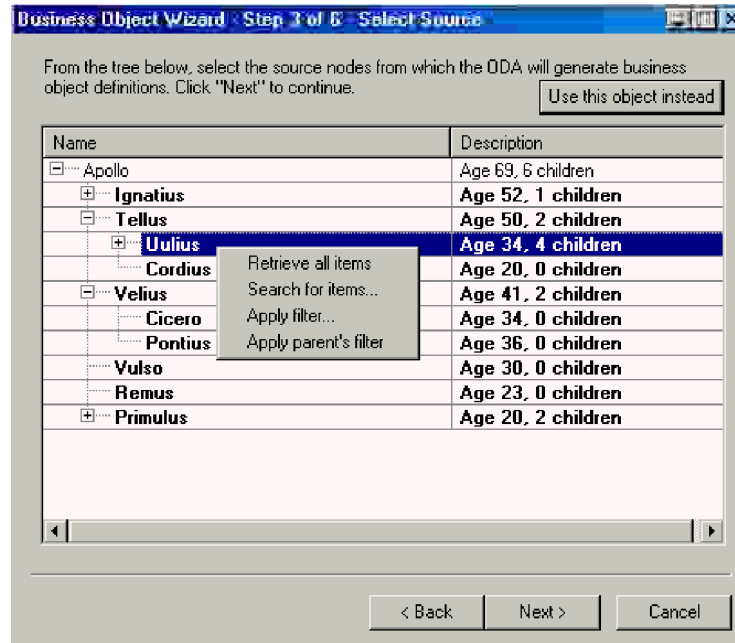
Figure 39. Right-clicking a node

> **Note:** Business Object Wizard provides several other mechanisms to move
> through the nodes of the source-node hierarchy. For more information,
> see "Moving through the source-node hierarchy" on page 78.

9. After you select the source nodes for which business object definitions are to
   be generated, click **Next**. Business Object Wizard displays the fourth wizard
   dialog box, Confirm Source, which is shown in Figure 40. This dialog box
   allows you to confirm your selection of source nodes. Selected source nodes
   are displayed in a bold font. In Figure 40, the source nodes for **Cordius**,
   Cicero, and **Vulso** are selected.
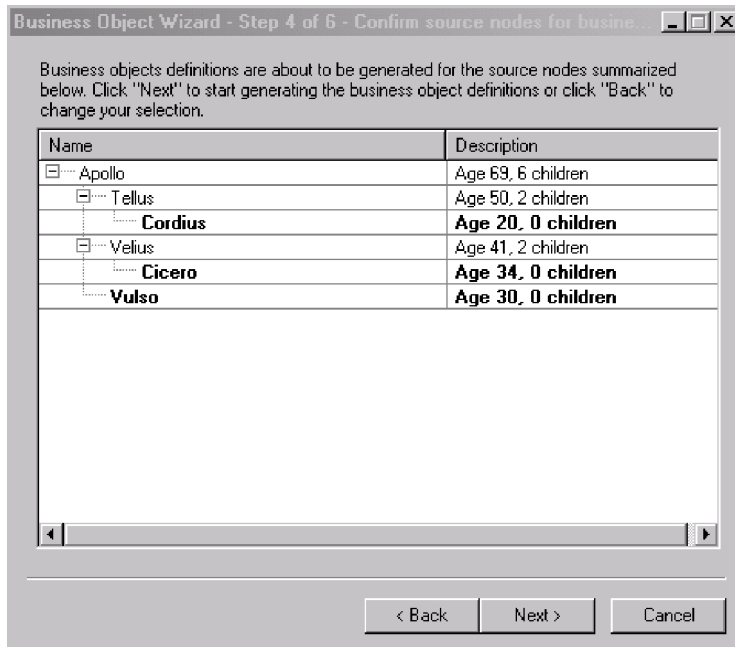
Figure 40. Confirming the objects for which to generate business object definitions

> If your selection is not correct, click **Back** to return to the previous dialog box and make the necessary changes.

10. When your selection is correct, click **Next**. Business Object Wizard displays the wizard's fifth screen, Generating Business Objects, which is shown in Figure 41. This screen informs you that the ODA is generating the business object definitions.
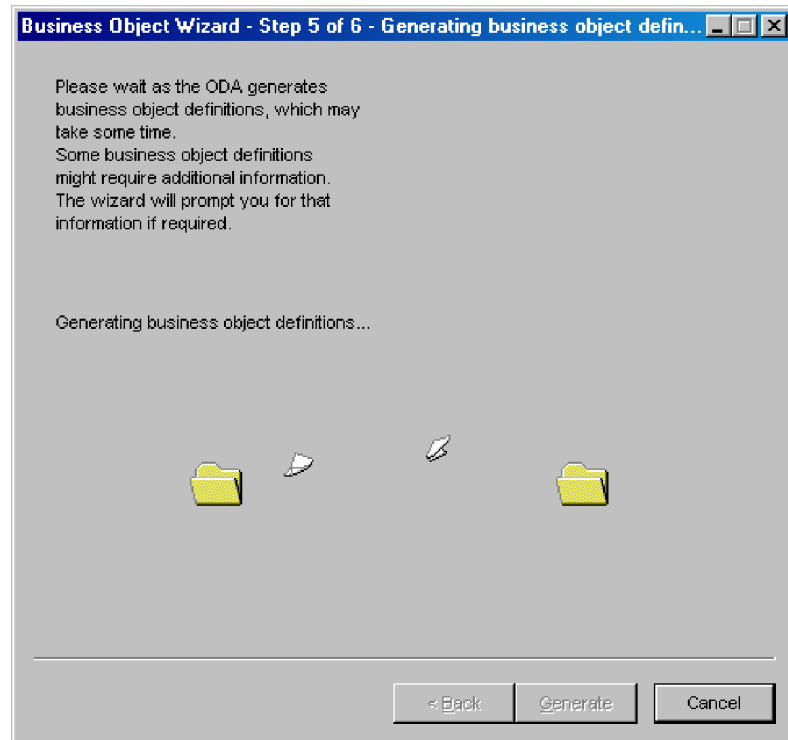
Figure 41. Generating the definitions

If the ODA needs additional information, Business Object Wizard prompts you for this information by displaying the BO Properties dialog box. However, this sample ODA does not require additional information. For more information about the BO Properties dialog box, see "Providing additional information" on page 82.

11. After the ODA completes the generation of business object definitions, Business Object Wizard displays the final dialog box in the wizard, Save Business Objects, shown in Figure 42. This dialog box offers the following options to save the business object definitions that the ODA has generated:

- Save the business object definitions to an ICL-based project if Business Object Designer Express is running from System Manager.
- Save the business object definitions to a file (for InterChange Server Express).
- Open the business object definitions for editing in Business Object Designer Express.
- Shut down the ODA.

> **Important**
>
> If the ODA generates a business object definition from a data-source object that does *not* identify a key element, this business object definition will *not* have a key attribute. Every business object must have *at least one key*. If the ODA might have generated business object definitions that do not include keys, you might want to choose the "Open the new BOs in separate windows" option instead of saving the business object definitions. Within Business Object Designer Express, you can verify that each business object definition has a key attribute, adding one if none exists. Business Object Designer Express does not allow you to save any business object definition that does not include a key.
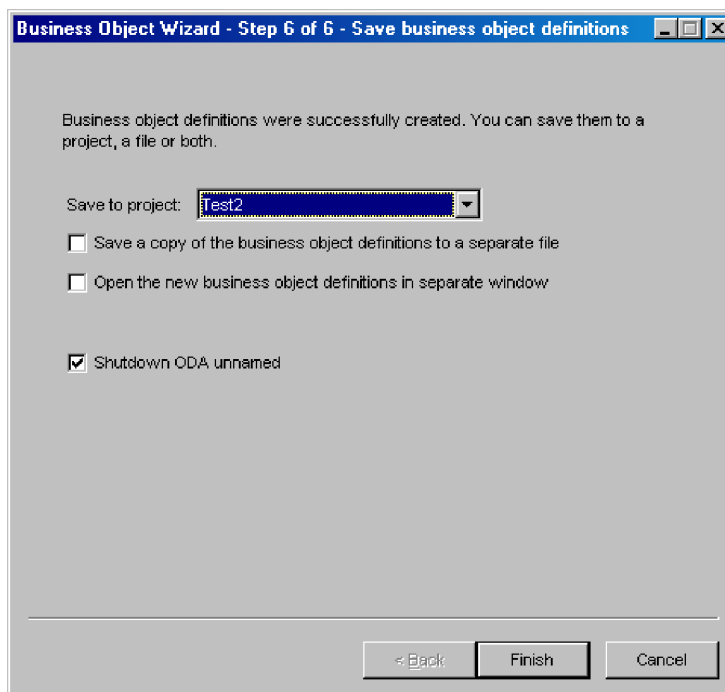
Figure 42. Saving the business object definition

Click **Finish** to save the business object definitions or **Cancel** to exit without saving these definitions. In either case, Business Object Wizard disconnects from the ODA. This dialog box also provides the option to have Business Object Wizard shut down the ODA after it disconnects. If you no longer need to use the ODA, select this option.

After you click **Finish**, if you have selected the option to save the business object definitions to a file, a browse window opens and allows you to specify the name of this file, where to save it, and what format to use (text file or InterChange Server Express specific format).

You have now successfully created business object definitions using an Object Discovery Agent.

Entering values and saving a profile

You can save a particular set of ODA configuration values in a profile so that they can be available for future uses of the ODA. To save a profile:

1. On the Step 2, Configure Agent dialog box of the Business Object Wizard, click the **New** button under **Profiles**.

   Note: To base a profile on an existing one, locate the desired profile in the profile drop-down list. Do *not* click the New push button.

2. Enter a name for the profile in the **Current** list (see Figure 36 on page 68 for an illustration).

   Note: If you are basing a profile on an existing one, overwrite the name of the existing profile in the profile drop-down list.

3. Enter the desired configuration values in the **Configure Agent** table.

4. Click the **Save** button.

   Business Object Wizard saves the profile under the following directory:

   ```
   C:\Documents and Settings\All Users\Application Data\CrossWorlds\
       BusObjDesigner\profiles.bod
   ```

Setting up logging and tracing

As part of the configuration of the ODA, you must set up the logging and tracing. You specify the logging and tracing information for an ODA in the Configure Agent dialog box of Business Object Wizard. Business Object Wizard always provides the standard configuration properties (shown in Table 14) for an ODA.

Table 14. Standard ODA configuration properties.

Property name	Property type	Description
TraceFileName	String	Specifies the file into which the ODA writes trace information. For more information, see "Specifying the trace file and trace level" on page 75.
TraceLevel	Integer	Trace level enabled for the ODA. For more information, see "Specifying the trace file and trace level" on page 75.
MessageFile	String	Name of the ODA's error and message file. Use this property to verify or specify an existing file. For more information, see "Specifying the ODA message file" on page 77.

This section provides the following information:

- "Specifying the trace file and trace level"
- "Specifying the ODA message file" on page 77

Specifying the trace file and trace level

Figure 43 shows the Configure Agent dialog box in Business Object Wizard, in which you specify the name of the trace file and the trace level.
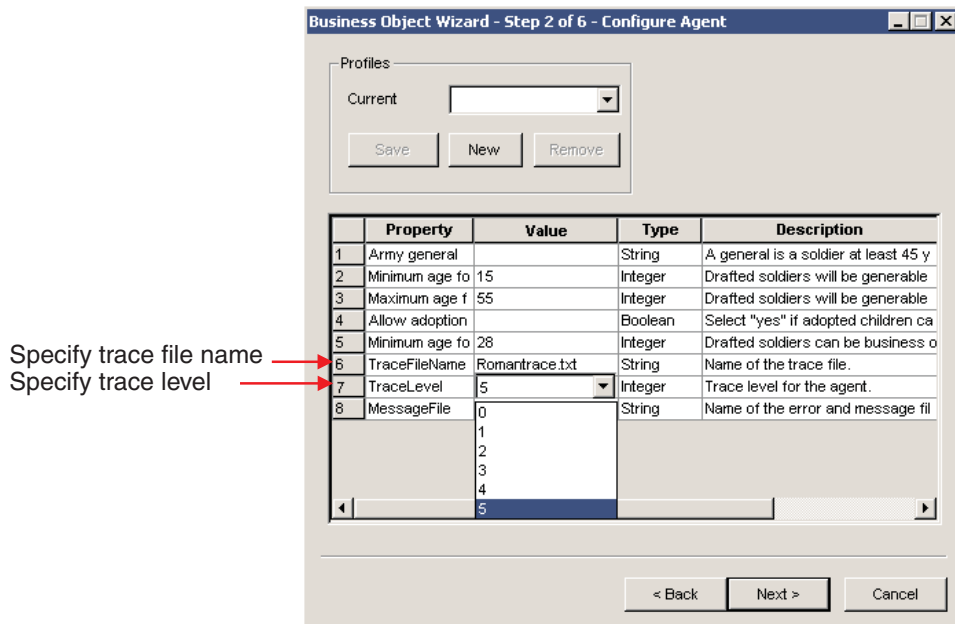
Figure 43. Specifying tracing information

Specifying a trace file: The `TraceFileName` configuration property specifies the name of the ODA's *trace file*. This file is the destination for *all* trace and error messages that the ODA logs. By default, the ODA run time names the trace file according to the following naming convention:

`ODAnametrace.txt`

In the preceding line, *ODAname* is the name that uniquely identifies the ODA. For more information, see "Naming the ODA" on page 159. For example, if the ODA is named `HTMLODA`, it generates a trace file named `HTMLODAtrace.txt`.

Note: Because the ODK API provides one method to log *both* trace and error messages, an ODA has only one file to hold both these kinds of messages. Therefore, although this file is called a trace file, it also contains any error messages that the ODA generates.

If the specified trace file does *not* exist, the ODA creates it in the ODA's runtime directory, which is the `ODA\`*srcDataName* subdirectory of the product directory. If the specified trace file already exists, the ODA appends to it. When configuring the ODA, you can use specify a different name for the trace file by resetting the `TraceFileName` property.

Setting the trace level: The `TraceLevel` configuration property specifies the ODA's *system trace level*. The ODA's trace method sends the specified message to the trace file when the message's trace level is less than or equal to this system trace level. Therefore, the system trace level determines the level of detail that the trace messages provide. Table 15 lists trace levels and their associated behavior.

Table 15. Trace levels

Level	Behavior
0	Writes error messages to the specified trace file.
1	Traces whenever a method is entered—useful for status messages and key information for each business object definition.

Table 15. Trace levels (continued)

Level	Behavior
2	Traces the agent properties and the values received.
3	• Traces the names of the business object. • Traces the business object properties and the values received.
4	• Traces the spawning of all threads. • Traces a message whenever a method is entered and exited.
5	• Indicates the initialization of the Object Discovery Agent and log the values retrieved for all the Object Discovery Agent properties. • Traces detailed status of each thread spawned by the Object Discovery Agent. • Traces the business object definition dump.

For information on how to generate trace messages within the ODA, see "Handling trace and error messages" on page 150.

Specifying the ODA message file

The `MessageFile` configuration property specifies the name of the ODA's *message file*. An ODA can store its error and trace messages in this ODA message file. It can then retrieve these messages by message number, instead of creating the message text itself. Isolating messages into the message file provides an easy way for ODA messages to be translated into the languages of the different locales the ODA can run in.

By default, the ODA run time names this message file according to the following naming convention:

ODAname`Agent.txt`

In the preceding line, *ODAname* is the name that uniquely identifies the ODA. For more information, see "Naming the ODA" on page 159. For example, if the ODA is named HTMLODA, the value of the `MessageFile` property defaults to `HTMLODAAgent.txt`. The message file must reside in the following message-file directory:

ProductDir`\ODA\messages`

> **Important**
>
> If the specified message file does *not* exist or does not exist in the message-file directory, the ODA generates a runtime exception. You must ensure that the message file (which `MessageFile` specifies) exists before you continue with the execution of the ODA.

If the ODA uses a different message file, set the `MessageFile` property to specify a different name for the trace file.

If you are using a non-US English locale, Business Object Wizard automatically looks for an ODA message file that includes the name of the locale in the file name, as follows:

ODAname`Agent_`*locale*`.txt`

where *locale* has the format "`ll_TT`", with `ll` as the two-character language name (in lowercase) and `TT` as the two-character country or territory name (in

uppercase). For example, if the ODA named `HTMLODA` has its message file localized to the Japanese locale, its message file would have the name:

```
HTMLODAAgent_ja_JP.txt
```

Note: When you are logged into a non-US English locale, you do *not* have to specify the non-US English name in the `MessageFile` property. For example, if you are using the HTML ODA, you set `MessageFile` to the US English file name (`HTMLODAAgent.txt`). If you are logged into a Japanese local, Business Object Wizard locates the correct message file for the Japanese locale: `HTMLODAAgent_ja_JP.txt`.

If you create multiple instances of the ODA script or batch file and provide a unique name for each represented ODA, you can have a message file for *each* ODA instance. For more information, see "Using multiple ODAs simultaneously" on page 83.

Moving through the source-node hierarchy

The Business Select Source dialog box in Business Object Wizard provides the following mechanisms for moving through the nodes of the source-node hierarchy:

- "Limiting display of child nodes"
- "Specifying an object path" on page 80
- "Associating an operating-system file" on page 80

Limiting display of child nodes

The ways to expand a source node given in step 8 on page 70 describe how to display *all* child nodes of an expandable node. To limit which objects are displayed, you can use either of the following menu items when right-clicking a node name (see Figure 39 on page 71):

- **Apply filter**
- **Search for items**

Using a filter: The **Apply Filter** menu item allows you to specify a *filter*, which can limit which of the currently selected source nodes opens. When you click this menu item, Business Object Wizard displays the Apply filter to node dialog box, as shown in Figure 44.
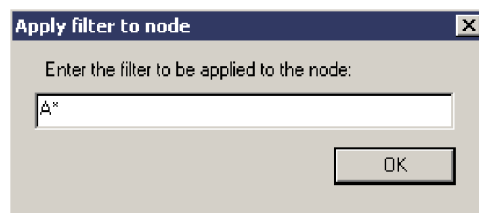
Figure 44. Specifying a filter to limit results

In the filter text, you can use the asterisk (*) character as a wildcard (to represent zero or more matching characters). This wildcard character can appear in any position and in as many positions as required. For example, `SAP*`, `*SAP`, `*SAP*`, or `*S*AP*`.

When you click **OK**, Business Object Wizard searches the currently retrieved child nodes of the parent node for those whose names match the filter text. When it expands this parent node, it displays only those child nodes whose names match this text.

Important: When Business Object Wizard receives a filter, it searches for matching child nodes of the parent node in the currently retrieved source node; that is, it does *not* search the data source for matching child nodes. To have Business Object Wizard search the data source, you can specify a search pattern. For more information, see "Specifying a search pattern."

For example, in the sample Roman ODA, the `Uulius` node has four child nodes: `Ares`, `Cronus`, `Atlas`, and `Metis`. If you apply the filter in Figure 44 to the `Uulius` node ("A*"), Business Object Wizard displays this node as shown in Figure 45 when you expand the node.
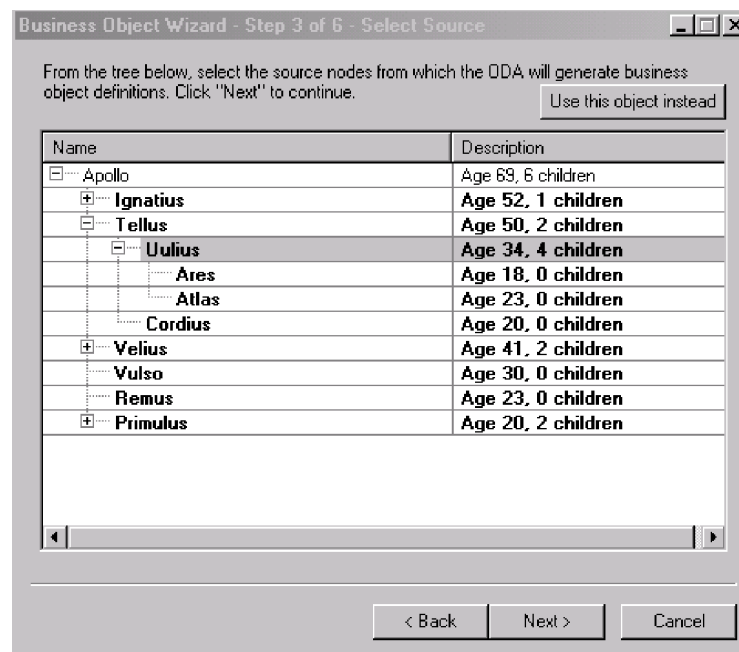
Figure 45. Filtered node after expansion

If you specify a filter at the top of a node and then expand the node, you can apply the same filter to child objects by right-clicking on the node and clicking **Apply parent's filter**. If you click **Retrieve all items** menu item, the parent node filter is applied to all elements.

Specifying a search pattern: The **Search for items** menu item allows you to specify a *search pattern*, which can limit which source nodes Business Object Wizard selects from the data source. When you click **Search for items**, Business Object Wizard displays the Enter a Search Pattern dialog box. Figure 46 on page 80 illustrates this dialog box.

Note: An ODA must support the search-pattern feature for the **Search for items** menu item to be enabled. If this menu item is not available, the ODA does *not* support search patterns.
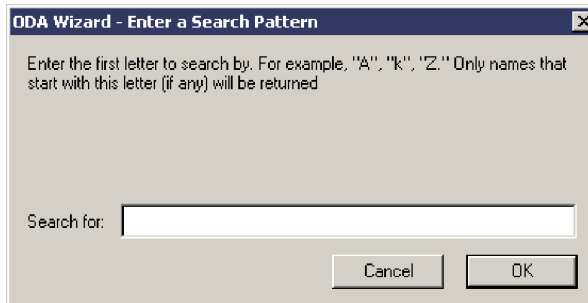
Figure 46. Specifying a search pattern to limit retrieval results

The Enter a Search Pattern dialog box provides a description of the search criteria that your search pattern can use. In Figure 46, the text in this dialog box specifies that the search pattern can consist of one letter. The ODA provides a customized description of the search criteria. Make sure that the search pattern you enter follows the described search criteria. Otherwise, the ODA throws an exception.

When you click **OK**, Business Object Wizard searches the data source for child nodes of the parent node whose names match the search pattern. When it expands this parent node, it displays only those child nodes whose names match this pattern.

Important: When Business Object Wizard receives a search pattern, it searches for matching child nodes of the parent node in the data source; that is, it retrieves a new tree node from the data source. It does *not* simply search the currently retrieved tree node for matching child nodes. To have Business Object Wizard search the currently retrieved tree node, you can specify a filter. For more information, see "Using a filter" on page 78.

Specifying an object path

Instead of moving through the source-node hierarchy, you can specify an exact path for the desired object. To do so, click **Use this object instead**, at the upper right of the Select Source dialog box. Business Object Wizard displays the Object Path dialog box, shown in Figure 47, in which you specify the path.
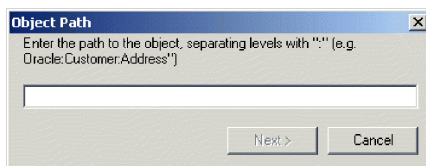
Figure 47. Specifying an object's path.

You specify the object path as the fully qualified path of the source node (from the top-level parent node down to the desired node). Node names within this path are separated with a colon (:).

Associating an operating-system file

To associate an operating-system file with the current node of the source-node hierarchy, right-click on a node and click **Associate files** (see Figure 48). When you associate a file with a source node, the ODA uses the file as the source for that source node's data (instead of using the ODA's data source).

Note: An ODA must support the associate-files feature for the **Associate files** menu item to be enabled. If this menu item is not available, the ODA does *not* support associating files with the current source node.
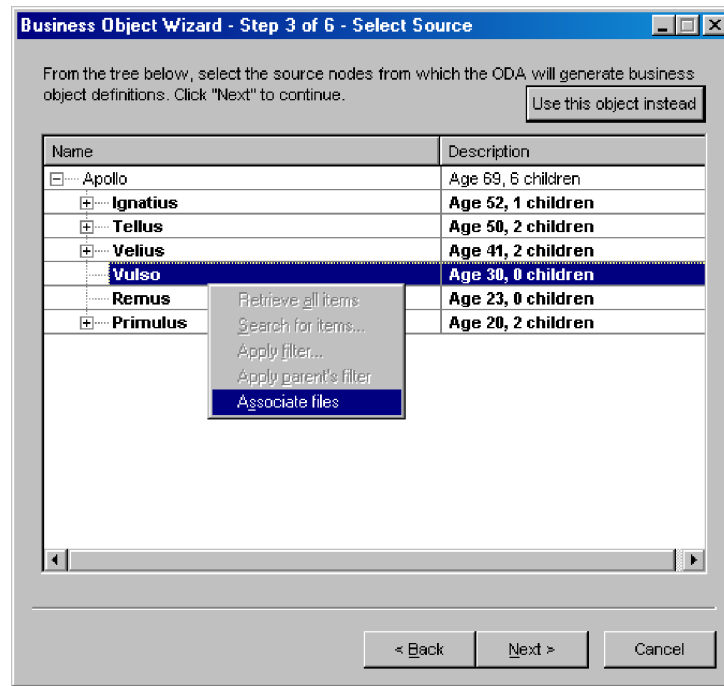
Figure 48. Associating a file with a source node

When you click the **Associate files** menu item, Business Object Wizard displays the Open window shown in Figure 49. From this window, you can browse the file structure and choose the file to associate with the current node.
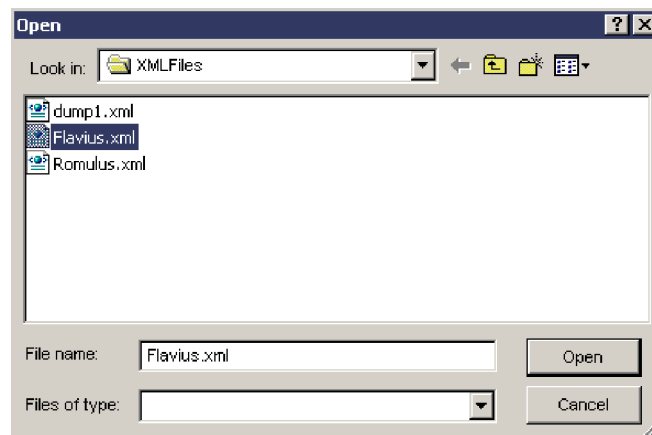
Figure 49. Open window for selecting the file to associate

After you have selected the file to associate with the source node, click **Open**. When Business Object Wizard returns control to the Select Source dialog box, the file you selected is displayed under the source node with which it is associated, as Figure 50 shows.
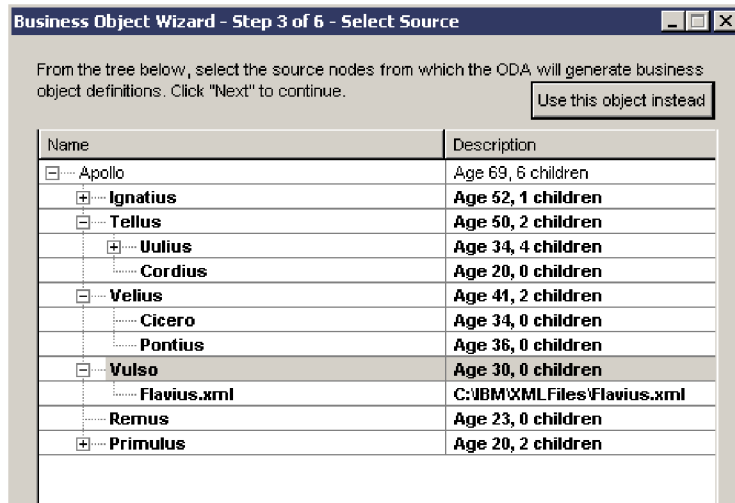
Figure 50. File associated with a source node

Providing additional information

In Step 5, Generating Business Objects, if the ODA needs additional information, the BO Properties dialog box opens, as shown in Figure 51..
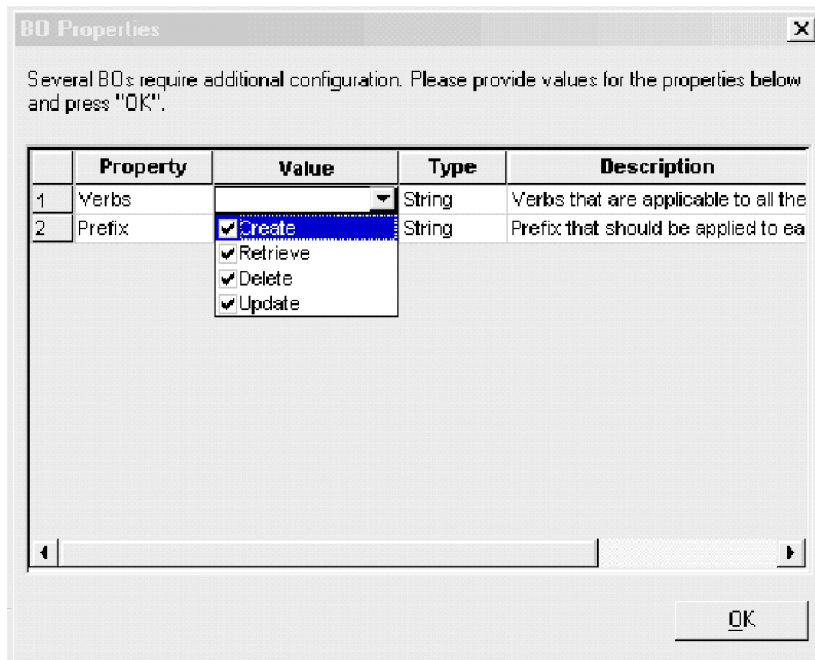
Figure 51. Providing additional information.

Note: If a cell in the BO Properties dialog box has multiple values, it appears to be empty when the dialog box first opens. Click the cell for a list of its values.

After you provide all required information in the BO Properties dialog box, click **OK**. The ODA continues with its generation of business object definitions.

Using multiple ODAs simultaneously

You can run multiple instances of an ODA either on the local host machine or a remote host machine. Each instance runs on a unique port. You can specify this port number when you start each ODA from within Business Object Wizard.

To run multiple Object Discovery Agents simultaneously in Business Object Designer, do the following:

1. Start each Object Discovery Agent by running its `start_ODAname.bat` or `start_ODAname.sh` files.
2. Open Business Object Designer.
3. Click **File -> New Using ODA**.

   The first dialog box in the Business Object Wizard, Select Agent, opens (see Figure 34 on page 67).
4. Click the **Find Agents** button to display currently running ODAs in the **Located agents** list. You can also find the ODA using its host name and port number.
5. Select the first ODA from the displayed list. Your selection is listed as **Agent's name**.
6. Click **File -> New Using ODA** again.
7. Click the **Find Agents** button to display currently running ODAs in the **Located agents** list, or find the ODA using its host name and port number.
8. Select the second ODA from the displayed list.
9. Proceed with the configuration of each ODA as described in step 4 of "Using the ODA to create business object definitions" on page 67.

If you create multiple instances of the ODA script or batch file and provide a unique name for each represented ODA, you can have a message file for *each* ODA instance. Alternatively, you can have differently named ODAs use the same message file. There are two ways to specify a valid message file:

- If you change the name of an ODA and do *not* create a message file for it, you must change the name of the message file in Business Object Wizard as part of ODA configuration. Business Object Wizard provides a name for the message file but does not actually create the file. If the file displayed as part of ODA configuration does not exist, change the value to point to an existing file.

- You can copy the existing message file for a specific ODA, and modify it as required. Business Object Wizard assumes you name each file according to the naming convention. For example, if the `AGENTNAME` variable (within the ODA startup script) specifies `HTMLODA`, the tool assumes that the name of the associated message file is `HTMLODAAgent.txt`. Therefore, when Business Object Wizard displays the file name for verification as part of ODA configuration, the file name is based on the ODA name. Verify that the default message file is named correctly, and correct it as necessary.

Part 2. Developing an Object Discovery Agent

Chapter 5. Developing an Object Discovery Agent

This chapter presents information on how to use classes defined in the Object Discovery Agent Development Kit (ODK) API to develop an Object Discovery Agent (ODA). An ODA works with Business Object Designer Express Business Object Wizard to develop business object definitions for a specific connector or data handler that works with a specific application, database, or filesystem.

The main topics of this chapter are:

- "Running an ODA"
- "Overview of the ODA development process" on page 95
- "Extending the ODA base class" on page 100
- "Determining the ODA generated content" on page 107
- "Starting the ODA" on page 101
- "Generating business object definitions as content" on page 110
- "Generating binary files as content" on page 133
- "Shutting down the ODA" on page 149
- "Handling trace and error messages" on page 150
- "Handling exceptions" on page 157

Running an ODA

At run time, running an ODA involves the following components:

- Business Object Designer Express provides a graphical interface in the form of a wizard to interact with the ODA: Business Object Wizard. The wizard displays a series of dialog boxes to obtain information that the ODA needs to generate the content.
- The *ODA runtime* is the intermediary component between Business Object Wizard and the ODA. It uses the classes of the ODK API and the ODK infrastructure to communicate with the ODA. It is the ODA runtime that you start with the ODA startup script.
- The *ODA* is the component that "discovers" source nodes in the data source and generates the content. The ODA receives information in the dialog boxes of Business Object Wizard from the ODA runtime. It then sends information (such as the generated content) to the ODA runtime, which sends it to Business Object Wizard.

Figure 52 shows the components of the ODA runtime architecture.
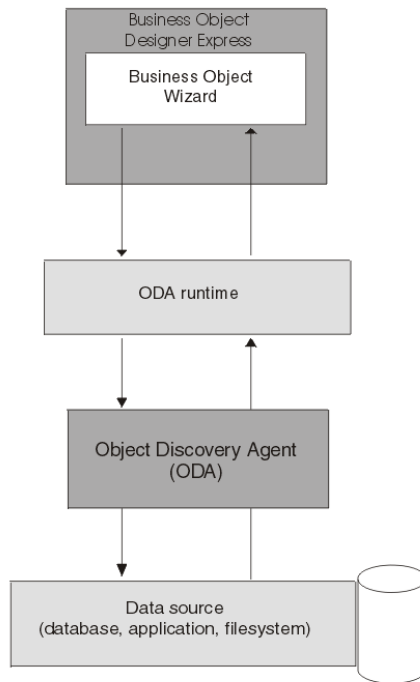
Figure 52. Object Discovery Agent Architecture

To generate the business object definitions, the ODA must take the following steps:

1. Obtain values for the ODA configuration properties (such as user name and database type) that the ODA requires to connect to the data source (such as an application, database, or filesystem).

2. Use these configuration properties to connect to the data source.

3. Obtain the list of source nodes for which business object definitions are to be created.

4. Discover the requirements for the data-source entity underlying the source node (as defined by an application, database table, filesystem, or DTD).

5. Generate business object definitions that meet the requirements of the WebSphere business integration system and the component that processes the business object, and return the business object definitions to users.

Note: In addition to business object definitions, an ODA can also generate files as content. For more information, see "Generating content" on page 91.

Table 16 summarizes the steps in the running of an ODA and the steps in Business Object Wizard that initiate them.

Table 16. Running the Object Discovery Agent

Task		Step in Business Object Wizard	For more information
1.	Select the desired ODA to start	Step 1: Select Agent	"Selecting the ODA" on page 89
2.	Obtain the ODA configuration properties, including those that describe the data source to open.	Step 2: Configure Agent	"Obtaining ODA configuration properties" on page 89
3.	Obtain the source data for which to generate the ODA content.	Step 3: Select Source	"Selecting and confirming source data" on page 91

Table 16. Running the Object Discovery Agent (continued)

Task		Step in Business Object Wizard	For more information
4.	Confirm the source data that you have selected.	Step 4: Confirm Source Nodes	"Selecting and confirming source data" on page 91
5.	Generate the business object definitions.	Step 5: Generating Business Objects	"Generating content" on page 91
		Business Object Properties	"Obtaining business-object properties" on page 93
6.	Save the business object definitions.	Step 6: Save Business Objects	"Saving content" on page 95

Selecting the ODA

When users choose the **File -> New Using ODA** Business Object Designer Express invokes Business Object Wizard to run the ODA. Step 1 of Business Object Wizard displays the Select Agent dialog box, which provides graphical access to all available Object Discovery Agents. From this dialog box, users select the ODA to run.

Business Object Wizard connects to this ODA with the following steps:

- Instantiates an ODA object, which is an object of the *ODA class*. The ODA class is the extension of the ODA base class, `ODKAgentBase2`. It defines the behavior of the ODA.
- Obtains a handle to the ODA object, which can be used to access this object when started.

Note: An ODA must already be started for Business Object Wizard to list it as an ODA available to run. For more information, see "Before using an ODA" on page 64.

For more information on how to create the ODA class, see "Extending the ODA base class" on page 100.

Obtaining ODA configuration properties

Step 2 of Business Object Wizard displays the Configure Agent dialog box, which shows the ODA's configuration properties. *Configuration properties* are those properties that the ODA needs to be able to begin running. The ODK API represents a configuration property as an agent-property (`AgentProperty`) object. In this step, the wizard displays the configuration properties, allows you to update them, and then writes the user-initialized properties into the ODA runtime memory.

Business Object Wizard	Object Discovery Agent
Begin Step 2	1 → getAgentProperties()
Display Configure Agent dialog	2 ← Send initialized configuration-property array
Write user-initialized configuration properties into ODA-runtime memory	3 → Retrieve with getAgentProperty() or getAllAgentProperties()
Initialize ODA	4 → init()
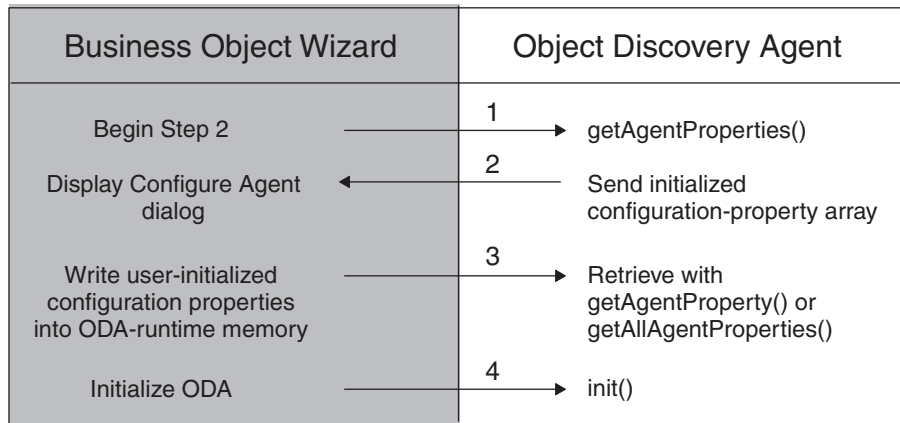
Figure 53. Configure Agent (Step 2) of Business Object Wizard

As Figure 53 shows, Business Object Wizard takes the following actions:

1. Obtains the configuration properties from the selected ODA and displays them in the Configure Agent dialog box.

   To obtain the configuration properties from the ODA, the wizard calls the `getAgentProperties()` method, which is defined in the ODA base class, `ODKAgentBase2`. This method is an abstract method that the ODA developer must implement as part of the ODA class. It returns the ODA's configuration properties to Business Object Wizard as an array of `AgentProperty` objects. These configuration properties can include the names, types, any valid values, descriptions, input restrictions, and any default values.

   In addition to the configuration properties that `getAgentProperties()` provides, Business Object Wizard always provides a set of standard configuration properties, which are common to all ODAs:

   - `MessageFile`
   - `TraceLevel`
   - `TraceFileName`

   For more information, see "Obtaining configuration properties" on page 102.

2. From the Configure Agent dialog box, accepts entered values or changes for the configuration properties. The wizard sends the user-initialized configuration properties to the ODA.

   Business Object Wizard saves these properties in the ODA runtime memory. Within the ODA, you can access these properties through an instance of the `ODKUtility` class, which provides the `getAgentProperty()` and `getAllAgentProperties()` methods for this purpose.

3. Initializes the ODA's metadata, which provides information about the ODA and its capabilities.

   After it calls `getAgentProperties()`, Business Object Wizard calls the `getMetaData()` method of the ODA base class, `ODKAgentBase2`. This method is an abstract method that the ODA developer must implement as part of the ODA class. It returns an initialized `AgentMetaData` object that contains the ODA metadata.

4. Initializes the ODA based on the user-initialized startup properties.

   To initialize the ODA, the wizard calls the `init()` method of the ODA base class, `ODKAgentBase2`. This method is an abstract method that the ODA developer must implement as part of the ODA class. It performs initialization tasks such as resource allocation and creating a connection to the data source.

This chapter provides the following information on how to implement the methods involved in the initialization of an ODA:

Initialization method	For more information
getAgentProperties()	"Obtaining configuration properties" on page 102
getMetaData()	"Initializing ODA metadata" on page 104
init()	"Initializing the ODA start" on page 105

Selecting and confirming source data

Step 3 of Business Object Wizard displays the Select Source dialog box, which displays the *source nodes* of the data source. The source nodes are arranged in the source-node hierarchy. Each source node is the name of an object that the ODA has "discovered" in the data source. It can either be expanded to display other child nodes or selected for generation into content. Users can expand this source-node hierarchy to choose objects in the data source for conversion to content. For information, see "Moving through the source-node hierarchy" on page 78..

In Step 3, the wizard takes the following actions:

1. Obtains the source-node hierarchy from the selected ODA and displays it top level in the Select Source dialog box.

   To obtain the source-node hierarchy, the wizard calls the getTreeNodes() method of the IGeneratesBoDefs interface. The ODA developer must implement this method as part of the ODA class's implementation of the IGeneratesBoDefs interface. It searches the data source to "discover" source nodes and returns these source nodes to Business Object Wizard as an array of TreeNode objects. When users expand a node for the first time, the wizard calls getTreeNodes() to display that particular level in the source-node hierarchy. Users can traverse this hierarchy to select the level of detail. For more information, see "Moving through the source-node hierarchy" on page 78.

2. From the Select Source dialog box, keeps track of the names of the source nodes in the hierarchy that you select for content generation. The wizard generates an array that contains the names of the selected source nodes.

Step 4 of Business Object Wizard displays the Confirm Source Nodes dialog box, which displays the selected source nodes. Users can either confirm the selections or go back to the Select Source dialog box to reselect source nodes. When the **Next** button is clicked, the wizard begins the content generation.

For information on how to implement the getTreeNodes() method, see "Generating source nodes" on page 111.

Generating content

You can write an ODA to generate one or both of the *content types* listed in Table 17. The content type determines the structure of the data that the ODA generates. For an ODA to support a particular content, it must implement the appropriate *content-generation interface* for the ODA. Table 17 lists the content types that an ODA can support as well as the associated content-generation interface the ODA must implement.

Table 17. Content types for an ODA

Content type	Description	Content-generation interface
Business object definitions	The ODA generates business object definitions to represent the objects in the data source.	IGeneratesBoDefs
Binary files	The ODA generates file objects to hold information about the generated content.	IGeneratesBinFiles

Note: With this release, an ODA *must* support the generation of business object definitions as its content. Therefore, it *must* implement the IGeneratesBoDefs interface. Additionally, the ODA can support the generation of files as its content by implementing the IGeneratesBinFiles interface.

After source nodes are selected and confirmed, Business Object Wizard enters Step 5 of the content generation. It displays the Generating Business Objects screen and passes the array of user-selected source nodes (from Step 4) to the ODA by calling the content-generation method for business object definitions, generateBoDefs(). This method generates the corresponding business object definitions for the selected source nodes. Because an ODA *must* support the generation of business object definitions in the on-request content protocol, Business Object Wizard *always* calls the generateBoDefs() method. Therefore, the ODA developer must implement this method as part of the ODA's implementation of the IGeneratesBoDefs interface.

Whether the ODA generates file content depends on whether it implements the IGeneratesBinFiles interface. If the ODA class implements this interface, the method that actually provides the generated content depends on the *content protocol* that the ODA uses for the file content type, as follows:

- If the ODA uses the *on-request* content protocol to generate content, Business Object Wizard initiates content generation as part of Step 5 by calling the content-generation method, generatesBinFiles(). It passes to this method the array of user-selected source nodes. Therefore, for the ODA to support file content, the ODA developer must implement this method as part of the ODA's implementation of the IGeneratesBinFiles interface.
- If the ODA uses the *callback* content protocol to generate content, the ODA (or some external process) initiates content generation by calling a user-defined method. The ODA developer must implement a mechanism to generate the files.

Therefore, whether Business Object Wizard calls the content-generation method for files, generateBinFiles(), depends on the following:

- Whether the ODA implements that IGeneratesBinFiles interface
- If it implements IGeneratesBinFiles, which content protocol the ODA uses to generate files

Note: For more information on content protocols, see "Choosing the ODA content protocol" on page 108.

Regardless of the content protocol uses, the generation of content involves the following steps:

1. Optionally, obtaining any additional information, such as verb values, as business-object properties.

2. Generating the requested content and saving it in the generated-content structure in ODA memory.

The following sections summarize these steps. For a more detailed overview of the content-generation process, Table 18 shows where to find more information for each of the supported content types.

Table 18. Content-generation process

Content type	For more information
Business object definitions Binary files	"Generating business object definitions" on page 118 "Generating files" on page 135

Obtaining business-object properties

Often the ODA needs additional information before it can generate the business object definitions. The ODA can request this additional information by defining *business-object properties*. The ODK API represents a business-object property as an agent-property (`AgentProperty`) object. To collect business-object properties, the ODA can have Business Object Wizard display the BO Properties dialog box. In this dialog box, the wizard displays the business-object properties, allows updates, and writes the user-initialized properties into the ODA runtime memory, as Figure 51 on page 82 shows.

To display the BO Properties dialog box, the content-generation method of the ODA calls the `getBOSpecificProps()` method (defined in the `ODKUtility` class).
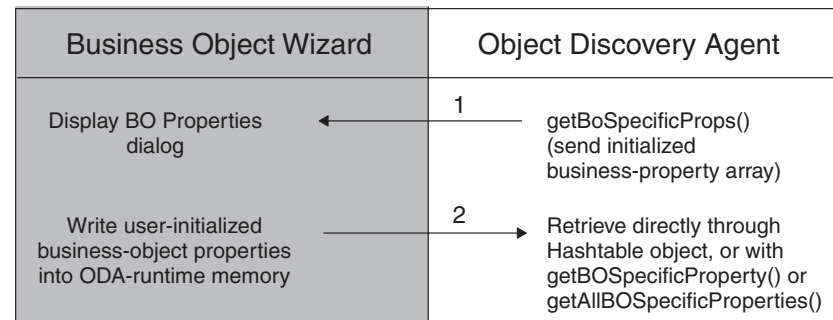
Figure 54. Obtaining business-object properties

As Figure 54 shows, the `getBOSpecificProps()` method takes the following steps:

1. Sends the business-object properties to Business Object Wizard, which displays them in the BO Properties dialog box.

   To send the business-object properties, the `getBOSpecificProps()` method sends as an argument the initialized array of agent-property (`AgentProperty`) objects, one object for each business-object property to display.

2. From the BO Properties dialog box, values can be added or changed. After the **Next** button is clicked, the wizard sends the user-initialized business-object properties back to the `getBOSpecificProps()` method in the ODA.

   You can access these business-object properties within the ODA through the Java `Hashtable` object that `getBOSpecificProps()` returns. Alternatively, you can

access these properties through an instance of the `ODKUtility` class, which provides the `getBOSpecificProperty()` and `getAllBOSpecificProperties()` methods.

The ODA can call `getBOSpecificProps()` repeatedly to obtain different sets of business-object properties. For more information on how to use the `getBOSpecificProps()` method, see "Requesting business-object properties" on page 119.

Providing generated content

The ODA provides its generated content to Business Object Wizard in two parts:

- The content metadata

   A content-metadata (`ContentMetaData`) object contains information about the ODA's generated content. Business Object Wizard uses this information to determine which content-retrieval method to use to retrieve the generated content.

- The content itself

   The ODA writes the generated content to a *generated-content structure*, somewhere that is accessible by the methods of the ODA class. For example, it could write the content to an array that is a member variable of the ODA class.

The method that provides the generated content depends on the content protocol that the ODA uses for a particular content type, as follows:

- If the ODA uses the on-request content protocol to generate content, it is the content-generation method that populates the generated-content structure and returns a content-metadata object to Business Object Wizard. Business Object Wizard invokes the content-generation method based on the content type, as follows:
  - For business object definitions, `generateBoDefs()` in the `IGeneratesBoDefs` interface
  - For files, `generateBinFiles()` in the `IGenerateBinFiles` interface
- If the ODA uses the callback content protocol to generate content, it is a user-defined method that populates the generated-content structure and sends a content-metadata object to Business Object Wizard.

Note: For more information on content protocols, see "Choosing the ODA content protocol" on page 108.

The following table shows where to find more information on how to provide generated content:

Content type	For more information
Business object definitions	"Providing generated business object definitions" on page 130
Binary files	"Providing generated files" on page 138

To retrieve the generated content, Business Object Wizard calls the appropriate content-retrieval method as Table 19 shows.

Table 19. Content-retrieval methods

Content type	Content-retrieval method	For more information
Business object definitions	`IGeneratesBoDefs.getBoDefs()`	"Providing access to generated business object definitions" on page 131
Binary files	`IGeneratesBinFiles.getBinFile()`	"Providing access to generated files" on page 139

The content-retrieval method accesses the generated-content structure within the ODA object and returns specified content in an array to Business Object Wizard. Business Object Wizard must have access to the generated content before it can process the request to save the content in Step 6. For more information, see "Saving content."

Saving content

Step 6 of Business Object Wizard displays the Save Business Objects dialog box, which provides options for saving the generated business object definitions. As Figure 42 on page 74 shows, Business Object Wizard provides the ability to save generated content to an ICL project or a file, or to open each business object definition in Business Object Designer Express. To save the generated business object definitions in the specified format, Business Object Wizard must access the generated content. It has retrieved this content in the previous step (Step 5), using the ODA's content-retrieval method listed in Table 19.

Overview of the ODA development process

This section provides the following information about the process to develop an ODA:
- "Tools for ODA development"
- "ODA development process" on page 98

Tools for ODA development

An ODA is one of the possible components of an IBM WebSphere Business Integration Adapter. An *adapter* includes run-time components to support communication between InterChange Server Express and applications or technologies. One of these run-time components is the ODA, which creates the business object definitions for the connector to use at run time. The *connector* is the run-time component that handles communication between an application (or technology) and InterChange Server Express. The adapter also includes an *adapter framework*, which includes components for the configuration, run time, and development of custom adapters in cases where a prebuilt adapter for a particular legacy or specialized application is not currently available as part of the WebSphere Business Integration Adapter product.

For development of an ODA, the adapter framework includes the development support listed in Table 20.

Table 20. Adapter framework support for the development of an ODA

Adapter component	Configuration Tool Express	API
Business object definition	Business Object Designer Express	Not applicable

Table 20. Adapter framework support for the development of an ODA (continued)

Adapter component	Configuration Tool Express	API
Object Discovery Agent (ODA)	Business Object Designer Express	Object Discovery Agent Development Kit (ODK)

Note: The adapter framework also provides support for the development of connectors.

In addition to the WebSphere Business Integration Adapter Framework, the Adapter Development Kit (ADK) is a toolkit that provides code samples of ODAs and connectors. For more information, see "Adapter Development Kit."

Adapter Development Kit

The Adapter Development Kit (ADK) provides files and samples to assist in the development of an adapter. It provides samples for many of the adapter components, including an Object Discovery Agent (ODA), a connector, and a data handler. The ADK provides these samples in the `DevelopmentKits` subdirectory of the product directory.

Note: The ADK is part of the WebSphere Business Integration Adapters product and requires a separate installation. Therefore, to have access to the development samples in the ADK, you must have access to the WebSphere Business Integration Adapters product and install the ADK. Please note that the ADK is available *only* for Windows systems.

Table 21 lists the samples that the ADK provides for the development of an ODA as well as the subdirectory of the `DevelopmentKits` directory in which they reside.

Table 21. ADK Samples for ODA Development

Adapter Development Kit component	Description	DevelopmentKits subdirectory
Object Discovery Agent Development Kit (ODK)	Provides ODA samples	`Odk`
Twineball adapter sample	Provides a sample adapter, which includes an ODA	`Twineball_sample`

As Table 21 shows, the Adapter Development Kit includes samples of Object Discovery Agents (ODAs). These samples reside in the following directory:

`DevelopmentKits\Odk`

For more information, see "Development support for ODAs" on page 97.

Note: As Table 21 shows, the ADK also provides support for the development of connectors, another adapter component.

Development support for business object definitions

Table 22 shows the tools that the WebSphere Business Integration Adapters and WebSphere InterChange Server Express products provide to assist in the development of business object definitions.

Table 22. Tools for development of business object definitions

Development tool	Description
Business Object Designer Express	Graphical tool that assists in the creation of business object definitions, either manually or through an ODA.

For a brief introduction to business object definitions, see "Business object definitions" on page 4.

Development support for ODAs

Table 23 shows the tools that the WebSphere Business Integration Adapters and WebSphere InterChange Server Express products provide to assist in the development of an ODA.

Table 23. Tools for development of ODAs

Development tool	Description
Business Object Designer Express Object Discovery Agent Development Kit (ODK)	Graphical tool that assists in the creation of business object definitions, either manually or through an ODA. Contains: • ODK API: a set of Java classes with which you can create a custom ODA. For an overview of these classes, see Chapter 7, "Overview of the ODK API," on page 165. • ODA runtime: a set of Java classes that the ODA runtime uses to handle communication between the ODA and Business Object Designer Express • ODA samples: installed as part of the Adapter Development Kit (ADK). For more information, see "Adapter Development Kit" on page 96.

As Table 23 shows, the ODK provides for the ODA developer both the ODK API (which is the library of methods to use in the ODA) and sample ODAs, which reside in the following product subdirectory:

```
DevelopmentKits\Odk\Samples
```

The ODK includes the following sample ODAs

Table 24. Sample ODAs

ODA sample	Description	Subdirectory of DevelopmentKits\Odk
Roman Army ODA	Converts the names of Roman generals and soldiers from an XML file to business object definitions and provides some binary files that describe the conversion. This ODA uses the ODK API, as described in this chapter.	For startup scripts: `Samples` For external files and.jar file: `RomanArmy` For Java source: `com\ibm\btools\ODK2\ RomanArmy`

Table 24. Sample ODAs (continued)

ODA sample	Description	Subdirectory of DevelopmentKits\Odk
JDBC ODA	Converts JDBC data (tables and schemas) to business object definitions. For this sample ODA to run, it must have access to a JDBC database. This sample is based on a previous version of the ODK API, which handles generation of business object definitions *only*, not generation of file content. **Note:** If you are developing a new ODA, use this sample *only* as an example of more complex business-object-definition creation. Use the Roman Army ODA sample as an example of how your new ODA should be structured.	For startup scripts: `Samples` For Java source: `com\crossworlds\JDBC`

For a brief introduction to ODAs, see "Using an Object Discovery Agent to create a business object definition" on page 64. For instructions on how to run the sample Roman Army ODA, see "Using the sample ODA" on page 66.

ODA development process

This section provides an overview of the ODA development process, which includes the following high-level steps:

1. Install and set up the WebSphere business integration system software and install the Java Development Kit (JDK).

2. Design and implement the ODA.

Setting up the development environment

Before you start the development process, the following must be true:

- The WebSphere business integration system software is installed on a machine that you can access.

  For an ODA to run, it must be able to access the ODA library, `CwODA.jar`. Therefore, this ODA library must be installed. For more information, see your product installation information.

  > **InterChange Server Express**
  >
  > If your business integration system uses InterChange Server Express, the `CwODA.jar` file is installed as part of the software. Refer to the for product installation information, which includes how to install and start up the system.

- The Java Development Kit (JDK) or a JDK-compliant development product is installed on the development machine.

  For the required version of the JDK and how to install it, refer to your product installation information. Make sure to update the PATH environment variable to include the installed Java directory. Since InterChange Server Express is your integration broker, restart InterChange Server Express after you have updated the path.

- The development environment must be able to access the directory that contains the ODA library file, `CwODA.jar`:

```
ProductDir\lib
```
To compile the ODA, the compiler *must* be able to access this directory ODA. For information on how to compile an ODA, see "Compiling the ODA" on page 159.

Note: To create an ODA and test its generated content, you do *not* need to have InterChange Server Express or a connector running. However, at some point, the connector must be running to test that the ODA's generated content correctly describes the connector's business objects. To test the entire IBM WebSphere business integration system, InterChange Server Express and the connector must be able to communicate.

Stages of ODA development

To develop an ODA, you must take the steps listed in Table 25.

Table 25. Steps in the development of an ODA

	ODA development step	For more information
1.	Extend the ODA base class, `ODKAgentBase2`, to create your ODA class.	"Extending the ODA base class" on page 100
2.	Implement the methods of the ODA class, which provide the means of starting the ODA.	"Starting the ODA" on page 101

3.	Design and implement the ODA content:
	
	• Which content types the ODA supports:
	
	  – Business object definitions: implement the `IGeneratesBoDefs` interface (required)
	
	  – Binary files: implement the `IGeneratesBinFiles` interface (optional)
	
	• Which content protocols the ODA uses:
	
	  – on request (required for business object definitions)
	
	  – callback
		"Determining the ODA generated content" on page 107

4.	Implement error and message handling for all ODA methods. Implement trace messages at the appropriate trace levels.	"Handling exceptions" on page 157 and "Handling trace and error messages" on page 150
5.	Create any classes needed to handle data-source interactions, such as: • Connection management • Content analysis and definition	IBM recommends that you modularize the Object Discovery Agent into component classes that handle its separate significant processes. Details depend on your data source.
6.	Build the ODA.	"Compiling the ODA" on page 159
7.	Create a startup script for the new ODA.	"Starting up a new ODA" on page 160
8.	Test and debug the ODA, recoding as necessary.	

Writing ODA code is only one part of the overall task for developing business objects. Before beginning to write the Object Discovery Agent code, you should clearly understand business object design issues, the application whose entities the business objects will represent, and the connector and data handler that will process the business objects. You should also be familiar with the steps users follow in Business Object Designer Express to create a business object definition using an Object Discovery Agent.

Note: For information on business object design, refer to Chapter 2, "Business object design," on page 17. For information on using an Object Discovery Agent in Business Object Designer Express, see "Using an Object Discovery Agent to create a business object definition" on page 64.

Extending the ODA base class

To create an ODA, you extend the ODA base class, `ODKAgentBase2`, to create your own *ODA class*. The `ODKAgentBase2` class includes methods for initialization, setup, and termination of the ODA. To implement your own ODA, you must extend this ODA base class to create your ODA class.

To derive an ODA class, follow these steps:

1. Create an ODA class that extends the `ODKAgentBase2` class. A suggested name for this ODA class is:

   ODAname`.java`

   where *ODAname* uniquely identifies the ODA and has the format of the ODA's source data with the `ODA` extension (*srcDataName*`ODA`). For information about source-data names, see "Naming the ODA" on page 159. For example, to create an ODA for HTML objects, you create an ODA-class file called `HTMLODA.java`.

2. In the ODA-class file, define a package name to contain your ODA. By convention, an ODA package name has the following format:

   `com.ibm.oda.`*srcDataName*`.`*ODAname*

   In the format above, *ODAname* is the same as defined in step 1 and *srcDataName* is the same as in step 1 *except* that it is in lowercase letters. For example, the package name of an ODA for HTML objects could be defined in the ODA class as follows:

   `package com.ibm.oda.html.HTMLODA;`

3. Ensure that the ODA-class file imports the classes of the `com.crossworlds.ODK` package:

   `import com.crossworlds.ODK.*;`

   To access the methods of the ODK API, the ODA class must import the ODK package, which is contained in the `CwODK.jar` file in the `lib` subdirectory of the product directory. If you create several files to hold the ODA-class code, you must import the classes of the ODK package into *every* source file.

4. Define the ODA class and include in the definition any content-generation interfaces that the ODA uses. An ODA must implement the `IGeneratesBoDefs` content-generation interface to generate business-object-definition content. Optionally, it can also implement the `IGeneratesBinFiles` content-generation interface to generate binary-file content.

   For example, suppose that the ODA for HTML implements *only* the required the `IGeneratesBoDefs` interface. Its definition would be as follows:

   `public class HTMLODA extends ODKAgentBase2 implements IGeneratesBoDefs {`

   For more information about content-generation interfaces, see "Determining the ODA generated content" on page 107.

5. Implement the abstract methods of the `ODKAgentBase2` class for your ODA class. Table 26 provides an overview of these methods, listing them in the order in which Business Object Wizard calls them. For more information on how to implement these abstract methods, see Table 26.

Table 26. Extending abstract methods of the ODKAgentBase2 class

Abstract ODKAgentBase2 method	Description	For more information
getAgentProperties()	This method performs the following tasks: • Define the configuration properties needed to initialize the ODA, including information the ODA needs to connect to the data source. • Send the configuration properties in an array to Business Object Wizard.	"Obtaining configuration properties" on page 102
getMetaData()	Instantiate the AgentMetaData object that contains the ODA's metadata, including its ability to generate content.	"Initializing ODA metadata" on page 104
init()	Initialize the ODA, including allocation of resources and connection to the data source.	"Initializing the ODA start" on page 105
terminate()	Perform cleanup, including disconnecting from the data source and releasing any resources that the ODA uses.	"Shutting down the ODA" on page 149

6. Implement the methods of the appropriate content-generation interface (or interfaces) in your ODA class. Table 26 lists the methods of the content-generation interfaces and indicates where to find more information on how to create these methods.

Table 27. Defining methods of the content-generation interface

Content-generation interface	Description	For more information
IGeneratesBoDefs	getContentProtocol()	"Determining the ODA generated content" on page 107
	getTreeNodes() generateBoDefs() getBoDefs()	"Generating business object definitions as content" on page 110
IGeneratesBinFiles	getContentProtocol()	"Determining the ODA generated content" on page 107
	generateBinFiles() getBinFile()	"Generating binary files as content" on page 133

Starting the ODA

When the ODA is started, the ODA runtime instantiates the associated ODA class (an extension of ODKAgentBase2) and then calls the class methods in Table 28.

Table 28. Starting the ODA

Initialization task	ODKAgentBase2 method	For more information
1. Obtain the configuration properties, including those that describe the data source to open.	getAgentProperties()	"Obtaining configuration properties" on page 102
2. Initialize the ODA metadata so that Business Object Wizard can obtain information about the ODA (especially its supported content).	getMetaData()	"Initializing ODA metadata" on page 104
3. Initialize the ODA to perform any necessary startup steps, such as opening a connection to the data source.	init()	"Initializing the ODA start" on page 105

The following sections describe each of the steps in Table 28.

Obtaining configuration properties

To begin ODA initialization, Business Object Wizard calls the `getAgentProperties()` method of the ODA class. The `getAgentProperties()` method is part of the low-level ODA base class, `ODKAgentBase`. It is inherited by the ODA base class, `ODKAgentBase2`, then inherited in turn by your ODA class.

Important: As part of the implementation of your ODA class, you *must* implement an `getAgentProperties()` method.

The `getAgentProperties()` method performs the following tasks:
- "Obtaining the handle to the ODKUtility object"
- "Initializing the configuration-property array"

Obtaining the handle to the ODKUtility object

Because `getAgentProperties()` is the *first* ODA method that Business Object Wizard calls, it is a good place to instantiate the `ODKUtility` object, which provides the ODA code with access to the following:
- Objects in the memory of the ODA runtime, such as configuration properties and business-object properties
- Utility methods that provide tracing and display user-response dialog boxes

To obtain access to an `ODKUtility` object, use the `getODKUtility()` method. This method, defined in the `ODKUtility` class, returns a handle to the `ODKUtility` object.

```
odkUtil = ODKUtility.getODKUtility()
```

If you declare the handle to the `ODKUtility` object as global to the entire ODA class, *all* methods within this class can access the utility methods.

Note: Instead of instantiating the `ODKUtility` object in its `getAgentProperties()` method, the sample Roman Army ODA provides a member variable named `m_utility` in its ODA class and initializes it as follows:

```
final ODKUtility m_utility = ODKUtility.getODKUtility();
```

Initializing the configuration-property array

As "Obtaining ODA configuration properties" on page 89 describes, Business Object Wizard uses the configuration-property array that `getAgentProperties()` returns to initialize the Configure Agent dialog box (Step 2). This dialog box displays all ODA configuration properties and allows users to enter or change their values. The configuration-property array is an array of `AgentProperty` objects. The `AgentProperty` class provides support for the configuration property to have the following features:
- A default value
- Hold only one value or more than one value
- A list of valid values for the user to choose from
- Conditions that restrict the value the user can enter

Note: For more information, see "Working with agent properties" on page 140.

The purpose of `getAgentProperties()` is to send to Business Object Wizard an array of `AgentProperty` objects that describe the ODA configuration properties. To initialize the configuration-property array in `getAgentProperties()`, take the following steps:

1. Instantiate an `AgentProperty` object for a configuration property, initializing it with the appropriate property information.

   The implementation of the `getAgentProperties()` method must instantiate agent-property objects for each configuration property that the Business Object Wizard is to display to users. When you instantiate the agent-property object, you initialize some or all of its member variables (shown in Table 49 on page 141).

2. Store the initialized `AgentProperty` object in the configuration-property array.

3. Return the initialized configuration-property array from the `getAgentProperties()` method.

Figure 55 shows the implementation of the `getAgentProperties()` method (defined in the `ArmyAgent2` class of the sample Roman Army ODA).

```
public AgentProperties[] getAgentProperties()
   throws com.crossworlds.ODK.ODKException
{
   AgentProperty general = new AgentProperty("Army general",
      AgentProperty.TYPE_STRING, true, false, false,
      "A general is a soldier at least 45 years old", true,
      ODKConstant.SINGLE_CARD, m_generals.toArray(), null);

   AgentProperty recAdop = new AgentProperty("Allow adoption",
      AgentProperty.TYPE_BOOLEAN, true, false, false,
      "Select \"yes\" if adopted children can be business objects", true,
      ODKConstant.SINGLE_CARD, new Object[]{"true", "false"}, null);

   AgentProperty minAge = new AgentProperty("Minimum age for drafting",
      AgentProperty.TYPE_INTEGER, true, false, false,
      "Drafted soldiers will be generable nodes", false,
      ODKConstant.SINGLE_CARD, null, new Object[] {"15"});

   AgentProperty maxAge = new AgentProperty("Maximum age for drafting",
      AgentProperty.TYPE_INTEGER, true, false, false,
      "Drafted soldiers will be generable nodes", false,
      ODKConstant.SINGLE_CARD, null, new Object[] {"55"});

   AgentProperty minAdo = new AgentProperty("Minimum age for adopting",
      AgentProperty.TYPE_INTEGER, true, false, true,
      "Drafted soldiers will be generable nodes", false,
      ODKConstant.SINGLE_CARD, null, new Object[] {"" + m_minAdoptionAge});

   AgentProperty[] props = new AgentProperty[]
      {general, minAge, maxAge, recAdop, minAdo});

   return props;
}
```

Figure 55. Initializing the configuration-property array

Figure 55 initializes the five ODA configuration properties for the sample Roman Army ODA. The actual properties you define depend on the specific data source your ODA is accessing.

After values are specified for the configuration properties, Business Object Wizard saves these properties in the memory of the ODA runtime. The ODA can access these properties through methods such as the `getAgentProperty()` method in the `ODKUtility` class. For more information, see "Retrieving ODA configuration properties" on page 105.

Initializing ODA metadata

After Business Object Wizard calls the ODA's `getAgentProperties()` method, it calls the `getMetaData()` method to initialize the ODA metadata. The `getMetaData()` method is defined in the ODA base class, `ODKAgentBase2`, then inherited by your ODA class. It returns an `AgentMetaData` object, which contains the ODA's metadata, including the generated content it supports.

Important: The `getMetaData()` method is an abstract method. As part of the implementation of your ODA class, you *must* implement a `getMetaData()` method.

The `AgentMetaData` object provides the information in Table 29 to the ODA runtime when it needs to obtain metadata for the ODA.

Table 29. Contents of an AgentMetaData object

Member variable	Description
`agentVersion`	The version of the ODA
`searchableNodes`, `searchPatternDesc`	Information to specify the ODA search pattern, which the user can specify to reduce the number of tree nodes from the data source that are displayed
`supportedContent`	A description of the generated content that the ODA can support

To initialize the ODA metadata, you implement the `getMetaData()` method, which involves the following steps:

- Create an instance of the `AgentMetaData` class, passing in a reference to the ODA itself and an optional ODA version.

  Use either of the forms of the `AgentMetaData()` constructor. Both forms require that you pass in a `this` reference to the ODA object (an instance of your ODA class). The constructor queries the ODA object to obtain information about the content-generation interface (or interfaces) that the ODA implements. It then uses this information to initialize the `supportedContent` member variable with the content protocols that the ODA supports for each of its supported content types. For more information on the ODA's supported content, see "Determining the ODA generated content" on page 107.

  Optionally, you can also provide the ODA version as an argument to the constructor to initialize the `agentVersion` member variable.

- Initialize other member variables as appropriate for your ODA.

  For your ODA to support the search-pattern feature, you must explicitly initialize the `searchableNodes` and `searchPatternDesc` member variables after the `AgentMetaData` object is instantiated. For more information, see "Implementing the search-pattern feature" on page 114.

- Return the initialized `AgentMetaData` object from the `getMetaData()` method.

Figure 56 shows the implementation of the `getMetaData()` method (defined in the `ArmyAgent2` class from the sample Roman Army ODA).

```
public AgentMetaData getMetaData(){
   odkUtil.trace(TRACELEVEL1, XRD_TRACE, "Entering getMetaData()...");
   AgentMetaData amdObj = new AgentMetaData(this, "Sample ODA v1.0.0");

   //Initialize search-pattern feature for tree nodes
   amd.searchableNodes = true;
   amd.searchPatternDesc = "Enter the first letter to search by. For example, " +
      "\"A\", \"k\", \"Z\". Only names that start with this letter will be " +
      "returned."

   return amd;
}
```

Figure 56. Initializing ODA metadata

Because the `getMetaData()` method in Figure 56 is inherited by the `ArmyAgent3` class (which implements the `IGeneratesBoDefs` interface), the call to the `AgentMetaData()` constructor in this code fragment initializes the content type and its associated content protocol for the ODA. After `getMetaData()` starts in `ArmyAgent3`, the ODA's content type is initialized to `ContentType.BusinessObject` and its content protocol to "on request". For more information, see "Determining the ODA generated content" on page 107.

This `getMetaData()` method also provides support for the search-pattern feature by initializing the `searchableNodes` and `searchPatternDesc` member variables. The `searchPatternDesc` variable contains the text that displays in the Enter the Search Pattern dialog box (see Figure 46 on page 80).

Initializing the ODA start

After Business Object Wizard calls the ODA's `getMetaData()` method, it calls the `init()` method to begin initialization of the ODAstart. The `init()` method is part of the low-level ODA base class, `ODKAgentBase`. It is inherited by the ODA base class, `ODKAgentBase2`, then inherited in turn by your ODA class. This method performs initialization steps for the ODA.

Important: The `init()` method is an abstract method. As part of the implementation of your ODA class, you *must* implement an `init()` method.

The main tasks of the `init()` method include:
- "Retrieving ODA configuration properties"
- "Establishing a connection" on page 106
- "Checking the ODA version" on page 107

Retrieving ODA configuration properties

The `init()` method can retrieve any of the user-initialized configuration properties it needs to complete the initialization of the ODA. The ODA initializes its configuration properties in its `getAgentProperties()` method. Users can update these properties as needed in the Configure Agent dialog box of Business Object Wizard. After configuration properties are updated, Business Object Wizard writes them to the memory of the ODA runtime.

The ODK API provides the methods in Table 30 for retrieving the value of an ODA configuration property from the ODA runtime memory.

Table 30. Methods to retrieve the value of an ODA configuration property

ODK library method	Description
getAgentProperty()	Retrieves the value of a specified ODA configuration property
getAllAgentProperties()	Retrieves the values of *all* ODA configuration properties as a Java Hashtable object

All methods in Table 30 are defined in the ODKUtility class. Therefore, you must obtain a handle to the singleton object of this class *before* you can access any configuration properties. For more information, see "Obtaining the handle to the ODKUtility object" on page 102.

Figure 57 shows the implementation of the init() method (defined in the ArmyAgent3 class from the sample Roman Army ODA).

```
public void init() throws com.crossworlds.ODK.ODKException
{
   Hashtable h = m_utility.getAllAgentProperties();
// Obtain values of ODA configuration properties
   AgentProperty property = (AgentProperty) h.get("Army general");
   m_general = property.allValues[0].toString();

   property = (AgentProperty) h.get("Minimum age for drafting");
   m_minAge = Integer.parseInt(property.allValues[0].toString());

   property = (AgentProperty) h.get("Maximum age for drafting");
   m_maxAge = Integer.parseInt(property.allValues[0].toString());

   property = (AgentProperty) h.get("Allow adoption");
   m_allowAdoption = new Boolean(
      property.allValues[0].toString()).booleanValue();
// Clear the generated-content structure
   m_generatedBOs.clear();
}
```

Figure 57. Initializing the ODA

In Figure 57, the init() method uses the following to obtain configuration-property values:
- The getAllAgentProperties() method, defined in the ODKUtility class, to retrieve all configuration properties into a Java Hashtable object.
- The get() method, defined in the Java Hashtable class, to retrieve an element from the hashtable by its name.
- The allValues member variable, defined in the AgentProperty class, to get the value that the user has specified for each configuration property.

The configuration properties that this init() method obtains are all single-cardinality properties. Therefore, the allValues member variable contains only one value. For an example of using multiple-cardinality properties, see "Creating the business-property array" on page 120. This init() method also initializes the ODA's generated-content structure, a vector called m_generatedBOs. This vector will hold the generated business object definitions.

Establishing a connection
The main task of the init() initialization method is usually to establish a connection to the data source. The ODA searches the data source to "discover"

objects for potential conversion to business object definitions. To establish the connection, the `init()` method can perform the following tasks:

- Obtain any ODA configuration properties that provide connection information and use them to connect to the data source. If a required configuration property is empty, your `init()` method can provide a default value or it can throw the `ODKInvalidPropException` exception.

  You can use the `getAgentProperty()` method to obtain the value of an ODA configuration property. For more information, see "Retrieving ODA configuration properties" on page 105.

- Obtain any required connections or files. For example, the `init()` method usually establishes a connection with the data source. If the ODA *cannot* open a connection, the `init()` method must throw an `ODKException` exception (or one of its subclasses) to indicate the cause of the failure.

The `init()` method runs successfully if the ODA succeeds in opening a connection and the ODA is ready to begin processing data in the data source. If the ODA *cannot* open a connection, the `init()` method should throw an `ODKException` exception to indicate the cause of the failure.

Checking the ODA version

The `getVersion()` method returns the version of the ODA runtime. This method is part of the low-level ODA base class, `ODKAgentBase`. It is inherited by the ODA base class, `ODKAgentBase2`, then inherited by your ODA class. It is called in both of the following contexts:

- The `init()` method should call `getVersion()` to check the ODA runtime version.
- The ODA runtime calls the `getVersion()` method when it needs to get its version.

Note: The `getVersion()` method returns the version of the ODA runtime, *not* the version of the ODA (which is stored as part of the ODA's metadata).

Determining the ODA generated content

This section provides the following information on the issues you need to consider when determining the content that your ODA can generate:

- "Choosing the ODA content type"
- "Choosing the ODA content protocol" on page 108

Choosing the ODA content type

The ODK API identifies the valid content types that an ODA can support with the `ContentType` class. This class contains static member variables for each of the supported content types, as Table 31 shows.

Table 31. How content types are represented

Content type	ContentType member variable
Business object definitions	`BusinessObject`
Binary files	`BinaryFile`

The `ContentType` class simulates an enumerated list of the supported ODA content types. For example, a content-type object that represents business object definitions would use only the `BusinessObject` member variable, as follows:

`ContentType.BusinessObject`

To provide support for generation of a particular content type, an ODA must implement the appropriate content-generation interface, as listed in Table 17 on page 92. Every ODA must support generation of business object definitions. It can optionally also support generation of binary files as its content. The content-generation interfaces contain the kinds of methods listed in Table 32. As part of the implementation of the content-generation interface, you must implement these methods.

Table 32. Methods in a content-generation interface

Method	Method purpose	IGeneratesBoDefs	IGeneratesBinFiles
Source-node-generation method	Business Object Wizard calls this method to obtain the source-node hierarchy that it displays to the user (Step 3: Select Source).	getTreeNodes()	None
Content-generation method	Business Object Wizard calls this method to initiate generation of the specified content for the source data (Step 5: Generating Business Objects).	generateBoDefs()	generateBinFiles()
Content-retrieval method	Business Object Wizard calls this method to retrieve the generated content from ODA memory (Step 5: Generating Business Objects).	getBoDefs()	getBinFile()

To determine which content-generation interface's method to call, Business Object Wizard checks the ODA's metadata. One of the components of this metadata is the supportedContent member variable, which is initialized by the AgentMetaData() constructor, called within the ODA's getMetaData() method. For more information, see "Initializing ODA metadata" on page 104.

Table 33 shows the information that this chapter provides on how to implement methods in a content-generation interface.

Table 33. How to develop a content-generation interface

Content-generation interface	For more information
IGeneratesBoDefs	"Generating business object definitions as content" on page 110
IGeneratesBinFiles	"Generating binary files as content" on page 133

Choosing the ODA content protocol

An ODA can generate a particular content type using either of the *content protocols* listed in Table 34. The content protocol determines how the ODA interacts with Business Object Wizard to generate the supported content; that is, it determines whether Business Object Wizard can explicitly initiate content generation from the ODA.

Table 34. Content protocols for an ODA

Content protocol	Description	Content-protocol constant
On request	Business Object Wizard explicitly requests the ODA to generate content by calling the content-generation method. Once this method completes, on-request content is ready. Business Object Wizard can retrieve this content at its convenience with a call to the content-retrieval method.	`CONTENT_PROTOCOL_ONREQUEST`
Callback	The ODA generates the content in some fashion and notifies Business Object Wizard when its content is ready. Once notified, Business Object Wizard retrieves this content with a call to the content-retrieval method.	`CONTENT_PROTOCOL_CALLBACK`

Note: An ODA must support the generation of business-object-definition content with the on-request content protocol. Additionally, the ODA can support the generation of file content in either content protocol.

To support content protocols, your ODA must take the following steps:
- "Indicating the implemented content protocols"
- "Implementing the content-generation method" on page 110

Indicating the implemented content protocols

Both the `IGeneratesBoDefs` and `IGeneratesBinFiles` interfaces are extensions of the `IGeneratesContent` interface. Therefore, they both inherit the single method that `IGeneratesContent` defines, `getContentProtocol()`. As part of the implementation of the ODA's content-generation interface, you must implement the `getContentProtocol()` method to indicate which of the content protocols your ODA will use for its supported content types.

Note: An ODA can support *one* content protocol for a given content type.

The `getContentProtocol()` method accepts as an argument a `ContentType` object, which identifies a content type that the ODA supports. The `getContentProtocol()` method returns the content protocol that the ODA supports for this specified content type. It returns the supported content protocol as one of the content-protocol constants (shown in Table 34). These constants are defined in the `ODKConstant` interface.

Note: In this release, an ODA must generate business object definitions on request. Therefore, it must implement the `getContentProtocol()` method to return the `CONTENT_PROTOCOL_ONREQUEST` constant for a content type of `ContentType.BusinessObject`. Additionally, the ODA can support the generation of files in either protocol and return the appropriate content-protocol constant for a content type of `ContentType.BinaryFile`.

Figure 58 shows an implementation of `getContentProtocol()` that indicates the ODA supports the callback protocol for the generation of files and the on-request protocol for the generation of business object definitions.

```
public long getContentProtocol(ContentType contentType)
{
    if (contentType == ContentType.BinaryFile)
      return ODKConstant.CONTENT_PROTOCOL_CALLBACK;
    else
      return ODKConstant.CONTENT_PROTOCOL_ONREQUEST;
}
```

Figure 58. Indicating supported content protocols

Implementing the content-generation method

The implementation of the content-generation method depends on the content
protocol that the content type supports, as Table 35 shows.

Table 35. Content protocols and the content-generation method

Content protocol	How to call the content-generation method	Implementation of content-generation method
On request	Business Object Wizard explicitly calls the content-generation method to initiate content generation (business object definitions or files).	Method must generate content for the source nodes passed in its argument and return the appropriate content metadata to Business Object Wizard.
Callback	Business Object Wizard *never* explicitly calls the content-generation method because content generation (files only) is initiated by the ODA for this content protocol.	Method should throw an exception because it should *never* be called directly. Actual generation of content is performed external to the content-generation method, in a different method, class, or even process.

Table 36 shows the information that this chapter provides on how to implement the
content-generation methods.

Table 36. How to develop a content-generation method

Content-generation method	For more information
IGeneratesBoDefs.generateBODefs()	"Defining the generateBoDefs() method" on page 119
IGeneratesBinFiles.generateBinFiles()	"Defining the generateBinFiles() method" on page 136

Generating business object definitions as content

As discussed in "Business object definitions" on page 4, a *business object definition*
represents a template for data that can be treated as a collective unit. The purpose
of an ODA is to generate business object definitions for objects in a data source.
For an ODA to generate business-object-definition content, its ODA class must
implement the IGeneratesBoDefs interface.

Note: Because an ODA *must* support generation of business object definitions, its
ODA class must implement the IGeneratesBoDefs interface.

Table 37 lists the methods that the ODA class must define to implement the IGeneratesBoDefs interface.

Table 37. Methods in the IGeneratesBoDefs interface

Method	IGeneratesBoDefs method	Description
Source-node-generation method	getTreeNodes()	Iteratively performs the following: • Discover source nodes for objects within the data source. • Construct an array of tree nodes that represents the source-node hierarchy. • Return an array of tree nodes to Business Object Wizard, which displays them to users in the Select Source dialog box.
Content-generation method	generateBoDefs()	Generates the business object definitions for the user-selected source data, writing them to ODA memory
Content-retrieval method	getBoDefs()	Retrieves either a specified business object definition or all business object definitions from ODA memory

Note: In addition to the methods in Table 37, IGeneratesBoDefs also includes the getContentProtocol() method to specify the content protocol that the ODA supports for business-object-definition generation. For more information, see "Choosing the ODA content protocol" on page 108.

With the IGeneratesBoDefs interface implemented, Business Object Wizard invokes the methods shown in Table 38 to obtain source nodes, as well as generate and retrieve content.

Table 38. Business Object Wizard and IGeneratesBoDefs methods

Step in Business Object Wizard	IGeneratesBoDefs method	For more information
Step 3: Select Source	getTreeNodes()	"Generating source nodes"
Step 5: Generating Business Objects	generateBoDefs()	"Generating business object definitions" on page 118
Step 5: Generating Business Objects	getBoDefs()	"Providing access to generated business object definitions" on page 131

The following sections discuss the implementation of each of the methods in Table 38.

Generating source nodes

Business Object Wizard calls the getTreeNodes() method to discover the source nodes in the ODA's data source and create the source-node hierarchy, which Business Object Wizard displays in its Select Source dialog box (Step 3). The getTreeNodes() method is part of the IGeneratesBoDefs interface, which the ODA class must implement to support generation of business object definitions.

Important: As part of the implementation of the IGeneratesBoDefs interface, you *must* implement a getTreeNodes() method for your ODA.

As "Selecting and confirming source data" on page 91 describes, Business Object Wizard uses the tree-node array that getTreeNodes() returns to initialize the Select Source dialog box. This dialog box displays the source-node hierarchy, which allows users to move through the source nodes obtained from the data source and to select those for which the ODA generates business object definitions. Each time a source node is expanded, Business Object Wizard calls the getTreeNodes() method, which returns a tree-node array with the contents of the expanded source node.

For example, the getTreeNodes() method in the sample Roman Army ODA initializes the Select Source dialog box with the top-level army general, which it has obtained from the sample's data source, the RomanArmy.xml file. When a particular node is expanded, getTreeNodes() obtains the sons of that node's army general from the XML file and puts them into a tree-node array. Business Object Wizard uses this tree-node array to display the expanded source node.

Therefore, the purpose of getTreeNodes() is to discover source nodes in the data source, then construct and return an array of tree nodes. To do this, getTreeNodes() performs the following tasks:
- "Determining the parent-node path"
- "Implementing the search-pattern feature" on page 114
- "Querying the data source" on page 115
- "Constructing the tree nodes" on page 115

Determining the parent-node path

When Business Object Wizard calls the getTreeNodes() method, it passes to this method the value of the *parent-node path*. This path identifies the user-selected node that getTreeNodes() will expand. It is a String that contains the fully qualified path of the node, from the top-level parent down to the user-selected node. Node names within this path are separated with a colon (:).

For example, Figure 59 shows a Select Source dialog box that displays a view of the source-node hierarchy for the sample Roman Army ODA.
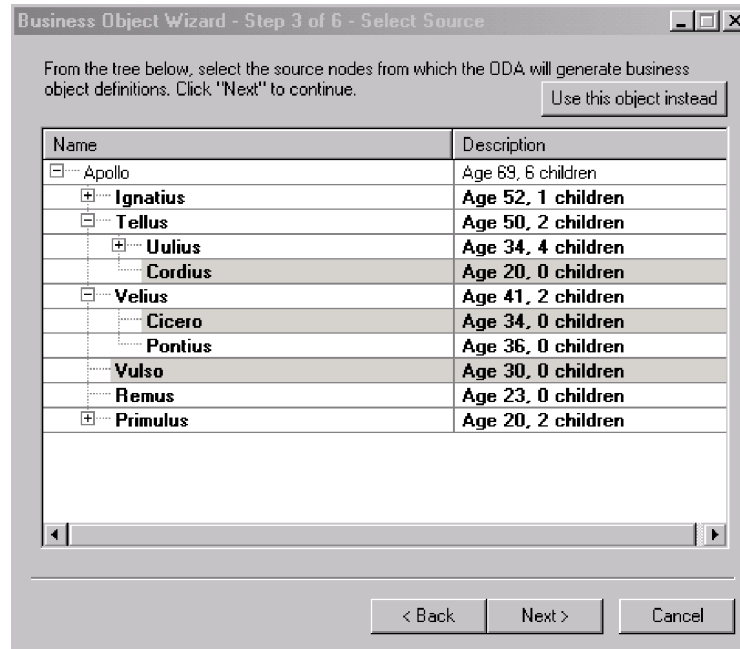
Figure 59. Sample source-node hierarchy

In Figure 59, the parent-node path for the Uulius source node is:

```
Apollo:Tellus:Uulius
```

Users can specify a parent node to expand by one of the following ways:

*   Clicking the + symbol to the left of parent node's name.

    Business Object Wizard constructs the parent-node path for the selected source node and passes this path to `getTreeNodes()`.

*   Selecting **Use this object instead** at the top of the Select Source dialog box and specifying an explicit parent-node path in the Object Path dialog box.

    You must specify the parent-node path with the same syntax that `getTreeNodes()` expects for its parent-node path. For more information, see "Specifying an object path" on page 80.

The `getTreeNode()` method uses the parent-node path to determine the level of the source-node hierarchy to return in its tree-node array. This tree-node array will contain all child nodes of the node that the parent-node path identifies. To tell `getTreeNodes()` to return the top-level of the source-node hierarchy, Business Object Wizard passes in an "empty" parent node path. Therefore, the `getTreeNodes()` method should check for an empty node path as its first step, as the following code fragment shows:

```
if (parentNodePath = null || parentNodePath.length() == 0)
    //return the top-level of the source-node hierarchy
```

If the parent-node path is *not* empty, `getTreeNodes()` should build the tree nodes for the children of the specified parent-node path, returning the appropriate array of `TreeNode` objects to Business Object Wizard.

Figure 60 shows the implementation of the `getTreeNodes()` method (defined in the `ArmyAgent3` class of the sample Roman Army ODA).

```
public TreeNode[] getTreeNodes(String parentNodePath, String searchPattern)
   throws ODKException
{
   if (parentNodePath == null || parentNodePath.length() == 0)
      return getNodes(m_army, searchPattern);
   return getNodes(findSon(parentNodePath, searchPattern));
}
```

Figure 60. Generating the tree-node array

Figure 60 shows an important concept in the implementation of the getTreeNodes()
method. This method is often modularized, putting the actual search of the data
source into a separate method or even into a separate class. This getTreeNodes()
method calls the getNodes() method to actually generate the tree-node array for
the selected data-source data. If the parent-node path is empty, getTreeNodes()
sends to getNodes() the entire contents of the XML file (in the m_army variable).
Otherwise, getTreeNodes() sends to getNodes() the results of the findSon()
method, which performs the actual query of the data source.

Implementing the search-pattern feature

A *search pattern* allows you to specify criteria that child nodes must meet to be
displayed when the parent node is expanded. You initiate the search-pattern
feature by right-clicking and then clicking **Search for items**. The Enter a Search
Pattern dialog box opens, where you can specify the search criteria.

Note: For more information on how to use the search-pattern feature, see
"Specifying a search pattern" on page 79.

When Business Object Wizard receives a search pattern, it calls getTreeNode()
again to retrieve a new tree-node array from the data source. Business Object
Wizard passes the search pattern as an argument to getTreeNodes(). The search
pattern contains wildcards and other symbols that the underlying data source
recognizes. For example, if the data source is a database, valid search criteria could
include SQL search symbols such as a percent (%) or question mark (?).

The getTreeNodes() method searches the data source for child nodes that match
the search pattern and puts the resulting child nodes in the tree-node array that it
returns to Business Object Wizard. In this way, you can dynamically specify new
conditions for source nodes to meet.

Note: Unlike a filter, a search pattern causes Business Object Wizard to call the
getTreeNodes() method again. A filter just causes Business Object Wizard to
search the child nodes of the parent node that is currently displaying; that
is, Business Object Wizard looks for child nodes already in the current
source-node hierarchy. It does *not* call getTreeNodes() to search the data
source for new matching child nodes.

To implement the search-pattern feature for your ODA, take the following steps:

• Enable the search-pattern feature in the ODA's metadata (AgentMetaData) by
  setting the searchableNodes member variable to true.

  You should also initialize the searchPatternDesc member variable to a string
  that describes to the user the valid search criteria. Business Object Wizard
  displays this string as the text for the Enter the Search Pattern dialog box. You
  initialize the ODA's metadata in the getMetaData() method. For more
  information, see "Initializing ODA metadata" on page 104.

• Implement your getTreeNodes() method to use the value of the *searchPattern*
  argument in the queries it makes of the data source.

For example, if the data source is a database, you can include the search pattern in SQL statements that query the database tables.

In the sample Roman Army ODA, the search pattern allows the user to enter one letter as search criteria. The `getTreeNodes()` method calls the `getNodes()` method to handle the actual generation of tree nodes. The following code fragment from this `getNodes()` method (defined in `ArmyAgent3`) shows how the method uses the search pattern in its search of the data source:

```
TreeNode[] getNodes(Son parent, String searchPattern)
{
   Vector nodes = new Vector();
   if (searchPattern == null || searchPattern.length() == 0)
      searchPattern = "";
   else
      searchPattern = new String(new char[] {searchPattern.charAt(0)});
```

When the `getNodes()` method later compares the name of the object in the XML file (the data source) with the current name in the parent-node path, it checks if the object name begins with the specified search pattern, as follows:

```
if (currSon.name.getValue().startsWith(searchPattern))
```

For the context of this use of the search pattern, see Figure 61 on page 117.

Querying the data source

The main purpose of the `getTreeNodes()` method is to query the data source to discover source nodes, which are objects for which the ODA can generate content. The mechanism to query the data source depends on the kind of data source with which the ODA works. For example, the XML ODA (a prebuilt ODA that is part of the WebSphere Business Integration Adapters product) queries XML files to present the names of objects within these files for possible content generation. As another example, the JDBC ODA (another prebuilt ODA part of WebSphere Business Integration Adapters) queries a JDBC database to present the names of tables within the database for possible content generation.

As suggested in Table 25 on page 99, if the logic necessary to query your data source is reasonably complex, you should develop special Java classes to handle this interaction. The `getTreeNodes()` method can then instantiate and access these classes as needed. Make sure you include these classes in the ODA library file. For more information, see "Compiling the ODA" on page 159.

For the sample Roman Army ODA, the `findSon()` method (defined in the `ArmyAgent3` class) performs the task of querying the data source. It finds a particular soldier (identified by its parent-node path) in the Roman-Army XML file. It returns the information for the specified name as a `Son` object. The sample defines the `Son` class to read an object in the XML file.

Constructing the tree nodes

As the ODA queries the data source for source nodes, it must generate the associated tree node to represent each source node it discovers. The ODK API represents a tree node as an object of the `TreeNode` class, which contains the information shown in Table 39.

Table 39. Contents of a tree node

Member variable	Description
metadata	

Table 39. Contents of a tree node (continued)

Member variable	Description
name	The name of this tree node, which displays in the Name column of the Select Source dialog box
description	A description of this tree node, which displays in the Description column of the Select Source dialog box
polymorphicNature	Whether the tree node's nature is "normal" (it is either expandable or a leaf node) or "file" (it can be associated with a file)
isExpandable	Whether this tree node is expandable; that is, whether the tree node contains child nodes or is a leaf (terminating) node
isGeneratable	Whether content can be generated for this tree node
Data nodes	An array of child nodes, if this tree node is expandable

To create a tree node, use one of the forms of the `TreeNode()` constructor. For a list of these forms, see "TreeNode()" on page 270.

Normal-nature nodes: When a node has a "normal" nature, it can have one of the following structures:

- An expandable node

  Business Object Wizard displays an expandable node with a plus (+) sign to the left of the node name (to indicate that the user can expand the node) or a minus (-) sign (to indicate that the user can contract the node). The following table shows the initialization that the `TreeNode` object requires for it to display as an expandable node:

TreeNode member variable	Value
isExpandable	true
nodes	An array of the child nodes
isGeneratable	false (usually)

  For information about how to move through the source-node hierarchy, see "Moving through the source-node hierarchy" on page 78

- A leaf node

  Business Object Wizard displays a leaf (terminating) node as just the node name. The following table shows the initialization that the `TreeNode` object requires for it to display as a leaf node:

Tree-node member variable	Value
isExpandable	false
nodes	null
isGeneratable	true (usually)

Both the leaf and expandable node are normal-nature nodes. Therefore, they both have the `polymorphicNature` member variable set to the `NODE_NATURE_NORMAL` node-nature constant. This constant is defined in the `ODKConstant` interface (which the `TreeNode` class implements). The first two forms of the `TreeNode()` constructor do *not* specify the `polymorphicNature` member variable. Therefore, this member variable defaults to `NODE_NATURE_NORMAL`.

Suppose the ODA generates the source-node hierarchy shown in Figure 59 on page 113. If the user expands the Uulius node, the getTreeNodes() method must generate a tree-node array that contains the child nodes for Uulius. Because the parent-node path is *not* empty (it is Apollo:Tellus:Uulius), the getTreeNodes() method makes the following call to the getNodes() method (see Figure 60 on page 114):

```
getNodes(findSon(parentNodePath), searchPattern))
```

This call to getNodes() uses the findSon() method to query the data source for the Uulius node and return a Son object that contains the information from the XML file. One of the member variables in this Son object is a vector of XML objects (XmlObjectVector) with the information on the children of Uulius. Figure 61 shows a code fragment from the getNodes() method that loops through this XML-object vector and creates a TreeNode object for each of the children:

```
for (int i=0; i<sons.size(); i++)
   {
   Son currSon = (Son) sons.getAt(i);
   if (currSon.name.getValues().startsWith(searchPattern))
      {
      int age = currSon.age.getIntValue();
      int children = currSon.Son == null ? 0 : currSon.Son.size();
      int nature = TreeNode.NODE_NATURE_NORMAL;

      TreeNode tn = new TreeNode(currSon.name.getValue(), " ",
         canRecruit(currSon), children > 0, null, nature);

      nodes.add(tn);
      }
   }
```

Figure 61. Constructing tree nodes

The code fragment in Figure 61 initializes a new TreeNode object for each child soldier node with the soldier's name, whether this node is generatable (based on whether the soldier is of recruitable age), and whether this node is expandable (based on the number of children the soldier has). This call to the TreeNode() constructor does *not* initialize the tree node with a description (""), nor does it provide any child nodes. Once each new TreeNode object is instantiated, the code adds it to a Java Vector (nodes). When getNodes() has generated TreeNode objects for *all* child nodes, it copies the contents of this vector into a tree-node array with the following code:

```
TreeNode[] tn = new TreeNode[nodes.size()];
System.arraycopy(nodes.toArray(), 0, tn, 0, nodes.size());
```

The getNodes() method returns this tree-node array to getTreeNodes(), which in turn returns this array to its calling program, Business Object Wizard. Business Object Wizard uses this new tree-node array to display the expanded contents of the Uulius node, as Figure 62 shows.
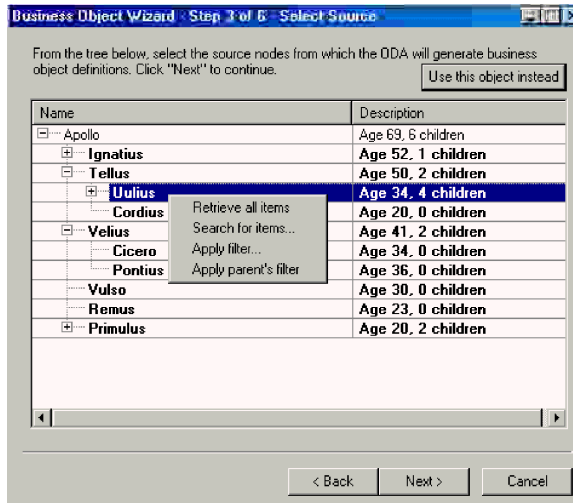
Figure 62. Expanding the Uulius source node

File-nature nodes: When a node has a "file" nature, the user can associate an operating system file with the node. Business Object Wizard indicates that a node has a file nature by activating the Associate files menu item when the user right-clicks the node name. For information about how to use this menu item, see "Associating an operating-system file" on page 80.

A file-nature node has its `polymorphicNature` member variable set to the `NODE_NATURE_FILE` node-nature constant. This constant is defined in the `ODKConstant` interface (which the `TreeNode` class implements). The following table shows the initialization that the `TreeNode` object requires for it to function as a file-nature node:

TreeNode member variable	Value
polymorphicNature	NODE_NATURE_FILE
isExpandable	false
nodes	null
isGeneratable	false (usually)

In the sample Roman Army ODA, the `ArmyAgent4` class implements a `getNode()` method that supports file-nature nodes. This sample allows the user to associate a file with a source node for any node that represents a soldier that is at least 28 years old (the default minimum age) and has no children of his own. The code for this version of `getNode()` is almost identical to the code in Figure 61 on page 117. The only difference is in the assignment of the value to the `polymorphicNature` member variable. Instead of assigning the `NODE_NATURE_NORMAL` constant to *all* nodes, the `ArmyAgent4` version of `getNode()` uses the following code line to set the node nature to `NODE_NATURE_FILE` if the node represents a soldiers at least 28 years old and having no children:

```
int nature = m_allowAdoption && canAdopt(currSon) ?
   TreeNode.NODE_NATURE_FILE : TreeNode.NODE_NATURE_NORMAL;
```

Generating business object definitions

After users have selected the source nodes in the Select Nodes dialog box (Step 3), the ODA is ready to begin content generation. Business Object Wizard calls the `generateBoDefs()` content-generation method to generate business object

definitions for the user-selected source nodes. To the ODA, Business Object Wizard sends the list of source nodes (selected in Step 3). The goal of the business-object-definition generation process is to create a business object definition for each selected source node. While the generateBoDefs() method runs, Business Object Wizard displays its Generating Business Objects screen (Step 5).

Note: Because the ODA generates business object definitions "on request", Business Object Wizard explicitly calls the generateBoDefs() method to initiate generation of the business object definitions. Therefore, you *must* implement generateBoDefs() so that it handles generating the business object definitions (BusObjDef objects), storing them in the generated-content structure, and returning of content metadata to Business Object Wizard.

This section describes the following steps that the generateBoDefs() method should take to generate business object definitions:
1. "Defining the generateBoDefs() method"
2. "Requesting business-object properties"
3. "Creating the business object definitions" on page 123
4. "Providing generated business object definitions" on page 130

Defining the generateBoDefs() method

To provide generation of business object definitions, your ODA class (derived from ODKAgentBase2) must implement the generateBoDefs() method, is defined in the IGeneratesBoDefs interface. The generateBoDefs() method receives these user-selected source nodes as an argument, an array of source-node paths (String objects). The method must generate a business object definition for each source node in this array. It can use its path to locate the source node in the data source. As its last step, generateBoDefs() returns a content-metadata (ContentMetaData) object to describe the business object definitions it has generated.

The sample Roman Army ODA supports the on-request content protocol for the generation of business object definitions (see Figure 58 on page 110). The implementation of this method generateBoDefs() in the ArmyAgent3 class includes the code fragment in Figure 63, which declares the variable for the generated-content structure (m_generatedBOs) and defines the generateBoDefs() method itself.

```
final Vector m_generatedBOs = new Vector();
public ContentMetaData generateBoDefs(String[] nodes)
    throws ODKException
{
```

Figure 63. Defining the generateBoDefs() method

Requesting business-object properties

If, during the content-generation process, the ODA requires additional information, it can display the BO Properties dialog box and request values for *business-object properties*. For an introduction to business-object properties, see "Obtaining business-object properties" on page 93.

Figure 64 illustrates a sample BO Properties dialog box that displays two business-object properties:

- The `Verbs` business-object property allows users to specify which verbs the business object definitions support. This property provides a drop-down list of valid verbs, from which users can choose one or more values.
- The `Prefix` business-object property allows users to enter the prefix (such as `JDBC`, `SAP`, `LegacyApp`) to add to the names of all generated business object definitions. This property just provides an empty field in which users specify the string prefix.
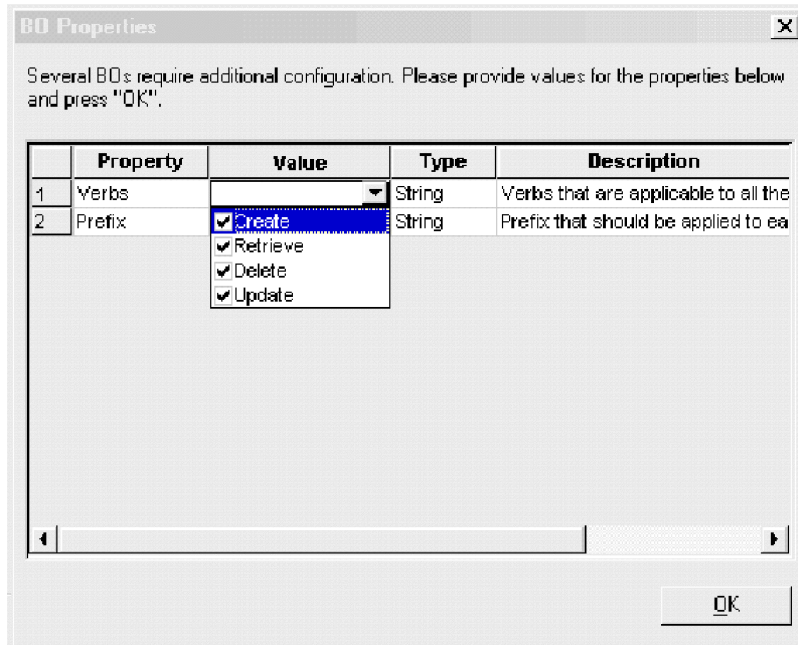

Figure 64. Additional property information needed.

To provide the user with the properties illustrated in Figure 64, the `generateBoDefs()` method takes the following steps:

1. Creates the business-property array for the Verbs and Prefix properties.
2. Calls the `getBOSpecificProps()` method to display the business-object properties.
3. Obtains user-initialized values for the business-object properties.

Creating the business-property array: The `getBOSpecificProps()` method requires an array of agent-property objects as an argument. This argument is the *business-object-property array* and contains one agent-property object for each business-object property to display in the BO Properties dialog box. Before `generateBoDefs()` calls `getBOSpecificProps()`, it must take the necessary steps to create the array that defines the business-object properties, initialize the business-object properties, and save these properties into the array.

The first step is to define a business-object-property array to hold the Verbs and Prefix properties. The next step is to initialize the business-object properties, using the `AgentProperty()` constructor. With this constructor, you specify values for the various metadata that the `AgentProperty` class supports. The `AgentProperty` class provides support for the business-object property to have the following features:

- A default value
- The ability to restrict values to only one value or to more than one value

- A list of valid values for the user to choose from
- Conditions that restrict the value the user can enter

Note: For more information, see "Working with agent properties" on page 140.

To initialize the Verbs and Prefix properties, you provide the `AgentProperty()` constructor with the following information:

- The Verbs property is a multiple-cardinality property that provides multiple values for the user to choose from. It also has default values.

  Therefore, this property requires the following metadata in the `AgentProperty()` constructor:

metadata	AgentProperty member variable	Value
Multiple cardinality	`cardinality`	`ODKConstant.MULTIPLE_CARD`
Allows user to choose from multiple values	`isMultiple`	`true`
	`allValidValues`	`validValues` array (which contains the valid values to display)
Provides default values	`allDefaultValues`	`defaultValues` array (which contains the default values to display)
User is not required to enter a value	`isRequired`	`false`

  Before the call to the `AgentProperty()` constructor, the code in Figure 65 on page 122 first creates and initializes the `validValues` and `defaultValues` arrays so they are available for the constructor.

- The Prefix property is a single-cardinality property that does *not* display multiple values for the user to choose from. It does not have a default value.

  Therefore, this property requires the following metadata in the `AgentProperty()` constructor:

metadata	AgentProperty member variable	Value
Single cardinality	`cardinality`	`ODKConstant.SINGLE_CARD`
Does not allow user to choose from multiple values	`isMultiple`	`false`
	`allValidValues`	`null`
Does not provide default values	`allDefaultValues`	`null`
User is not required to enter a value	`isRequired`	`false`

The code fragment in Figure 65 creates and initializes the business-object-properties array:

```
// Create the business-object-property array
   AgentProperty AgtProps[] = new AgentProperty[2];

 // Provide list of valid values for Verbs property
   Object[] validValues = new Object[4];
   validValues[0] = new String("Create");
   validValues[1] = new String("Retrieve");
   validValues[2] = new String("Delete");
   validValues[3] = new String("Update");

 // Provide list of default values for Verbs property
   Object[] defaultValues = new Object[4];
   defaultValues[0] = new String("Create");
   defaultValues[1] = new String("Retrieve");
   defaultValues[2] = new String("Delete");
   defaultValues[3] = new String("Update");

// Instantiate the Verbs property
   AgtProps[0] = new AgentProperty("Verbs", AgentProperty.TYPE_STRING,
         "Verbs that are applicable to all the selected objects",
         false, true, ODKConstant.MULTIPLE_CARD, validValues,
         defaultValues);

// Instantiate the Prefix property
   AgtProps[1] = new AgentProperty("Prefix", AgentProperty.TYPE_STRING,
         "Prefix that should be applied to each business object name",
         false, false, ODKConstant.SINGLE_CARD, null, null);
```

Figure 65. Creating the business-object array

For more information on the metadata of business-object properties, see "Working with agent properties" on page 140.

Displaying the BO Properties dialog box: Once the business-object-property array is initialized, the ODA can call the getBOSpecificProps() method to pass this array to Business Object Wizard for display to the user in the BO Properties dialog box. This method is defined in the ODKUtility class, and therefore must access an ODKUtility object. Usually, you instantiate this object as part of the ODA initialization. For more information, see "Obtaining the handle to the ODKUtility object" on page 102.

Note: If any property values are invalid, getBOSpecificProps() throws the ODKInvalidPropException exception.

The call to getBOSpecificProps() in Figure 66 sends the AgtProps array (initialized in Figure 65) to Business Object Wizard for display in the BO Properties dialog box.

```
// Display BO Properties dialog box, initializing it with AgtProps
   Util.getBOSpecificProps(AgtProps, "For all the Tables selected");
```

Figure 66. Displaying the BO Properties dialog box

Retrieving the user-specified values: Once users have specified values for the business-object properties in the BO Properties dialog box and clicked **Next**, Business Object Wizard sends the user-specified values back to the ODA. The ODA can retrieve these values in either of the following ways:

- Business Object Wizard saves the user-specified values in a Java Hashtable object and sends this object as the return value for getBOSpecificProps(). Each property is keyed on its name in this Hashtable object. The ODA can access these properties with the Hashtable methods. The user-specified values for the property are in the allValues member variable of its agent-property (AgentProperty) object.

- Business Object Wizard writes the user-specified values into the ODA runtime memory. The ODA can access these values with the `getBOSpecificProperty()` or `getAllBOSpecificProperties()` methods in the `ODKUtility` class.

The `getBOSpecificProps()` call in Figure 66 did not save the `Hashtable` object that Business Object Wizard creates. Therefore, this code fragment uses the `getBOSpecificProperty()` method to get the value of the properties specified for the verbs and each business object prefix:

```
// Get the value of the Verbs and the Prefix properties
AgentProperty propVerb =
      Util.getBOSpecificProperty("Verbs");
AgentProperty propPrefix =
      Util.getBOSpecificProperty("Prefix");
```

Creating the business object definitions

The `generateBoDefs()` method must generate a business object definition for each source node in the array it receives as an argument from Business Object Wizard. To generate a business object definition, `generateBoDefs()` takes the following steps:

1. Use the source node's path, from the array that `generateBoDefs()` receives from Business Object Wizard, to locate the associated object in the data source.
2. Obtain any information needed to populate the business object definition from the associated object in the data source.
3. Create a business-object-definition object to represent the source node.
4. Populate this business-object-definition object with the information obtained from the associated object in the data source (step 2).

The ODK API represents a business object definition as a business-object-definition (`BusObjDef`) object. You can use the `BusObjDef()` constructor to instantiate the new business object definition and provide it with a name. You can then provide the business object definition with the information shown in Table 40.

Table 40. Contents of a business object definition

Business-object-definition information	Description	Accessor method
metadata		
Name	The name of the business object definition	`getName()`
Application-specific information	The business-object-level application-specific information, which contains information applicable to the entire business object definition	`getAppInfo()`, `setAppInfo()`
Data		
Attribute list	A list of the attributes in the business object definition; each attribute is a `BusObjAttr` object.	`getAttributeList()`, `setAttributeList()`, `insertAttribute()`, `removeAttribute()`
Verb list	A list of supported verbs in the business object definition; each verb is a `BusObjVerb` object.	`getVerbList()`, `setVerbList()`, `insertVerb()`, `removeVerb()`

As Table 40 shows, a business object definition contains both metadata and data. The following sections describe how to access these parts of the business object definition:

- "Defining the metadata for the business object definition"
- "Generating attributes" on page 125
- "Supplying supported verbs" on page 129

Defining the metadata for the business object definition: As Table 40 shows, the metadata of a business object definition consists of the following information:

- Name of the business object definition
- Application-specific information (at the business-object-definition level)

Naming the business object definition: The generateBoDefs() method receives the list of user-selected source nodes as an argument. This list is an array of String objects that contains the node paths for the user-selected source nodes. (For information on node paths, see "Determining the parent-node path" on page 112.) With this array, the ODA must create the appropriate name for the business object definition associated with each source node. Usually, the assumption that the ODA makes is that the name of the business object definition matches (or is based on) the name of the data-source object that the source node represents. The ODA must parse the source-node path to obtain the name of the source node, use this source-node name to locate the associated data-source object, then obtain the name from the data-source object.

For example, in the Roman Army sample, the names of the data-source objects and business object definitions match. Therefore, the sample code calls the findSon() method (defined in the ArmyAgent3 and ArmyAgent4 classes) to obtain the data-source object that the source node represents using the source node's node path from the input array of source nodes (nodes), as follows:

```
for (int i=0; i<nodes.length; i++)
   {
   Son sonNode = findSon(nodes[i]);
   BusObjDef sonBo = new BusObjDef(sonNode.name.getValue());
   ...
```

Note: All forms of the BusObjDef() constructor specify the name of the business object definition.

The findSon() method parses the source-node path to obtain the name of the last node in the path.

As another example, suppose the data source is a database and its source nodes represent tables. If the source-node paths include the schema names (*schema*:*table*), your ODA needs to parse the source-node paths to assign just a table name to the corresponding business object definitions. If your ODA supports a user-specified prefix for business object definitions (with a configuration variable), the ODA must prepend this prefix *before* it calls BusObjDef() constructor to create the business-object-definition object, as the following code fragment shows:

```
AgentProperty propPrefix = getBOSpecificProperty("Prefix");
for (int i=0; i<names.length; i++)
   {
   strToken = new StringTokenizer(names[i], ":");
   schemaName = strToken.nextToken();
   tableName = strToken.nextToken()
```

```
if (propPrefix.allValues != null && propPrefix.allValues[0] != null)
   boDef = new BusObjDef(propPrefix.allValues[0] + tableName);
else
   boDef = new BusObjDef(tableName);
...
```

If your data-source objects do *not* have the exact names you want to assign to your business object definitions, the ODA must parse or in some way format these names as needed.

Generating business-object application-specific information: As "Business object application-specific information" on page 7 describes, application-specific information is a powerful way to put application-specific processing information within the business object definition. By moving this information from the processing program (such as a connector), the processing program can be metadata-driven; that is, it can be written in a more generic fashion and obtain its application-specific processing instructions from the business object definition. Therefore, if your business object definitions are to be used with metadata-driven processing programs, it is important that they include the correctly formatted application-specific information at the business-object, attribute, and verb levels.

Note: For information on attribute application-specific information, see "Generating attributes." For information on verb application-specific information, see "Supplying supported verbs" on page 129.

The business object definitions that the Roman Army sample generates do not provide application-specific information. However, suppose the data source was a database with tables as its source nodes. The ODA would generate business object definitions for each user-selected table. In this business object definitions, you might include the name of the table as business-object-level application-specific information. The following code fragment uses the setAppInfo() method, defined in the BusObjDef class, to create the appropriate name-value pairs for this business-object-level application-specific information:

```
boDef.setAppInfo("TN=" + tableName + ";SCN=" + schemaName +";");
```

This code creates the TN and SCH name-value pairs to represent the table and schema names, respectively. It concatenates the table name and schema name with the tag used to name the element. It then uses the setAppInfo() method to assign this entire string as the business-object-level application-specific information.

Generating attributes: A business object definition contains *attributes*, which describe the object that the business object definition represents. The business object definition holds the attributes in its *attribute list*. The ODK API represents an attribute as an attribute (BusObjAttr) object. To instantiate an attribute object, use the BusObjAttr() constructor.

Table 41 summarizes the properties in an attribute object. These properties correspond to the attribute metadata.

Table 41. Properties of an attribute

Attribute property	Description	Accessor method
Name	The name of the attribute	getName(), setName()
Application-specific information	The attribute-level application-specific information, which contains information applicable to the attribute	getAppText(), setAppText()

Table 41. Properties of an attribute (continued)

Attribute property	Description	Accessor method
Type	The data type of the attribute's value	`getAttrType()`, `getAttrTypeName()`, `setAttrType()`
Cardinality	The cardinality of the attribute, which identifies the number of values the attribute holds	`getCardinality()`, `setCardinality()`
Default value	The value to assign to the attribute before the user enters a value	`getDefault()`, `setDefault()`
Maxlength	The maximum length of the attribute's value	`getMaxLength()`, `setMaxLength()`
Comments	Optional comments to describe the purpose of the attribute	`getComments()`, `setComments()`
Relationship type	A string to identify the type of relationship in which the attribute participates	`getRelationType()`, `setRelationType()`
Primary key	Whether the attribute is part of a primary key	`isKey()`, `setIsKey()`
Foreign key	Whether the attribute is part of a foreign key	`isForeignKey()`, `setIsForeignKey()`
Required key	Whether the attribute is required	`isRequiredKey()`, `setIsRequiredKey()`

> **Important**
>
> The business-object-definition generation process automatically creates the `ObjectEventId` attribute. If Business Object Wizard saves the business object definition to a file, it automatically adds the repository version to the top of this file. The repository version is necessary if the integration broker is InterChange Server.

In the sample Roman Army ODA, each business object definition represents a Roman soldier. The `generatesBoDefs()` method creates the following attributes for business object definition:

- The Age attribute holds the roman soldier's age.
- The ChildNo attribute holds the number of children the soldier has (adopted or not).

Figure 67 contains a code fragment that creates these attribute objects for the business object definition.

```
// 1. Create an attribute object for Age attribute
   BusObjAttr attr = new BusObjAttr("Age", BusObjAttrType.INTEGER,
      BusObjAttrType.AttrTypes[BusObjAttrType.INTEGER]);
//    Set the Age attribute as the business object definition's key
   attr.setIsKey(true);
//    Add the attribute to the business object definition's attribute list
   sonBo.insertAttribute(attr);
// 2. Create an attribute object for ChildNo attribute
   attr = new BusObjAttr("ChildNo", BusObjAttrType.INTEGER,
      BusObjAttrType.AttrTypes[BusObjAttrType.INTEGER]);
//    Set the default value to number of children
   attr.setDefault(sonNode.Son == null ? "0" : "" + sonNode.Son.size());
//    Add the attribute to the business object definition's attribute list
   boDef.insertAttribute(attr);
```

Figure 67. Generating attributes

To create the Age attribute, the code fragment in Figure 67 takes the following steps:

1. Use the BusObjAttr() constructor to create the Age attribute object (attr). It uses the form of this constructor that initializes the attribute object with its name, type, and type name.

   To initialize the type, the code specifies the attribute-type constant for Integer (BusObjAttrType.INTEGER). To initialize the type name, it uses the AttrTypes member variable in the BusObjAttrType interface. This static member variable provides the type names for all supported attribute types and can be indexed by the attribute-type constants. In this way, you can assign the type name without hardcoding the type-name string.

2. Use the setIsKey() method to explicitly set the primary-key property to true.

   Because this form of the BusObjAttr() constructor specifies only three attribute properties, all other attribute properties default to "undefined". Therefore, after the BusObjAttr() call, the primary-key attribute property is false. To indicate that the Age attribute *is* the key attribute, the code sample calls setIsKey().

3. Use the insertAttribute(), defined in the BusObjDef class, to add the Age attribute to the business object definition's attribute list.

The code fragment in Figure 67 repeats these basic steps to generate the ChildNo attribute. The main difference is that because ChildNo is *not* the key attribute, no call to setIsKey() is needed. However, the code fragment does provide a default value for this attribute by calling the setDefault() method.

The business object definitions that the Roman Army sample generates are very simple. Only two attributes exist in the business object definition and their names are known at compile time. In addition, only a few attribute properties must be set. For a more complex example, suppose the data source was a database, with tables as its source nodes and as the names of its business object definitions. In this case, the database columns would correspond to the attributes of the business object definition. Many more of the attribute properties would need to be set for these attributes.

The following code fragment creates attributes for the columns in a database table:

```
Vector Attributes;
// 1. Retrieve columns from database table into 'rst' result set
try{
   ResultSet rst = null;
```

```
                  // Retrieve columns from database
                  rst = db.dbmd.getColumns(null, schemaName, tableName, "%");

                  String colName = null;
                  String colType = null;
                  int cType = 0;
                  int colSize = 0;

                  // Obtain next column from result set
                  rst.next();
                  do{
                     // Get column name & type
                     colName = rst.getString(4);
                     colType = rst.getString("DATA_TYPE");

                     // Convert database types to supported types.
                     // Load converted types into the cType variable
                     // (steps not shown)
// 2. Create an attribute object for each column in the result set.
                  Attributes = new Vector(1, 10);

                  try
                    {
                     // Create attribute object for column
                     BusObjAttr attrib = new BusObjAttr(colName, cType);

                     // Set the cardinality and maxLength attribute properties
                     attrib.setCardinality(BusObjAttr.CARD_SINGLE);
                     colSize = rst.getInt("COLUMN_SIZE");
                     attrib.setMaxLength(colSize);

                     // Determine whether it is a primary key in the table: compare
                     // column name against earlier retrieve of table's primary keys
                     // (stored in pKeys -- code not included here)
                     if (pKeys.contains(colName))== true {
                        attrib.setIsKey(true);
                     }else
                        attrib.setIsKey(false);

                     // Determine whether it is a foreign key in the table: compare
                     // column name against earlier retrieve of table's primary keys
                     // (stored in fKeys -- code not included here)
                     if (fKeys.contains(colName))== true {
                        attrib.setIsForeignKey(true);
                     }else
                        attrib.setIsForeignKey(false);

                     // Set the isRequired property
                     if ((rst.getString("IS_NULLABLE").equals("NO")) &&
                           (attrib.isKey() != true)){
                        attrib.setIsRequiredKey(true);
                     }

                     // Create attribute application-specific information:
                     // CN tag provides column name
                     String asi = "CN="+colName;
                     attrib.setAppText(asi);
                     attrib.setDefault("");

                     // Add attribute object to Attributes vector
                     Attributes.add(attrib);
                     ...
// 3. Save the attribute vector as the business object definition's attribute list
                  boDef.setAttributeList(Attributes);
```

The steps in this process are as follows:

1. Use the BusObjAttr() constructor to create a simple business object attribute
   from the column information. This form of the constructor specifies only the
   attribute name and type.

2. Set the cardinality and maxLength attribute properties, based on these values from the column in the database.

   Note: To create an attribute that represents a child business object or an array of child business objects, specify the name of the child business object as the type, and set the cardinality to 1 or n, as appropriate. For example, to create an attribute named `LineItems` that represents an array of OrderLineItems business objects, use the following code:

   ```
   BusObjAttr attrib = new BusObjAttr(LineItems, OrderLineItems);
   attrib.setCardinality(BusObjAttr.CARD_MULTIPLE);
   ```

3. Get primary-key and foreign-key information to set any attributes that represent primary or foreign keys. The code fragment compares the current column name with names in existing arrays that contain primary-key columns (`pKey`) and foreign-key columns (`fKey`) selected from the database. Code that selects the primary- and foreign-key columns is not shown here.

4. Set the "is required key" attribute property, based on whether the attribute is the primary key.

5. Set the attribute-level application-specific information.

   For business object definitions generated for database tables, you might include the name of the column as attribute-level application-specific information for each attribute in the business object definition. This code fragment uses the `setAppText()` method, defined in the `BusObjAttr` class, to create the `CN` name-value pair for the attribute-level application-specific information. The code concatenates the column name with the `CN` tag. It then uses the `setAppText()` method to assign this entire string as the attribute's application-specific information.

6. Use the `setAttributeList()` method, defined in the `BusObjDef` class, to assign the generated attributes vector (`Attributes`) as the attribute list of the business object definition.

Supplying supported verbs: A business object definition contains supported *verbs*, which describe the operations that can be performed on business objects of that business object definition. The business object definition holds its supported verbs in its *verb list*. The ODK API represents a verb as a business-object-verb (`BusObjVerb`) object. To instantiate a verb object, use the `BusObjVerb()` constructor.

Table 42 summarizes the metadata in a verb object.

Table 42. metadata for a verb

Verb metadata	Description	Accessor method
Name	The name of the supported verb (such as Create, Retrieve, Update, or Delete)	`getName()`, `setName()`
Application-specific information	The verb-level application-specific information, which contains information applicable only to the verb	`getAppInfo()`, `setAppInfo()`

In the Roman Army sample, the `generatesBoDefs()` method assigns to each business object definition one supported verb of Create. The following code fragment uses the `insertVerb()` method, defined in the `BusObjDef` class, to add the Create verb to the business object definition's verb list:

```
sonBo.insertVerb("Create", null);
```

The business object definitions that the Roman Army sample generates do not provide application-specific information. Therefore, the second argument to this insertVerb() call, which provides verb application-specific information, is null.

The ODA can use the BO Properties dialog box to obtain verb support for the generated business object definitions. By defining a business-object property called Verbs and allowing users to select the supported verbs, the ODA can obtain more customized verb support. For more information on the use of the BO Properties dialog box, see "Requesting business-object properties" on page 119.

The following code fragment assumes that the ODA has obtained user-specified values for a business-object property called Verbs and uses this property to obtain the verbs to the business object definition's verb list.

```
Vector Verbs;
AgentProperty propVerbs = getBOSpecificProperty("Verbs");

if (propVerbs.allValues[0] != null)
    {
    int len = propVerbs.allValues.length;
    BusObjVerb verb;

    for(int i=0; i<len; i++)
        {
        if(propVerbs.allValues[i] != null)
            {
            try {
                verb = new BusObjVerb(propVerbs.allValues[i].toString(), "");
                Verbs.add(verb);
            }
        }
    }
...
boDef.setVerbList(Verbs);
```

This code fragment uses the BusObjVerb() constructor to copy a verb into the verb variable of type BusObjVerb. It then loads a String version of that verb object into the Verbs vector. The code does not specify verb application-specific information. Finally, the code fragment uses the setVerbList() method, defined in the BusObjDef class, to assign the generated verbs vector (Verbs) as the verb list of the business object definition.

Providing generated business object definitions

As discussed in "Providing generated content" on page 94, the ODA must return the generated content to Business Object Wizard in two parts. Therefore, if the ODA generates business object definitions as content, it must return the following:

- A generated-content structure, which contains the generated business object definitions
- A content-metadata (ContentMetaData) object that describes the business object definitions in the generated-content structure

Because the ODA must generate business object definitions "on request", the generateBoDefs() method provides this content information, as follows:

- It populates the generated-content structure. This structure must in some way be to global to the ODA class so that both generateBoDefs() and getBoDefs() can access it.
- It returns a content-metadata (ContentMetaData) object that describes the generated business objects to its caller, Business Object Wizard.

Once Business Object Wizard receives this content-metadata object, it can access the generated business object definitions (within the generated-content structure) as needed with the `getBoDefs()` method.

Note: For more information on `getBoDefs()`, see "Providing access to generated business object definitions."

The code sample in Figure 68 shows the last part of the `generateBoDefs()` method for the sample Roman Army ODA.

```
        m_generatedBOs.add(sonBo);
    } // this for loop terminates when all bus obj defs are generated
    return new ContentMetaData(ContentType.BusinessObject, -1,
        m_generatedBOs.size());
} // end of generateBoDefs()
```

Figure 68. Providing generated business object definitions

Figure 68 handles the generated content as follows:

- The `generatedBoDefs()` method saves the generated business object definition in its generated-content structure, `m_generatedBOs`.

  As shown in Figure 63 on page 119, the Roman Army sample uses a Java vector called `m_generatedBOs` as its generated-content structure. This structure is global to the methods of the Roman Army ODA class. To save the business object definition it has generated, `generateBoDefs()` saves it in the `m_generatedBOs` vector. This step is within a loop that terminates when the ODA has generated business object definitions for *all* source nodes that users selected. When Business Object Wizard needs to access the generated-content structure, it calls the content-retrieval method, `getBoDefs()`.

- As its last step, `generateBoDefs()` returns the content-metadata object that describes the generated content.

  The `generateBoDefs()` method instantiates a `ContentMetaData` object and into this constructor passes the information shown in Table 43.

Table 43. Initializing the content metadata for business-object-definition generation

ContentMetaData information	Code	Description
Content type	`ContentType.BusinessObject`	Indicates that the content type is business object definitions
Size of the generated content	-1	Indicates that total size is *not* required. The length value is *not* needed in the current implementation of a `ContentMetaData` object.
Count of the generated content	`m_generatedBOs.size()`	The `size()` method returns the number of elements currently in the vector.

Providing access to generated business object definitions

The `generateBoDefs()` method does *not* return the actual generated business object definitions. For Business Object Wizard to be able to access the generated content, the ODA class must implement the content-retrieval method for business object definitions. Business Object Wizard uses the information in the content-metadata

object (which generateBoDefs() *does* return) to determine whether to call the appropriate content-retrieval method. If generateBoDefs() has successfully generated business objects, Business Object Wizard calls the getBoDefs() method to retrieve the generated business object definitions.

Note: In this release, Business Object Wizard always calls the generateBoDefs() method to initiate generation of business object definitions because the ODA must support the on-request content protocol. The ODA should *not* support the callback content protocol for generation of business object definitions. For more information on content protocols, see "Choosing the ODA content protocol" on page 108.

To provide access to generated business object definitions, the ODA class must implement the getBoDefs() method. This method is defined as part of the IGeneratesBoDefs interface. The method accepts as an argument an index, which identifies the number of business object definitions to return. It access these business object definitions in the generated-content structure and returns an array of the retrieved business-object-definition (BusObjDef) objects. The number of business object definitions in this array depends on the value of the index argument, as Table 44 shows.

Table 44. Retrieving business object definitions

Value of index	Description	Number of elements in array that getBoDefs() returns
In the range 0 to *count* - 1, where *count* is the total number of business object definitions in the generated-content structure	Specifies the index position into the generated-content structure of the business object definition to retrieve	One business object definition
ODKConstant.GET_ALL_OBJECTS	Special constant to indicate the return of *all* business object definitions in the generated-content structure	All business object definitions in the generated-content structure (*count*)

For the sample Roman Army ODA, the generateBoDefs() method (defined in the ArmyAgent3 class) populates the m_generatedBOs vector with its generated business object definitions. Therefore, the getBoDefs() method (also defined in ArmyAgent3) retrieves the specified number of business object definitions from this vector and copies them into its return array. The following code shows the getBoDefs() method for the sample Roman Army ODA:

```
public BusObjDef[] getBoDefs(long index) throws ODKException
{
   BusObjDef[] bos = null;
   if (index == ODKConstant.GET_ALL_OBJECTS)
      {
      bos = new BusObjDef[m_generatedBOs.size()]
      System.arraycopy(m_generatedBOs.toArray(), 0, bos, 0,
        m_generatedBOs.size());
      }
   else
      bos = new BusObjDef[] {(BusObjDef)m_generatedBOs.get((int)index)};
   return bos;
}
```

Generating binary files as content

A *binary file* is an operating-system file, which is represented as a Java `File` object. For an ODA to generate binary-file content, its ODA class must implement the `IGeneratesBinFiles` interface. Table 45 lists the methods that the ODA class must define to implement the `IGeneratesBinFiles` interface.

Table 45. Methods in the IGeneratesBinFiles interface

Method	IGeneratesBinFiles method	Description
Source-node-generation method	None	Generation of source nodes must be performed by the `getTreeNodes()` method of the `IGeneratesBoDefs` interface. For more information, see "Using files."
Content-generation method	`generateBinFiles()`	Generates the binary files, writing them to ODA memory
Content-retrieval method	`getBinFile()`	Retrieves either a specified binary file or all binary files from ODA memory

Note: In addition to the methods in Table 45, `IGeneratesBinFiles` also includes the `getContentProtocol()` method to specify the content protocol that the ODA supports for file generation. For more information, see "Choosing the ODA content protocol" on page 108.

Business Object Wizard generates and retrieves content while it displays the Generating Business Objects (Step 5) dialog box. With the `IGeneratesBinFiles` interface implemented, Business Object Wizard invokes the methods shown in Table 46 to generate and retrieve content.

Table 46. Business Object Wizard and IGeneratesBinFiles methods

Use of method	IGeneratesBinFiles method	For more information
Generate files as content	`generateBinFiles()`	"Generating files" on page 135
Retrieve the generated files	`getBinFile()`	"Providing access to generated files" on page 139

The following sections discuss the implementation of each of the methods in Table 46.

Using files

When an ODA that implements the `IGeneratesBinFiles` interface, it can support the use of operating-system files in the following contexts:

- The ODA can *create* new files to support generation of file content.
- The ODA can *read* existing files to support the association of files with source nodes.

Creating files for file content

When an ODA implements the `IGeneratesBinFiles` interface, it supports creation of files as content. The files that the ODA creates hold the information that the ODA collects from the business-object-definition generation process and elsewhere. If the file-generation process needs the array of user-selected source nodes (which Business Object Wizard creates as a result of Step 3, Select Source), the ODA can

receive this array from Business Object Wizard. For information on how to implement the method that generates the files, see "Generating files" on page 135.

However, the `IGeneratesBinFiles` interface does *not* define a source-node-generation method, which discovers source nodes and generates the array of tree nodes for Business Object Wizard to display in the Select Source dialog box. Instead, if the ODA supports generation of file content and this file generation requires an array of user-selected source nodes, the ODA must use the source-node-generation method in the `IGeneratesBoDefs` interface, `getTreeNodes()`. This method queries the data source for the child nodes of the specified parent node and constructs the associated tree nodes, as described in "Generating source nodes" on page 111.

Note: In this release, every ODA is required to support generation of business object definitions. Therefore, it must implement the `IGeneratesBoDefs` interface and all its methods (including the `getTreeNodes()` method).

If an ODA supports *only* creation of new files (file generation), it can use the `getTreeNodes()` method as defined in `IGeneratesBoDefs`. This method queries the data source for the child nodes of the specified parent node and constructs the associated tree nodes, as described in "Generating source nodes" on page 111.

Reading files for source data

When an ODA implements the `IGeneratesBinFiles` interface, it can support reading operating system files that you associate with source nodes (For information on how to associate a file with a node, see "Associating an operating-system file" on page 80.). The files that the ODA reads hold source data, which the ODA must search for objects that are represented as source nodes. To support the association of files with nodes, an ODA must take the following steps:

- Set the node nature of any tree node that can have a file associated with it to "file" (its `polymorphicNature` member variable set to `ODKConstant.NODE_NATURE_FILE`). For more information, see "File-nature nodes" on page 118.
- Implement any methods that need to access objects represented by source nodes so that they query not just the ODA's data source, but any file associated with a source node. To retrieve the contents of an operating-system file specified by its source-node path, use the `getClientFile()` method, defined in the `ODKUtility` class.

Important

An ODA must implement the `IGeneratesBinFiles` interface for the `getClientFile()` method to successfully retrieve a specified operating-system file. If the ODA implements only the `IGeneratesBoDefs` interface, `getClientFile()` throws the `UnsupportedContentException` exception.

The `getClientFile()` method expects as an argument the source-node path of the file to retrieve. This source-node path has the following format:

fileNodePath:*filePath*

where *fileNodePath* is the node path (node names separated by colon (:)) of the node that has an associated file and *filePath* is the operating-system path of the associated file. When users expand or select a node that is an associated file, Business Object Wizard creates this path for the node.

For example, the `ArmyAgent5` class of the sample Roman Army ODA supports both the `IGeneratesBinFiles` interface and the association of files with nodes. Suppose you associate the `Flavius.xml` file (in the directory `C:\IBM\XMLFiles`) to the Vulso source node, as shown in Figure 49 on page 81.. If you select the `Flavius.xml` node (see Figure 50 on page 82) from the source-node hierarchy, Business Object Wizard puts the following node path into the array of source nodes:

`Apollo:Vulso:Flavius.xml:C:\IBM\XMLFiles\Flavius.xml`

This ODA provides the `findSon()` method to parse a source-node path and locate the associated object that the source node represents. The version of `findSon()` in the `ArmyAgent3` class queries only the ODA's data source (an XML file called `RomanArmy.xml`) for the object associated with the specified source node. A revised version in the `ArmyAgent4` class adds the ability to query an associated file by providing the `remoteSon()` method, which uses `getClientFile()` to obtain the contents of the specified file and return this content as a `Son` object.

Note: The `ArmyAgent4` class, which implements the `remoteSon()` method, does *not* support the `IGeneratesBinFiles` interface. Therefore, the `remoteSon()` method catches the `UnsupportedContentException` exception that `getClientFile()` throws and creates a "dummy" `Son` object (see Figure 76 on page 158). The `ArmyAgent5` class, which extends `ArmyAgent4`, *does* implement `IGeneratesBinFiles`. Therefore, this version of the ODA can fully support access to files associated with source nodes with `getClientFile()`.

If a source node can have a file associated with it, then the ability to interpret the file's source-node path and to read the contents of this file is needed during content generation, the method that generates content must be able to access information in nodes that are in a file. Implement the method that generates content so that it uses `getClientFile()` to retrieve an operating-system file that is associated with a node. The method that provides this support is as follows:

• The `generateBoDefs()` method generates business object definitions. The `getClientFile()` method provides the contents of the specified file so that `generateBoDefs()` can obtain the information it needs to create a business object definition. If the `generateBoDefs()` method has already been implemented to obtain source-node information from the ODA's data source, it must be enhanced so that it can obtain information from an associated file as well.

   Note: The `getClientFile()` method *cannot* retrieve a specified file's contents when called from within `generateBoDefs()` unless the ODA also implements the `IGeneratesBinFiles` interface.

• The method that generates files depends on the content protocol that the ODA supports. For on-request generation, the `generateBinFiles()` method generates the files. For callback generation, a user-defined method generates the files. In either case, the `getClientFile()` method provides the contents of the specified file so that the method can obtain the information it needs to create a file.

For more information on how to generate file content, see "Generating files."

Generating files

After users have selected the source nodes in the Select Nodes dialog box, the ODA is ready to begin content generation. The goal of the file generation process is to create a file (or files) that the ODA or other process requires. The step that initiates the generation of files depends on the content protocol associated with the file content type (`ContentType.BinaryFile`), as follows:

- If files are to be generated *on request*, Business Object Wizard initiates content generation by calling the content-generation method, `generateBinFiles()`. This method is part of the `IGeneratesBinFiles` interface.
- If files are to be generated through *callbacks*, the ODA initiates content generation in a user-defined way. Business Object Wizard does *not* call `generateBinFiles()` but waits for a "content generation is complete" signal from the ODA before it accesses the generated content.

This section describes the following steps that the `generateBinFiles()` method should take to generate files:

1. "Defining the generateBinFiles() method"
2. "Requesting properties for file information" on page 137
3. "Creating the files" on page 137
4. "Providing generated files" on page 138

Defining the generateBinFiles() method

The `generateBinFiles()` method is defined in the `IGeneratesBinFiles` interface. Therefore, your ODA class (derived from `ODKAgentBase2`) must implement this method when it implements the `IGeneratesBinFiles` interface. The purpose of the `generateBinFiles()` method depends on the content protocol that the ODA uses for generation of file (`ContentType.BinaryFile`) content, as follows:

- If the ODA generates files "on request", Business Object Wizard explicitly calls the `generateBinFiles()` method to generate the files.
- If the ODA generates files through callbacks, Business Object Wizard *never* explicitly calls the `generateBinFiles()` method. Instead, the ODA uses some other way to generate the files, which Business Object Wizard can then access.

Generating files on request: If the ODA generates files "on request", Business Object Wizard explicitly calls the `generateBinFiles()` method to initiate generation of the files. Therefore, you must implement `generateBinFiles()` so that it handles generating the file objects, storing them in the generated-content structure, and returning of content metadata to Business Object Wizard.

While the `generateBinFiles()` method runs, Business Object Wizard displays its Generating Business Objects screen (Step 5). As its last step, `generateBinFiles()` returns a content-metadata (`ContentMetaData`) object, which describes the generated files it has generated (though it does *not* contain the actual generated files).

Generating files through callbacks: If the ODA generates files through callbacks, Business Object Wizard *never* explicitly calls the `generateBinFiles()` method. Instead, the ODA uses some other way to "spontaneously" generate the files. You must develop a method to handle generating the files, storing them in the generated-content structure, and notifying Business Object Wizard that content generation is complete. However, the `IGeneratesBinFiles` interface requires that you define the `generateBinFiles()` method. Therefore, you must implement `generateBinFiles()` so that it warns the caller that it should never be called.

The sample Roman Army ODA supports the callback content protocol for the generation of files (see Figure 58 on page 110). It defines the `generateBinDefs()` method in the `ArmyAgent5` class. This implementation of the method includes the code in Figure 69, which defines the `generateBinFiles()` method so that it throws an exception if it is ever called.

```
public ContentMetaData generateBinFiles(String[] nodes)
    throws ODKException
{
   throw new ODKException(
      "Files are produced as callbacks. Do not call for file generation.");
}
```

Figure 69. Defining the generateBinFiles() method

As an alternative to throwing an exception, the `generateBinFiles()` method can use the `contentUnavailable()` method (defined in `ContentMetaData`) to return its content metadata to Business Object Wizard, as follows:

```
return (ContentMetaData.contentUnavailable(ContentType.BinaryFile));
```

Requesting properties for file information

If, during the process of generating the files, the ODA requires additional information, it opens the BO Properties dialog box where users can provide values for business-object properties. Even though these properties are called business-object properties, you can use the `getBOSpecificProps()` method to display information that the file-generation process might require. For more information on how to use the BO Properties dialog box, see "Requesting business-object properties" on page 119.

Creating the files

The ODK API does not provide a special class to represent a binary file because Java already provides the `File` class in its `java.io` package. This package contains many input/output classes that can be useful in the generation and access of files. For each file that the ODA generates, it must take the following steps:

- Create a new `File` object with the appropriate file name.
- Write the contents to this file, closing the file when writing is complete.

The actual file generation that your ODA performs depends on the design of the ODA. Implement the file generation as best fits the requirements of your ODA and any components that require the files.

The `ArmyAgent5` class of the sample Roman Army ODA defines a separate class, `FileCreator`, to handle the actual generation of the files. To simulate "spontaneous" file generation, the sample calls the `FileCreator()` constructor from the `generateBoDefs()` method, as the following code fragment shows:

```
public ContentMetaData generateBoDefs(String[] nodes) throws ODKException
{
   ContentMetaData cmd = super.generateBoDefs(nodes);
   new FileCreator(this, nodes).start();
   return cmd;
}
```

The `FileCreator()` constructor spawns a thread to generate the files. It receives as an argument a reference to the current ODA object (`this`) and the array with the node paths of the selected source nodes. It then creates the following files:

- The `stats.zip` file, which contains the number of business object definitions that the ODA has generated
- The `adopted.txt` file, if any user-selected source nodes are adopted children

Providing generated files

As discussed in "Providing generated content" on page 94, the ODA must return the generated content to Business Object Wizard in two parts. Therefore, if the ODA generates files as content, it must return the following:

- A generated-content structure, which contains the generated files.
- A content-metadata (`ContentMetaData`) object that describes the files in the generated-content structure.

The method that provides this information depends on the content protocol that the ODA uses to generate files, as follows:

- If the ODA generates files "on request", the `generateBinFiles()` method provides this content information.
- If the ODA generates files through callbacks, a user-defined method must provide this content information.

Providing content for on-request files: If the ODA generates files "on request", Business Object Wizard invokes the `generateBinFiles()` method to handle file generation. Therefore, `generateBinFiles()` provides the generated content as follows:

- It populates the generated-content structure. This structure must in some way be visible to both `generateBinFiles()` and `getBinFile()`, so that both can access it.
- It returns a content-metadata (`ContentMetaData`) object that describes the generated files to its caller, Business Object Wizard.

    Once Business Object Wizard receives this content-metadata object, it can access the generated files (within the generated-content structure) as needed with the `getBinFile()` method.

For more information on `getBinFile()` , see "Providing access to generated files" on page 139.

Providing content for callback-generated files: If the ODA generates files through callbacks, Business Object Wizard does *not* invoke the `generateBinFiles()` method to handle file generation. Instead, the ODA uses some user-defined method to "spontaneously" generate files. This method could be part of the ODA class or in a class within the ODA's package. However, it must provide the generated content as follows:

- It populates the generated-content structure. This structure must in some way be visible to the user-defined method that generates the files *and* to `getBinFile()` (which the ODA class implements), so that both methods can access it.
- It sends a content-metadata (`ContentMetaData`) object that describes the generated files to Business Object Wizard.

    The user-defined method that generates files cannot return the content metadata directly to Business Object Wizard because Business Object Wizard has not invoked this method. Instead, the method must send a "content generation is complete" signal to Business Object Wizard by calling the `contentComplete()` method (defined in the `ODKUtility` class). This method accepts a content-metadata object as an argument. See Table 47 on page 139 for the information that this content-metadata object should contain. It sends this content metadata to Business Object Wizard. Once Business Object Wizard receives the content-metadata object, it can use the `getBinFile()` method to access the generated files (within the generated-content structure).

Note: For more information on getBinFiles(), see "Providing access to generated files."

In the ArmyAgent5 class of the sample Roman Army ODA, the generated-content structure is defined an array of File objects called m_files, as follows:

```
File[] m_files = null;
```

The code fragment in Figure 70 shows the last part of the FileCreator.run() method (defined in the ArmyAgent5.java file):

```
        for (int i=0; i<fileV.size(); i++)
            m_agent.m_files[i] = (File) fileV.get(i);
    }
    ODKUtility.getODKUtility.contentComplete(
        new ContentMetaData(ContentType.BinaryFile, 0,
        m_agent.m_files.length);
} // end of run() in FileCreator class
```

Figure 70. Providing file content

Figure 70 handles the generated content as follows:

* The FileCreator.run() method saves the generated files in the generated-content structure, m_files.

  The Roman Army sample ODA uses the m_files array as its generated-content structure. To save the files it has generated, run() saves them in this m_files array. This step occurs after run() has generated *all* files. Business Object Wizard can access the m_files array through a call to the content-retrieval method, getBinFile().

* As its last step, FileCreator.run() sends the content-metadata object that describes the generated content to Business Object Wizard.

  The run() method calls the contentComplete() method, passing it a new ContentMetaData object. Into this ContentMetaData() constructor, run() passes the information shown in Table 47.

Table 47. Initializing the content metadata for file generation

ContentMetaData information	Code	Description
Content type	ContentType.BinaryFile	Indicates that the content type is files
Size of the generated content	0	Indicates that total size is *not* required. The length value is not needed in the current implementation of a ContentMetaData object.
Count of the generated content	m_files.length	The length member variable contains the number of elements currently in the array.

Providing access to generated files

The generateBinFiles() method does *not* return the actual generated business object definitions. For Business Object Wizard to be able to access the generated content, the ODA class must implement the content-retrieval method for files. Business Object Wizard uses the information in the content-metadata object (which generateBinFiles() *does* return) to determine which content-retrieval method to

call. For file content, Business Object Wizard calls the getBinFile() method to retrieve the generated business object definitions.

Note: Business Object Wizard calls the generateBinFile() method for generation of files if the ODA supports the on-request content protocol. If the ODA supports the callback content protocol for generation of files, a user-defined method actually generates the files. However, this method does not return the actual generated content either. Therefore, Business Object Wizard *still* requires the getBinFile() method to access the generated files.

Regardless of the content protocol your ODA supports for generation of files, the ODA class must implement the getBinFile() method. This method is defined as part of the IGeneratesBinFiles interface. The method accepts as an argument an index, which identifies the number of files to return. It accesses these files in the generated-content structure and returns an array of the retrieved file (File) objects. The number of files in this array depends on the value of the index argument, as Table 48 shows.

Table 48. Retrieving files

Value of index	Description	Number of elements in array that getBinFile() returns
In the range 0 to *count* - 1, where *count* is the total number of files in the generated-content structure	Specifies the index position into the generated-content structure of the file to retrieve	One file object
ODKConstant.GET_ALL_OBJECTS	Special constant to indicate the return of all files in the generated-content structure	All file objects in the generated-content structure (*count*)

For the sample Roman Army ODA, the FileCreator.run() method (defined in the ArmyAgent5 class) populates the m_files array with the generated files. Therefore, the getBinFile() method (also defined in ArmyAgent5) retrieves the specified number of files from this array. The following code shows the getBinFile() method for the sample Roman Army ODA:

```
public File[] getBinFile(long index) throws ODKException
{
   if (index == ODKConstant.GET_ALL_OBJECTS)
      return m_files;
   else
      return new File[] {m_files[(int)index]};
}
```

Working with agent properties

There are two situations in which an ODA provides agent properties to Business Object Wizard:

- To provide initialized ODA configuration properties (in the Configure Agent dialog box)
- To provide initialized business-object properties (in the BO Properties dialog box)

To represent an agent property, the ODK API defines an agent-property object, which is an instantiation of the AgentProperty class. When you instantiate the agent-property object, you initialize some or all of its member variables, shown in

Table 49.

Table 49. Contents of an agent-property object

Member variable	Description
propName	The name of the agent property
description	A text string that describes the purpose of the agent property
type	The data type of the agent property, as represented by a property-type constant
cardinality	The cardinality of the agent property; that is, whether the property can have one or multiple values
isHidden	Determines whether Business Object Wizard displays the property value as normal text or in an encrypted format.
isMultiple	Determines whether Business Object Wizard displays a drop-down list of valid values for the agent property, for users to choose from
isReadOnly	Determines whether the agent property's value is read-only; that is, whether users can change the displayed value
isRequired	Determines whether the agent property's value is required; that is whether users are required to specify a value
allDefaultValues	An array of default values for the agent property
allDependencies	An array of conditions for the agent property
allValidValues	An array of valid values for the agent property
allValues	An array of user-initialized values for the agent property

To instantiate an agent-property object, use one of the forms of the `AgentProperty()` constructor:

- The first form defines a new agent-property object and initializes it with *only* a property name.
- The second form defines a new agent-property object and initializes it with *all* member variables.
- The third form defines a new agent-property object and initializes it with all member variables *except* `isHidden` and `isReadOnly`.

Defining the agent property

Table 50 shows the basic information about an agent property that the agent-property object contains.

Table 50. Basic information for an agent property

Basic property information	AgentProperty member variable	Description
Name	propName	Identifies the agent property. Business Object Wizard displays this value in the Property column of the Configure Agent (configuration property) or BO Properties (business-object property) dialog box. You can initialize the agent property's name with any of the forms of the `AgentProperty()` constructor.

Table 50. Basic information for an agent property (continued)

Basic property information	AgentProperty member variable	Description
Description (optional)	description	Provides additional information to describe the purpose of the agent property. Business Object Wizard displays this value in the Description column of the Configure Agent (configuration property) or BO Properties (business-object property) dialog box. You must use either the second or third form of the `AgentProperty()` constructor to initialize the agent property's description.
Data type	type	Defines the data type of the values that the agent property holds. Business Object Wizard displays this value in the Type column of the Configure Agent (configuration property) or BO Properties (business-object property) dialog box. If you use the first form of the `AgentProperty()` constructor to initialize the agent property (which specifies only the property name), the agent property's type defaults to String. To specify a type, use the second or third form of the `AgentProperty()` constructor to initialize the agent property. Represent the agent property's type with one of the property-type constants shown in Table 67 on page 173.

Defining the property value

Business Object Wizard displays the agent-property value in the Value column of the Configure Agent (configuration property) or BO Properties dialog box. As part of the process of initializing an agent property, you must address the following tasks:

- "Choosing the type of display control"
- "Specifying default values" on page 144
- "Initializing a single-cardinality property" on page 145
- "Initializing a multiple-cardinality property" on page 145

Choosing the type of display control

Business Object Wizard uses the following `AgentProperty` metadata to determine the type of control for displaying the property value:

- The `isMultiple` parameter determines whether the control for the property's value displays multiple values in a drop-down list. To initialize the drop-down list with values, you can specify the values in the `allValidValues` array.
- The `cardinality` parameter determines whether the control allows users to specify one or multiple values for the property:

Cardinality	Description	Cardinality constant
Single	The property can hold only one value. Therefore, users can specify *only one* value for the property.	`ODKConstant.SINGLE_CARD`
Multiple (n)	The property can hold one or more values. Therefore, users can specify *multiple* values for the property.	`ODKConstant.MULTIPLE_CARD`

Table 51 illustrates the possible combinations for displaying the property-value control.

Table 51. Possible property-value control types.

Cardinality	Displays multiple values (isMultiple)	Are valid values (all ValidValues) provided?	Explanation
1	`false`	No	The property value displays as a plain edit control; that is, a simple box in which users can enter and edit one value.
1	`true`	Yes	The property value displays as a drop-down list that displays the specified valid values (see Figure 71 on page 144). From this list, users can choose only one value.
n	`true`	Yes	The property value displays as a drop-down list that contains the specified valid values. Each value in this list displays with a check box that, if selected, allows the value to be included in the property's value set (see Figure 71 on page 144).
n	`true`	No	The property value displays as a grid control that contains no displaying values. Initially, this grid displays a sub-grid with one empty row. If users enter text in that row, Business Object Wizard inserts another empty row. This process continues until users finish entering new rows. To delete a value, users delete the value's text. Business Object Wizard includes only non-empty rows in the property's value set.

When Business Object Wizard displays a single-cardinality property that does *not* have valid values, it just leaves the property's Value field empty. You can, however, define a default value for the property. In this case, Business Object Wizard displays the default value in the Value field. For more information, see "Specifying default values" on page 144.

Figure 71 illustrates two controls that display multiple values (`isMultiple = true`) in Business Object Wizard.

Single-cardinality control   Multiple-cardinality control
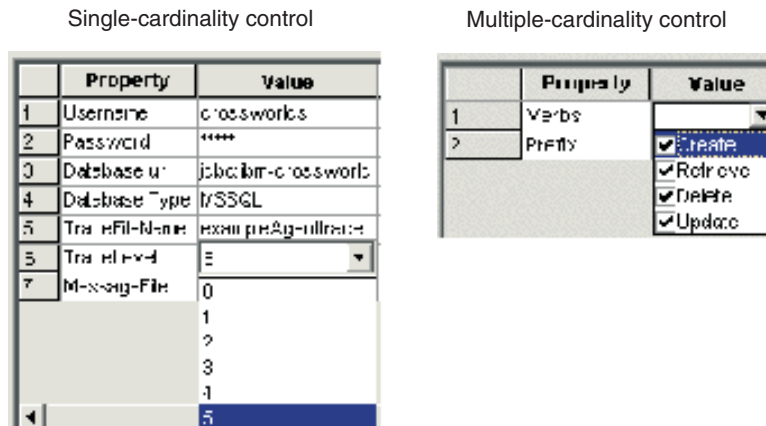
Figure 71. Single- and multiple-cardinality controls for properties with multiple values.

Figure 71 shows single- and multiple-cardinality controls, both of which display multiple values in a drop-down list:

- The single-cardinality control (on the left in Figure 71) displays multiple trace levels in a drop-down list, but allows users to select *only one* value (`cardinality = ODKConstant.SINGLE_CARD`) from this list.
- The multiple-cardinality control (on the right in Figure 71) displays multiple verbs in a drop-down list and allows users to select *any number* of them (`cardinality = ODKConstant.MULTIPLE_CARD`) from this list.

Specifying default values

To specify a default value for an agent property, you provide its default value (or values) in the its `allDefaultValues` member variable. This member variable is an array of `Object` values. The number of elements in this array must correspond to the cardinality of the property, as follows:

- For a single-cardinality property, the `allDefaultValues` array must contain only one element.
- For a multiple-cardinality property, the `allDefaultValues` array can contain one or more elements.

Business Object Wizard assigns the default value to the property before it displays the property. If users do not override this default by specifying a property value, this default value remains as the property value.

Note: Any valid values specified for the property are *not* automatically its default values. You must explicitly specify default values.

Table 52 summarizes the behavior of default values.

Table 52. Default values for agent properties

Cardinality	Contents of allDefaultValues	Display
Single	One element	With valid values (`isMultiple=true`): default value displays as a "checked" item in the drop-down list of valid values. With no valid values (`isMultiple=false`): default value displays in the property's Value field.

Table 52. Default values for agent properties (continued)

Cardinality	Contents of allDefaultValues	Display
Multiple	One or more elements	Default values display as "checked" items in the drop-down list of valid values.

Initializing a single-cardinality property

To initialize a single-cardinality agent property, take the following steps:

- Restrict the number of values that users can specify to one. Set the property's `cardinality` member variable to `ODKConstant.SINGLE_CARD`.
- Determine whether to provide a list of valid values from which users can choose the property's single value. If you provide a list of valid values:
  - Set the `isMultiple` variable to `true`.
  - Initialize the valid-values (`allValidValues`) array with the list of valid values.

  If you do *not* provide a list of valid values, set the `isMultiple` variable as `false` and do not pass in a valid-value array.
- Optionally, initialize the `allDefaultValues` array to contain an `Object` with the single default value.

The following code fragment initializes a single-cardinality agent property that does not provide a list of values to choose from, and has a default value of 256:

```
defaultVal[0] = 256;
AgentProperty("Property1", AgentProperty.TYPE_INTEGER, "Description of property",
   false, false, ODKConstant.SINGLE_CARD, null, defaultVal);
```

Initializing a multiple-cardinality property

To initialize a multiple-cardinality agent property, take the following steps:

- Indicate that users can specify a number of values for the property. Set the property's `cardinality` member variable to `ODKConstant.MULTIPLE_CARD`.
- Indicate that Business Object Wizard needs to handle entry of multiple values for the property. Set the property's `isMultiple` member variable to `true`.
- Determine whether to provide a list of valid values from which users can choose. If you provide a list of values, initialize the list of valid values in the `allValidValues` array. These values initialize the property's drop-down list.

  If you do *not* provide a list of valid values, Business Object Wizard provides a sub-grid for users to specify each property value.
- Optionally, initialize the `allDefaultValues` array to contain an `Object` for each default value.

The code fragment in Figure 65 on page 122 initializes a multiple-cardinality agent property named Verbs, which has a list of valid values and default values.

Setting conditions on the property value

The `AgentProperty` class provides the ability to define conditions on an agent property. A *condition* can restrict the values of one agent property, called a *dependent property*, based on the value of another agent property. A condition has two parts, each part a particular kind of subcondition, as Table 53 shows.

Table 53. Parts of an agent-property condition

Subcondition	Description	ODK API class
Input condition	Defines a condition on the current agent property's value	`InputCondition`
Dependent condition	Defines a condition that must be met on the dependent property when the associated input condition evaluates to true	`DependentCondition`

Defining the complete condition

To represent a condition, the ODK API defines a complete-condition object, which is an instantiation of the `CompleteCondition` class. Table 54 shows the member variables that a complete-condition object contains.

Table 54. Contents of a complete-condition object

Member variable	Description
`allInputConditions`	An array of input-condition (`InputCondition`) objects, each object defining one condition on the value of the agent property
`allDependentConditions`	An array of dependent-condition (`DependentCondition`) objects, each object defining one restriction on the value of a dependent property. This restriction applies to the dependent property's value when the associated input conditions (in the `allInputConditions` array) evaluate to true.

A complete-condition object contains the information that describes a *single* condition on an agent property. An agent property can have many conditions defined on it. Each condition's complete-condition object is stored in the `allDependencies` member variable of the agent property's `AgentProperty` object.

To create one condition on an agent property, take the following steps:

1. Instantiate a `CompleteCondition` object to hold the condition information.

2. Instantiate the appropriate `InputCondition` objects to describe input conditions for the agent property. Save each `InputCondition` object in the input-conditions array (`allInputConditions` member variable) of the complete-condition object. For more information about input conditions, see "Defining input conditions."

3. Instantiate the appropriate `DependentCondition` objects to describe dependent conditions for the agent property. Save each `DependentCondition` object in the dependent-conditions array (`allDependentConditions` member variable) of the complete-condition object. For more information, see "Defining dependent conditions" on page 147.

4. Save the complete-condition object in the agent property's condition array. The `allDependencies` member variable of the agent-property object contains this condition array.

Defining input conditions

The `InputCondition` class represents an *input condition*, which describes a condition on the current agent property's value. When an input condition evaluates to true, the associated dependent conditions are applied to the dependent agent property. Table 55 shows the information needed to define an input condition.

Table 55. Information for an input condition

Input-condition information	Description	InputCondition member variable
Operator	The kind of comparison to make on the agent-property value. A comparison is indicated as a relational operator and is specified as one of the operator constants in the `CompleteCondition` class.	`operatorType`
Specific value	The value with which to compare the agent property's value. This value can be a constant or the name of another agent property.	`specificValue, typeOfSpecificValue`
Whether the comparison of the agent property's value is performed dynamically	A boolean value to indicate whether to compare the current agent property's value with another property's value dynamically. Comparisons that involve constants do not require dynamic comparisons.	`isDynamic`

To create an input condition, use one of the forms of the `InputCondition()` constructor. The code fragment in Figure 74 on page 149 creates input conditions that compare the agent-property value with two constant string values, "optionA" and "optionB". You can also compare the agent-property value with some other property's value. The code fragment in Figure 72 creates an input condition to compare an agent property's value with the value currently in the `Property2` agent property.

```
// Instantiate a complete-condition object
condition1 = new CompleteCondition();

// Input condition to compare property value with Property2's value
condition1.allInputConditions[0] = new InputCondition(
   CompleteCondition.OP_NOT_EQUAL, true, AgentProperty.TYPE_INTEGER, "Property2");
```

Figure 72. Input condition to compare a property value with another property's value

In Figure 72, the `isDynamic` member variable is set to `true` so that Business Object Wizard knows to first obtain the current value of the `Property2` property *before* comparing the user-specified value with this value. In addition, the `specificValue` is set to "Property2", the name of the property against which the comparison is made. As a result of this input condition, the dependent conditions for the property apply *only* if this property's value *is not the same* as `Property2`'s value.

Defining dependent conditions

The `DependentCondition` class represents a *dependent condition*, which describes a restriction on the value a particular dependent property. A dependent property is a property whose value in some way depends on the current property's value. When the associated input condition (or conditions) evaluates to true, the dependent property's value must meet the restriction that the dependent condition specifies. Table 56 shows the information needed to define a dependent condition.

Table 56. Information for a dependent condition

Dependent-condition information	Description	DependentCondition member variable
Name	The name of the dependent property to which the dependent condition applies if the associated input condition (or conditions) evaluates to true.	`propertyName`
Operator	The kind of comparison to make on the dependent-property value. A comparison is indicated as a relational operator and is specified as one of the operator constants in the `CompleteCondition` class.	`operatorType`
Specific value	The value with which to compare the dependent-property value. This value can be a constant or the name of another agent property.	`specificValue`, `typeOfSpecificValue`
Whether the comparison of the user-specified value is performed dynamically	A boolean value to indicate whether to compare the dependent property's value with another property's value dynamically. Comparisons that involve constants do not require dynamic comparisons.	`isDynamic`

To create a dependent condition, use one of the forms of the `DependentCondition()` constructor. The code fragment in Figure 74 on page 149 creates the following dependent conditions:

- Four dependent conditions for the "`optionA`" input condition specify four possible values for the `DepProperty1` dependent property when the current property has a value of "`optionA`".
- Two dependent conditions for the "`optionB`" input condition specify a range of possible values for the `DepProperty2` dependent property when the current property has a value of "`optionB`".

You can also compare the dependent-property value with some other property's value. The code fragment in Figure 73 creates a dependent condition to compare a dependent property's value with the value currently in the `Property2` agent property.

```
// Dependent condition to compare property value with Property2's value
condition1.allDependentConditions[0] = new DependentCondition(
    CompleteCondition.OP_EQUAL, true, AgentProperty.TYPE_INTEGER, "Property2");
```

Figure 73. Dependent condition to compare a property value with another property's value

In Figure 73, the `isDynamic` member variable is set to `true` so that Business Object Wizard knows to first obtain the current value of the `Property2` property before comparing the dependent property's value with this value. In addition, the `specificValue` is set to "`Property2`", the name of the property against which the comparison is made.

Defining a sample condition

Suppose that you want to define conditions on an agent property (`Property1`) that specifies restrictions on two dependent properties, based on values of `Property1`, as follows:

- The first condition restricts the value of the DepProperty1 dependent property to one of four integer values (0, 1, 256, or 512) if Property1 has the value of "optionA".
- The second condition restricts the value of the DepProperty2 dependent property to be in the range from 1 to 5 (inclusive) if Property1 has the value of "optionB".

Figure 74 shows the code that implements these two conditions.

```
// 1. Instantiate the complete-condition object
condition1 = new CompleteCondition();

// 2. Create the condition on the "optionA" value
//    a) Instantiate the input condition on "optionA"
    condition1.allInputConditions[0] = new InputCondition(
        CompleteCondition.OP_EQUAL, false, AgentProperty.TYPE_STRING,
        "optionA");
//    b) Instantiate the dependent conditions for DepProperty1
    condition1.allDependentConditions[0] = new DependentCondition(
        "DepProperty1", CompleteCondition.OP_EQUAL, false,
        AgentProperty.TYPE_INTEGER, "0");

    condition1.allDependentConditions[1] = new DependentCondition(
        "DepProperty1", CompleteCondition.OP_EQUAL, false,
        AgentProperty.TYPE_INTEGER, "1");

    condition1.allDependentConditions[2] = new DependentCondition(
        "DepProperty1", CompleteCondition.OP_EQUAL, false,
        AgentProperty.TYPE_INTEGER, "256");

    condition1.allDependentConditions[3] = new DependentCondition(
        "DepProperty1", CompleteCondition.OP_EQUAL, false,
        AgentProperty.TYPE_INTEGER, "512");

// 3. Instantiate the next complete-condition object
condition2 = new CompleteCondition();

// 4. Create the condition on the "optionB" value
//    a) Instantiate the input condition on "optionB"
    condition2.allInputConditions[0] = new InputCondition(
        CompleteCondition.OP_EQUAL, false, AgentProperty.TYPE_STRING,
        "optionB");
//    b) Instantiate the dependent conditions for DepProperty2
    condition2.allDependentConditions[0] = new DependentCondition(
        "DepProperty2", CompleteCondition.OP_GREATER_THAN_EQUAL, false,
        AgentProperty.TYPE_INTEGER, "1");

    condition2.allDependentConditions[1] = new DependentCondition(
        "DepProperty2", CompleteCondition.OP_LESS_THAN_EQUAL, false,
        AgentProperty.TYPE_INTEGER, "5");

// Save conditions in the agent-property object
agentProp.allDependencies[0] = condition1;
agentProp.allDependencies[1] = condition2;
```

Figure 74. Defining two agent-property conditions

Shutting down the ODA

After the ODA generates the appropriate content, Business Object Wizard displays Step 6, Save Business Objects dialog box (Step 6). This dialog box allows users to specify how to save the generated content. As part of this step, Business Object Wizard terminates the ODA. The ODA runtime calls the terminate() method to perform clean-up tasks and to release resources for the ODA. For example, if your ODA has connected to a data source in its init() method, it should disconnect from this source in its terminate() method. In the ODK API, the terminate()

method for an ODA is part of the low-level ODA base class, `ODKAgentBase`. It is inherited by the ODA base class, `ODKAgentBase2`, and in turn by your ODA class.

Figure 75 shows a sample `terminate()` method for an ODA that closes a database connection and performs clean-up on objects that accessed the database.

```
public void terminate()
{
   specList = null;

   //close connection
   if(db != null)
      db.disconnect();

   if(dbAnalizer != null)
      dbAnalizer.cleanup();
 }
```

Figure 75. A sample ODA terminate() method

Handling trace and error messages

A *message* is a string of information that the ODA can send to an external ODA log, where it can be reviewed by the system administrator or the developer to provide information about the run-time state of the ODA. There are two different categories of messages that an ODA can send to the ODA log:

- Error or informational messages
- Trace messages

Messages can be generated within the ODA code or obtained from a message file. The ODK API provides the `trace()` method, defined in the `ODKUtility` class, to log trace and error messages. This section provides the following information:

- "Indicating a log destination"
- "Sending a message to the trace file"
- "Message files" on page 153

Indicating a log destination

An ODA sends its messages into its log destination. The *log* is an external destination that is available for viewing by those needing to review the start state of the ODA. The log destination is defined at ODA configuration time by the configuration property `TraceFileName` as the absolute path name of an external file, which must reside on the same machine as the ODA's process.

Note: Because the ODK API provides one method to log both trace and error messages, an ODA has only one file to hold both these kinds of messages. Therefore, although this file is called a *trace file*, it also contains any error messages that the ODA generates.

For information on the format of the trace-file name, see "Specifying a trace file" on page 76.

Sending a message to the trace file

The ODK API provides the `trace()` method, defined in the `ODKUtility` class, to log trace and error messages. The type of message that the ODA tracing mechanism sends depends on the message's trace level, as follows:

Table 57. Trace level and message type

Message trace level	Description
zero (0)	The ODA tracing mechanism allows you to log *error messages* to the trace file.
Any level between 1 and 5	The ODA tracing mechanism allows you to log *trace messages* to the trace file. Trace messages are for information such as status messages, property values, and business object names.

Note: The ODA runtime handles the ODA tracing mechanism implicitly. This mechanism does *not* take effect until the trace file is set in the Configure Agent dialog box of Business Object Wizard. For more information, see "Starting the ODA" on page 101.

In the call to `trace()`, you specify the trace level as an argument. The ODK API provides the trace-level constants for this purpose. For more information on how to generate a message, see "Generating a message string" on page 155. For information on setting the trace level, see "Specifying the trace file and trace level" on page 75.

The ODA tracing mechanism generates files in the same format as those in the Connector Development Kit and InterChange Server Express.

Error and informational messages

When the trace level is zero (0), an ODA can send information about its state to a log destination. Creating a record of errors and status is often called *logging*. The following types of information are recommended for logging:

- Errors and fatal errors from your code to a log file.
- Warnings require a system administrator's attention, from your code to a log file.
- Informational messages such as:
  - ODA startup and termination messages
  - Important messages from the application

Although an ODA can send informational or error messages, this logging process is referred to as *error logging*.

Important: It is recommended that for every exception, you both throw the exception so that it displays in Business Object Wizard, and write an error message that describes the exception to the trace file. By logging all exceptions to the trace file, you can still locate them should the ODA or Business Object Designer Express fail.

Error logging is turned on when the trace level is zero (0). By default, logging on an ODA is turned off because the default trace level is 5. You set the trace level with the `TraceLevel` ODA configuration property. You can set `TraceLevel` to a value of 0 to indicate that a message is an error message.

To send an error message to the log, use the `trace()` method. Table 58 summarizes the trace information for `trace()` to send an error message.

Table 58. Trace information for error messages

Trace information	Description	ODKConstant constant
Trace level	0	TRACELEVEL0
Message type	Errors	XRD_FATAL, XRD_ERROR
	Warnings	XRD_URGENTWARNING, XRD_WARNING
	Informational	XRD_INFO

In addition to the information in Table 58, the `trace()` method also requires the content of the error message. You can obtain the message content as message text in one of the following ways:

- A message string (a `String` value)

```
Util.trace(ODKConstant.TRACELEVEL0, ODKConstant.XRD_ERROR,
    "Invalid property value");
```

You can also retrieve the message string from an exception with the `getMsg()` method of the `ODKException` class, as follows:

```
try

    {
    boDef.setAttributeList(Attributes);
    boDef.setVerbList(Verbs);
    defList[i] = boDef;
    }
catch(BusObjInvalidAttrException e)

    {
    Util.trace(ODKConstant.TRACELEVEL0,
        ODKConstant.XRD_ERROR, e.getMsg());
    }
```

- A message that is retrieved from a message file

```
Util.trace(ODKConstant.TRACELEVEL0, ODKConstant.XRD_WARNING, 1009);
```

For more information on the use of message files, see "Message files" on page 153.

Trace messages

Tracing is an optional troubleshooting and debugging feature that can be turned on for ODAs. When tracing is turned on, system administrators can follow generation of content as the ODA performs its tasks. Tracing allows you and other users of your ODA code to monitor the behavior of the ODA. Tracing can also track when specific ODA methods are called.

Tracing is turned on when the trace level is between 1 and 5. By default, tracing on an ODA is turned on because the default trace level is 5. You set the trace level with the `TraceLevel` ODA configuration property. You can set `TraceLevel` to a value from 1 to 5 to obtain the appropriate level of detail. Level 5 tracing logs the trace messages of *all* lower trace levels. You are responsible for defining what kind of information your ODA returns at each trace level. Table 15 on page 76 shows the recommended content for ODA trace messages. For more information, see "Setting the trace level" on page 76.

To send a trace message to the trace file, use the `trace()` method. Table 58 summarizes the trace information for to the `trace()` to send a trace message.

Table 59. Trace information for trace messages

Trace information	Description	ODKConstant constant
Trace level	1	TRACELEVEL1
	2	TRACELEVEL2
	3	TRACELEVEL3
	4	TRACELEVEL4
	5	TRACELEVEL5
Message type	Trace	XRD_TRACE

In addition to the information in Table 59, the `trace()` method also requires the content of the trace message. You can obtain the message content in one of the following ways:

- As message text:
  - A message string (a `String` value)

    ```
    Util.trace(ODKConstant.TRACELEVEL1, ODKConstant.XRD_TRACE,
          "Entering method getProperties");
    ```

  - A message that is retrieved from a message file

    ```
    Util.trace(ODKConstant.TRACELEVEL1, ODKConstant.XRD_TRACE, 1009);
    ```

    For more information on the use of message files, see "Message files."

- As a business object definition (a `BusObjDef` object)

  In this case, `trace()` formats the contents of the specified business object definition.

  ```
  BusObjDef boDef = new BusObjDef();
  // code that populates business object definition
  ...
  // write out the business object definition
  ODKUtility.getODKUtility().trace(ODKConstant.TRACELEVEL5,
     ODKConstant.XRD_TRACE, boDef);
  ```

- As an array of agent properties (`AgentProperty` objects)

  In this case, `trace()` formats the list of agent properties, preceding it with a string that you can specify.

  ```
  AgentProperties[] propArray;
  // code that populates agent-property array
  ...
  // write out the agent-property array
  ODKUtility.getODKUtility().trace(ODKConstant.TRACELEVEL2,
     ODKConstant.XRD_TRACE, propArray, "List of configuration properties:");
  ```

Message files

In both an error or trace, message, you can provide the message content as a hardcoded string or as a string retrieved from a *message file*. A message file is a text file that contains message numbers and associated message text. The message text can contain positional parameters for passing run-time data out of your ODA. You can provide a message file by creating a file and defining the messages you need.

This section provides the following information about message files:

Message format

Within a message file, messages have the following format:

```
MessageNum
Message
[EXPL]
Explanation
```

The *MessageNum* is an integer that uniquely identifies the message. This message number must appear on one line. The *Message* text can span multiple lines, which a carriage return terminating each line. The *Explanation* text is a more detailed explanation of the condition that causes the message to occur. Do not insert a blank line after the last line of the explanation text. The number of the next message should appear on the line immediately after the explanation. Edit the message file with any text editor, such as Notepad.

For example, message number 1005 might look like the following:

```
1005
ODA content generation is complete.
[EXPL]
This is a log message that indicates successful completion of the ODA.
```

Messages can contain parameters whose values are replaced at run time by values from the program. These parameters are positional and are indicated in the message by a number in braces. For example the following message has three parameters to specify agent-property names:

```
1003
The agent configuration properties are {1}, {2}, {3}.
[EXPL]
This is a trace message that provides startup properties.
```

For more information on how to provide message parameters, see "Using parameter values" on page 156.

Name and location of a message file

An ODA can obtain its messages from one of two message files:

- An ODA message file is named

  *ODAname*Agent.txt

  where *ODAname* is the name that uniquely identifies the ODA. For more information, see "Naming the ODA" on page 159. Put messages that are specific to your ODA in this message file. For example, if you create an ODA named LegacyApp, name its message file LegacyAppAgent.txt.

  Note: Business Object Wizard automatically includes MessageFile in the list of configuration properties with the message-file name in the form *ODAname*Agent.txt. When configuring the ODA, you can change this message-file name to point to an existing file. The specified message file *must* exist for the ODA to continue running. For information on how to specify the message file, see "Specifying the ODA message file" on page 77.

- The global ODA message file is named useragentmessages.txt.

  If you create messages that are global to *all* Object Discovery Agents, add those messages to the global message file.

Both these message files must be located in the following subdirectory of the product directory:

ProductDir\ODA\messages

Generating a message string

The methods in retrieve a predefined message from a message file.

Table 60. Methods that generate a message string

Message method	Description
getMsg()	Generates a message of the specified severity from a message file.
trace()	Generates a message of the specified severity from a message file and sends it to the trace file.

The message-generation methods in Table 60 are defined in the ODKUtility class. These methods require the following information:

- "Specifying a message number"
- "Specifying a message type"
- "Using parameter values" on page 156

Specifying a message number: The message-generation methods in Table 60 require a message number as an argument. This argument specifies the number of the message to obtain from the message file. As described in "Message format" on page 154, each message in a message file must have a unique integer message number associated with it. These message-generation methods search the message file for the specified message number and extract the associated message text.

These methods search the ODA message files for the message number in the following order:

1. The ODA-specific message file, whose default name is *ODAname*Agent.txt
2. The global ODK message file, useragentmessages.txt

Specifying a message type: The message-generation methods in Table 60 also require a message type as an argument. This argument indicates the *severity* of a message. Table 61 lists the valid message types and their associated message-type constants.

Table 61. Message types

Message-type constant	Severity level	Description
XRD_FATAL	Fatal error	Indicates an error that stops program running
XRD_ERROR	Error	Indicates an error that should be investigated
XRD_URGENTWARNING	Urgent warning	Indicates a condition that probably represents a problem and should probably not be ignored
XRD_WARNING	Warning	Indicates a condition that might represent a problem but that can be ignored
XRD_INFO	Informational	Information message only; no action required
XRD_TRACE	----	Use for trace messages

To specify a message type to associate with a message, use one of the message-type constants in Table 61, as follows:

- For an error message, use a message-type constant that indicates the message severity (in decreasing order of severity): `XRD_FATAL`, `XRD_ERROR`, `XRD_URGENTWARNING`, `XRD_WARNING`, or `XRD_INFO`.
- For a trace message, use the `XRD_TRACE` constant.

Message-type constants are defined in the `ODKConstant` class.

Using parameter values: It is not necessary to write separate messages for each possible situation. Instead, use parameters to represent values that change at run time. The use of parameters allows each message to serve multiple situations and helps to keep the message file small.

A parameter always appears as a number surrounded by curly braces: {*number*}. For each parameter you want to add to the message, insert the number within curly braces into the text of the message, as follows:

```
message text {number} more message text.
```

With the message-generation methods in Table 60 on page 155, you can specify an optional number of values for message parameters. The number of parameter values in the method call must match the number of parameters defined in the message text. The message-generation method must supply a value for each parameter. For example, consider message 1003 again:

```
1003
The agent configuration properties are {1}, {2}, {3}.
[EXPL]
This is a trace message that provides startup properties.
```

In the code that sends this message, the following lines might appear:

```
Vector params = new Vector(3);
for(int i=0; i<3; i++)
    params.add(agtProperties[i].propName);
Util.trace(ODKConstant.TRACELEVEL2, 1003, ODKConstant.XRD_TRACE,
    params);
```

The `trace()` method combines these parameter values with the message text in the message file and forms the message. Before writing the message to the trace file, `trace()` replaces the message parameters with the values of the `params` variable.

Message 1003 might appear in the trace file as follows:

```
The agent configuration properties are Username, Password, Url.
```

Because the message text uses parameters to specify the specific property types, rather than including them as hard-coded strings, you can use the same message for any set of missing properties.

Maintaining the message file

At a user site, an administrator might set up a procedure for filtering ODA messages and using e-mail or e-mail pager to notify someone who can resolve problems. Because of this, it is important that the error numbers and the meanings associated with the numbers remain the same after the first release of an Object Discovery Agent. You can change the text associated with an error number, but you should not change the meaning of the text or reassign error numbers.

However, if you do change the meanings associated with error numbers, make sure you document the change and notify users of the Object Discovery Agent.

You can change an Object Discovery Agent's message file while the Object Discovery Agent is running. However, the changes do not take effect until the next time the Object Discovery Agent's is started and the message file is read into memory. If InterChange Server Express fails while an Object Discovery Agent is running, the server automatically reads into memory the message files for all Object Discovery Agents that were previously running.

Handling exceptions

The methods of the ODK API can throw exceptions to indicate certain predefined conditions. This section provides the following information about how to handle exceptions in a Java connector:

- "What is an ODK exception?"
- "Exceptions from the ODK API library"

Note: You can also use error logging and message logging to handle error conditions and messages in your connector. For more information, see "Handling trace and error messages" on page 150.

What is an ODK exception?

When a method of the ODK API throws an exception, this *exception object* is of the `ODKException` class or one of its subclasses, which is an extension of the Java `Exception` class. To create an ODK exception, use the `ODKException()` constructor. Table 62 shows the accessor methods that the `ODKException` class provides to obtain information in the exception object.

Table 62. Information in the exception object

Member	Accessor method
Message text	`getMsg()`

Note: For more information on the methods in the `ODKException` class, see Chapter 23, "ODKException class," on page 253.

The `ODKException` class provides some subclasses to indicate specific error conditions, as Table 103 on page 254 shows.

Exceptions from the ODK API library

When you write code for an ODA, you can include Java `try` and `catch` statements to handle specific exceptions thrown by the methods of the ODK API. The reference description for most ODK API methods has a section entitled Exceptions, which lists the exceptions thrown by that method.

Figure 76 shows a code fragment from sample Roman Army ODA (in the `ArmyAgent4` class) that catches the exceptions that the `getClientFile()` method throws.

```
try
   {
   remotefile = ODKUtility.getODKUtility().getClientFile(filePath, this);
   }
catch (IOException ex)              //file was not found
   {
    return null;
   }
//agent doesn't implement IGeneratesBinFiles, so "getClientFile" failed.
catch (UnsupportedContentException ex)
   {                                //We'll return a random Son instance for now.
    return new Son("X" + ("" + new Date().hashCode()).substring(1),
       new Date().hashCode() % 10  + 2);
   }
```

Figure 76. Catching exceptions from getClientFile()

When an ODK API method throws an exception, it does not usually provide
message and status information in the exception object. However, you can choose
to fill the exception object with a message as needed.

Chapter 6. Adding an Object Discovery Agent to the business integration system

For the WebSphere business integration system to be able to access an Object Discovery Agent (ODA) that you have developed, you must take the following steps:

1. Establish the ODA name and its naming conventions.
2. Compile the ODA class into a jar file.
3. Create the ODA's startup script.

Naming the ODA

This chapter provides suggested naming conventions for the files and directories used in ODA development. Naming conventions provide a way to make your ODA's code more easy to locate and identify. Table 63 summarizes the suggested naming conventions for an ODA.

Table 63. Suggested naming conventions for an ODA

ODA name	ODA package and class name	ODA startup script	ODA library file	ODA runtime directory
*srcDataName*ODA	`com.ibm.oda.`*srcDataName.*`*ODAname*	`start_`*ODAname*	*ODAname*`.jar`	`ODA\`*srcDataName*

Each ODA should have a name that uniquely identifies it within the WebSphere business integration system. By convention, an ODA name (*ODAname*) takes the following form:

srcDataName`ODA`

where *srcDataName* is a unique string that identifies the source data that the ODA converts. For example, if an ODA converts HTML objects to business object definitions, its source data is in the HTML format. Therefore, its ODA name is `HTMLODA`. Alternatively, this ODA name can identify the adapter with which the ODA is associated. For example, the ODA that generates business object definitions for the WebSphere Business Integration Adapter for PeopleSoft has an ODA name of `PeopleSoftODA`.

Compiling the ODA

To compile an ODA, take the following steps:

- Use a JDK development environment. For information on how to install the JDK, see "Setting up the development environment" on page 98.
- Ensure that the library files for the Object Discovery Agent Development Kit (ODK) API are in the `lib` subdirectory of the product directory. The main ODK API library file is named:

  `CwODK.jar`

  Additional ODK library files are:

  `xrmi.jar, xerces.jar`

- Compile the ODA source (.java) files into class (.class) files with the Java compiler.

  These files include the source for your ODA class (which is an extension of the `ODAAgentBase2` class) as well as any other classes your ODA uses. For information on naming the ODK class file, see "Extending the ODA base class" on page 100.

- Create the ODA's library file, which is a Java archive (jar) file that contains the compiled Java code.

  By convention, the jar file's name takes the following form:

  `srcDataNameODA.jar`

  where *srcDataName* uniquely identifies the source data (or adapter) for the ODA. For more information about the ODA name, see "Naming the ODA" on page 159.

  For example, for an ODA that works with HTML, its ODA name could be `HTMLODA`. Therefore, you could name its jar file as:

  `HTMLODA.jar`

Starting up a new ODA

To start the ODA, you run an *ODA startup script*. This startup script starts the ODA runtime. This startup script is a batch file that starts the ODA runtime. By convention, a startup script's name takes the following form:

`start_ODAName.bat`

where ODAname is the unique name of the ODA (its source-data name) with the string "ODA" appended. For example, if an ODA has its source data in HTML format, its ODA name could be `HTMLODA`. Therefore, you could name its startup script as follows:

`start_HTMLODA.bat`

Before you can start up an ODA that you have developed, you need to ensure that a startup script exists to support your new ODA. To enable a startup script to start your own ODA, you must take the following steps:

1. Prepare an ODA runtime directory for your ODA.
2. Create the startup script for your ODA. For Windows systems, also create a shortcut for your ODA startup.
3. Set up the startup script as a Windows service (optional).

The following sections describe each of these steps.

Preparing the ODA runtime directory

The *ODA runtime directory* contains the runtime files for your ODA. To prepare the ODA runtime directory, take the following steps:

1. Create an ODA runtime directory for your new ODA under the `ODA` subdirectory of the product directory:

   `ProductDir\ODA\srcDataName`

   By convention, the directory name matches the ODA's source-data name (*srcDataName*). The source-data name is a string that uniquely identifies the source data (or adapter) with which the ODA works. For more information, see "Naming the ODA" on page 159.

2. Move your ODA's library file to this ODA runtime directory.

The ODA's library file is a Java archive (jar) file. You created this jar file when you compiled the ODA. For more information, see "Compiling the ODA" on page 159.

Creating startup scripts

As "System startup files" on page 64 describes, an ODA requires an ODA startup script for it to be able to start. An ODA requires a startup script for the system administrator to start the ODA-runtime process. When the WebSphere Business Integration Adapters Installer installs adapters on a Windows system, it takes the following steps for ODAs:

- Install the `start_ODAname.bat` startup script in the `ODA\srcDataName` subdirectory of the product directory.
- Create menu options for each ODA under the `Programs > IBM WebSphere Business Integration Adapters > Adapters > Object Discovery Agents` menu. Each menu item is a shortcut that invokes the Windows startup script, `start_ODAname.bat`, for each ODA.

To provide the ability to start up your own ODA, you must generate its startup script and provide the shortcuts that invoke this startup script.

Creating the startup script

In this `start_ODAname.bat` file, make sure you take the following steps:

- Set the following variables within the startup script:

Variable name	Value
PATH	Add the path of the ODA's runtime directory to the front of the PATH variable (so that runtime can locate the ODA's JRE): `PATH="%CROSSWORLDS%"\ODA\ODAruntimeDir;%PATH%` where *ODAruntimeDir* is the ODA's runtime directory, which has the form *srcDataName*. For more information, see "Preparing the ODA runtime directory" on page 160.
AGENTNAME	Specify the ODA name for your ODA (*ODAname*), which has the following form: `srcDataNameODA` where *srcDataName* is the name of the source data. For more information, see "Naming the ODA" on page 159.
AGENT	Specify the full path name for your ODA's library file, the jar file that contains the ODA class. This path name has the following form: `"%CROSSWORLDS%"\ODA\ODAruntimeDir\ODAlibrary.jar` where: • *ODAruntimeDir* is the ODA's runtime directory, which has the form *srcDataName*. For more information, see "Preparing the ODA runtime directory" on page 160. • *ODAlibrary* is the ODA's library file, which has the form `ODAname.jar`. For more information, see "Compiling the ODA" on page 159.

Variable name	Value
AGENTCLASS	Specify the name for your ODA package and class, which has the following form: `com.ibm.oda.`*`srcDataName.ODAname`* where: • *srcDataName* is the name of the ODA's source data, in all lowercase. For more information, see "Naming the ODA" on page 159. • *ODAname* is the name of the ODA's class (its extension of the ODA base class, `ODKAgentBase2`)
JCLASSES	Add any ODA-specific jar files to this variable. Jar files are separated with a semicolon (;). At a minimum, this variable should be set to include the following classes: • The ODK library files: `CwODK.jar`, `xrmi.jar`, `xerces.jar` • The ODA library file, which is stored in the AGENT variable (see above)

- Define and set any additional ODA-specific variables that your startup script needs.

  Define variables for information that can change from release to release. You can then set the variable to a value appropriate for this release and then include the variable in the appropriate command line of the startup script. If the information changes in the future, you only have to change the variable's value. You do not have to locate all command lines that use this information.

- Include the appropriate startup parameters on the line that invokes the ODA runtime (the last line of the startup script), including:

  – All required startup parameters: -l and -c

  – Any optional startup parameters that apply to all invocations of your ODA:

    -v: follow this parameter with the version of the ODA

- The line that invokes the ODA runtime should have the following format:

```
"%CROSSWORLDS%\bin\java" -Duser.home="%CROSSWORLDS%" -mx128m
-classpath %JCLASSES% com.crossworlds.ODKInfrastructure.XRmiAgent
-l%AGENTNAME% -c%AGENTCLASS%
```

Note: Make sure that the line to invoke the ODA runtime is all *on one line* in your startup script; that is, no carriage returns should exist at the line breaks shown in the sample startup line.

Creating the shortcut

A shortcut enables an ODA to be started from a menu item within `Programs > IBM WebSphere Business Integration Adapters > Adapters > Object Discovery Agents`. An easy way to create a shortcut to start an ODA running on Windows is to copy an existing ODA's shortcut and edit the shortcut properties to change the connector name or add any other startup parameters.

Part 3. ODK class reference

Chapter 7. Overview of the ODK API

The Object Discovery Agent Development Kit (ODK) Application Programming Interface (API) includes class libraries that you need to use when developing an Object Discovery Agent (ODA). This ODK API contains predefined classes for ODAs. You use these class libraries to derive ODA classes and methods. The ODK API also provides utilities, such as methods to implement tracing and logging services.

IBM provides a Java jar file (Java archive file), `CwODK.jar`, that contains the predefined classes and interfaces of the ODK API. This jar file resides in the `lib` subdirectory of the product directory.

Note: For instructions on building an ODA to run on Windows 2000, see "Compiling the ODA" on page 159.

Classes and interfaces

The classes and interfaces of the ODK API belong to the following package:
`com.crossworlds.ODK`

Table 64 lists the classes and interfaces in the ODK API.

Table 64. Classes and interfaces in the ODK API

Class or interface	Description	Page
AgentMetaData	Represents an agent-property object, which can represent either a startup property or a business-object property	167
AgentProperty	Defines the attribute-type constants	173
BusObjAttr	Represents an attribute within the business object definition	183
BusObjAttrType	Interface that defines the attribute-type constants	
BusObjDef	Represents a business object definition, which describes the business object	199
BusObjVerb	Represents a business object verb, which describes an action or operation that is valid on the business object	211
CompleteCondition		215
ContentMetaData	Represents the content metadata of the ODA, which describes the content that the ODA has generated	219
ContentType	Represents the content type that the ODA supports	223
DependentCondition		227
IGeneratesBinFiles	Interface to implement by the ODA to provide support for the generation of binary files from the source data	231
IGeneratesBoDefs	Interface to implement by the ODA to provide support for the generation of business object definitions from the source data	235

Table 64. Classes and interfaces in the ODK API (continued)

Class or interface	Description	Page
IGeneratesContent	Is the base class for the two content-generation interfaces, IGeneratesBinFiles and IGeneratesBoDefs. It defines the getContentProtocol() method. **Note:** This manual does *not* provide a separate chapter for this interface. For information on getContentProtocol(), see the description of the IGeneratesBinFiles or IGeneratesBoDefs interface	None
InputCondition		241
ODKAgentBase	Is the base class for the ODA base class, ODKAgentBase2. It defines several methods that ODKAgentBase2 inherits. **Note:** This manual does *not* provide a separate chapter for this class. For information, see the description of the ODKAgentBase2 class.	None
ODKAgentBase2	Represents the base class for an ODA. You extend this class to define your ODA class and implement the required methods	245
ODKConstant	Interface that defines constants for use with the ODK API: • outcome-status constants • verb constants	249
ODKException	Represents an exception object for the ODK API	253
ODKUtility	Provides miscellaneous utility methods for use in an ODA; These utility methods fall into the following general categories: • Static methods for generating and logging messages • Static methods for creating business objects • Static methods for obtaining connector configuration properties • Methods for obtaining locale information	255
TreeNode		267

Chapter 8. AgentMetaData class

The Object Discovery Agent Development Kit (ODK) API provides the `AgentMetaData` class to contain the metadata for the Object Discovery Agent (ODA). Member variables of this class represent the ODA metadata. Business Object Wizard can access the ODA's metadata by calling the `getMetaData()` method in the ODA's class.

The `AgentMetaData` class defines the following:
- "Member variables"
- "Methods" on page 169

The `AgentMetaData` class implements the `ODKConstant` interface. Therefore, all constants defined in `ODKConstant` are available to an `AgentMetaData` object. For a list of constants the `ODKConstant` interface defines, see Chapter 22, "ODKConstant interface," on page 249.

Member variables

Table 65 summarizes the member variables of the `AgentMetaData` class.

Table 65. Member variables of the AgentMetaData class.

Member variable	Description	Page
agentVersion	Specifies the version for the ODA.	174
searchableNodes	Determines whether the children of the expandable nodes (in the tree node) can be searched by a user-specified pattern.	174
searchPatternDesc	Specifies the description to display to users to explain valid search pattern criteria.	174
supportedContent	Stores a description of the content protocol that the ODA supports for each of its supported content types.	175

agentVersion

Specifies the version for the ODA.

Type
`public String agentVersion`

Notes
The second form of the `AgentMetaData()` constructor can initialize the `agentVersion` member variable. If you do not initialize `agentVersion`, it defaults to an empty string. An ODA should initialize its ODA version as part of the `getMetaData()` method, which initializes the ODA's metadata.

searchableNodes

Indicates whether the children of the expandable nodes (in a tree node) can be searched by a user-specified search pattern.

Type

```
public boolean searchableNodes
```

Notes

The `searchableNodes` member variable contains a boolean value that determines whether the user is allowed to search the children of an expandable node in the tree node (in the Select Source dialog box of Business Object Wizard):

- If this variable is `true`, Business Object Wizard enables the **Search for items** menu item when the user right-clicks on the name of an expandable node. The user can click this menu item to display the Enter a Search Pattern dialog box. In it, the user can specify a search pattern.

  Business Object Wizard calls the `getTreeNodes()` method to search the parent node, passing in the user-specified search pattern. The `getTreeNodes()` method searches the data source for children whose names match this search pattern, returning only those that do match. Business Object Wizard displays these children to the user when it displays the expanded parent node.

- If this variable is `false`, the **Search for items** menu item is not available when the user right-clicks on the name of an expandable node. In this case, the `getTreeNodes()` method does not need to handle a user-specified search pattern.

The `AgentMetaData()` constructor does *not* initialize the `searchableNodes` member variable. If you do not initialize `searchableNodes`, it defaults to a value of `false`. If the ODA supports the search-pattern feature, it should initialize the `searchableNodes` member variable as part of the `getMetaData()` method in the ODA class. For more information, see "Implementing the search-pattern feature" on page 114.

searchPatternDesc

Specifies the description to display to users that explains the valid search pattern criteria.

Type

```
public String searchPatternDesc
```

Notes

The `searchPatternDesc` member variable stores the search-pattern description, which displays on the Enter a Search Pattern dialog box. Business Object Wizard displays this dialog box when the user right-clicks a source node and clicks **Search for items**. This description provides information about semantics that the user should use to specify search criteria; that is, it describes what search criteria the ODA implements. This member variable contains a valid value *only* when the `searchableNodes` member variable is `true`. If the ODA supports the search-pattern feature, it should initialize the `searchPatternDesc` member variable as part of the `getMetaData()` method in the ODA class. For more information, see "Implementing the search-pattern feature" on page 114.

supportedContent

Contains a vector that describes which content protocol the ODA supports for each of its supported content types.

Type

```
public Vector supportedContent
```

Notes

The `supportedContent` member variable stores a Java `java.util.Vector` of `ContentProtocol` objects that describe what generated content the ODA supports. Each `ContentProtocol` object contains the following information:

Content-generation information	Description
Content type	A `ContentType` object, which lists one of the supported content types: • `BusinessObject` • `BinaryFile`
Content protocol	A mask of the content-protocol constants to indicate the content protocols supported for the specified content type: • `CONTENT_PROTOCOL_ONREQUEST` • `CONTENT_PROTOCOL_CALLBACK` Content-protocol constants are defined in the `ODKConstant` interface.

Note: The `ContentProtocol` class is part of the `ODAInfrastructure` package, which contains the classes that the ODA runtime and Business Object Wizard use. This package is *not* surfaced to ODA developers. All access to `ContentProtocol` objects is handled by the ODA runtime or Business Object Wizard. An ODA does *not* access objects of this class directly.

The `AgentMetaData()` constructor initializes the `supportedContent` member variable by querying the ODA object that it receives as an argument. You do *not* have to explicitly initialize this member variable.

Methods

Table 66 summarizes the methods of the `AgentMetaData` class.

Table 66. Member methods of the AgentMetaData class

Member method	Description	Page
AgentMetaData()	Creates an agent-metadata object.	170
toXml()	Copies the specified property into the current AgentProperty object.	171

AgentMetaData()

Creates an agent-metadata object.

Syntax
```
public AgentMetaData(ODKAgentBase2 ODAobject);
public AgentProperty(ODKAgentBase2 ODAobject, String version);
```

Parameters

ODAobject   Is a reference to the ODA object that represents the ODA. The constructor queries this object to initialize the `supportedContent` member variable of the `AgentMetaData` object ("supportedContent" on page 168).

version   Specifies the version of the ODA; the value of this parameter initializes the `agentVersion` member variable of the `AgentMetaData` object ("agentVersion" on page 167).

Return values
A newly instantiated `AgentMetaData` object.

Notes
The `AgentMetaData()` method queries the *ODAobject* ODA for its supported content. This constructor provides the following forms for instantiating a new `AgentMetaData` object:

- The first form defines a new `AgentMetaData` object and only initializes its supported content. This form assumes that the ODA does *not* have a version.
- The second form defines a new `AgentMetaData` object and initializes it with both its supported content and version.

Both of these forms of the constructor use the *ODAobject* reference to query the ODA for its supported content. Using this information, the constructor initialize the `supportedContent` member variable.

Note: The `AgentMetaData()` constructor does *not* initialize the member variables that support the search-pattern feature. For your ODA to support search patterns, you must explicitly initialize the `searchableNodes` and `searchPatternDesc` member variables after the `AgentMetaData` object is instantiated. If you do not initialize `searchableNodes`, it defaults to a value of `false`.

toXml()

Converts the ODA metadata into an XML format.

Syntax

```
public String toXml();
```

Parameters

None.

Return values

A String that contains the XML format for the current `AgentMetaData` object.

Chapter 9. AgentProperty class

The Object Discovery Agent Development Kit (ODK) API provides the `AgentProperty` class to represent an *agent-property object*. Each agent-property object contains information about the properties required for the Object Discovery Agent (ODA), such as:

- *Configuration properties*, which provide values that the ODA needs for initialization.
- *Business-object properties*, which provide additional information that the ODA needs for generation of business object definitions.

The `AgentProperty` class defines the following:
- "Property-type constants"
- "Member variables"
- "Methods" on page 179

Property-type constants

The `AgentProperty` class defines static member variables to represent property-type constants. Table 67 summarizes these property-type constants, which represent valid values for an agent property's data type. All property-type constants are of type integer (`int`).

Table 67. Property-type constants of the AgentProperty Class

Property-type constant	Description
TYPE_BOOLEAN	Indicates that the type of the property is `Boolean`.
TYPE_DOUBLE	Indicates that the type of the property is `Double`.
TYPE_FLOAT	Indicates that the type of the property is `Float`.
TYPE_INTEGER	Indicates that the type of the property is `Integer`.
TYPE_STRING	Indicates that the type of the property is `String`.

Member variables

Table 68 summarizes the member variables of the `AgentProperty` class.

Table 68. Member variables of the AgentProperty class.

Member variable	Description	Page
allDefaultValues	Specifies the default values to display for the agent property.	174
allDependencies	Specifies the conditions that describe the dependencies between this agent property and other dependent properties.	174
allValidValues	Specifies the value values to display for the agent property.	174
allValues	Stores the values that the user selects for the agent property.	175
cardinality	Specifies whether the agent property can hold one or multiple values.	175

Table 68. Member variables of the AgentProperty class. (continued)

Member variable	Description	Page
description	Provides a textual explanation of the agent property and can hold other relevant information.	176
isHidden	Determines whether the value of the agent property must display as encrypted.	176
isMultiple	Determines whether Business Object Wizard provides a mechanism for user entry of multiple values for the agent-property value.	177
isReadOnly	Determines whether a user can specify a value for the agent property or can only view the property value.	177
isRequired	Determines whether a value must always be specified for the agent property.	178
propName	Specifies the name of the agent property.	178
type	Specifies the data type of the agent property.	179

allDefaultValues

Specifies the default values to display for the agent property.

Type
```
public java.lang.Object[] allDefaultValues
```

Notes
The `allDefaultValues` member variable contains an array of default values for the agent property. The number of `Object` elements in this array must correspond to the cardinality of the property, as follows:

* For a single-cardinality property (`ODKConstant.SINGLE_CARD`), the `allDefaultValues` array must contain *only one* element.
* For a multiple-cardinality property (`ODKConstant.MULTI_CARD`), the `allDefaultValues` array can contain *one or more* elements.

For more information, see "Specifying default values" on page 144.

allDependencies

Specifies a list of conditions that describe the dependencies between this agent property and other dependent properties.

Type
```
public CompleteCondition[] allDependencies
```

Notes
The `allDependencies` member variable contains a list of conditions in the condition array, which is an array of `CompleteCondition` objects. Each `CompleteCondition` object contains one condition on the agent property's value. A condition contains input and dependency conditions. For more information, see "Setting conditions on the property value" on page 145.

allValidValues

Specifies the valid values to display for the agent property.

Type

```
public java.lang.Object[] allValidValues
```

Notes

The `allValidValues` member variable contains a list of values with which to initialize the drop-down list of an agent property. From this drop-down list, the user can choose one (single cardinality) or more (multiple cardinality) values for the property.

If `allValidValues` specifies a list of values, Business Object Wizard displays these values in the drop-down list for any agent property whose `isMultiple` member variable is `true`. If `isHidden` is `true` and `allValidValues` is null, Business Object Wizard displays a sub-grid for users to specify values.

Note: If the `isMultiple` member variable is `false`, the `allValidValues` member variable should be `null`.

For more information, see "Choosing the type of display control" on page 142.

allValues

Stores the values that the user provides for the agent property.

Type

```
public java.lang.Object[] allValues
```

Notes

The `allValues` member variable is an output variable; that is, it is populated by Business Object Wizard *after* user entry is complete. It contains the values that the user selects from the Value column in the Configure Agent step of Business Object Wizard. This variable is the *only* member variable that does *not* require initialization before the agent property displays to the user.

The number of values in the `allValues` array is determined by the agent property's cardinality:
- If the agent property has single cardinality (its `cardinality` variable is `ODKConstant.SINGLE_CARD`), the `allValues` array contains one value.
- If the agent property has multiple cardinality (its `cardinality` variable is `ODKConstant.MULTI_CARD`), the `allValues` array contains multiple values, one for each value the user has specified.

cardinality

Specifies whether the agent property can hold one or multiple values.

Type

```
public java.lang.String cardinality
```

Notes

The `cardinality` member variable determines whether an agent property's value consists of one value or multiple values. Therefore, it determines how many values the user can specify for the property.

Cardinality	Number of agent-property values the user can specify	Value of cardinality member variable
Single	One	ODKConstant.SINGLE_CARD
Multiple	Many	ODKConstant.MULTIPLE_CARD

The property's cardinality has an effect on the type of control that Business Object Wizard displays for the property. For more information, see "Choosing the type of display control" on page 142.

To initialize an agent property's cardinality, the following call to the third form of the AgentProperty() constructor specifies a string description value as the sixth argument:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING,
        "User Id for logging into the database", true, false,
        ODKConstant.SINGLE_CARD, null, null);
```

Note: You can also specify a value for the agent property's cardinality with the second form of the AgentProperty() constructor, using its eighth argument.

description

Provides a textual explanation of the agent property and may hold other relevant information.

Type
```
public java.lang.String description;
```

Notes
The description member variable displays in the Description column in the Configure Agent step of Business Object Wizard. To initialize an agent property's description, the following call to the third form of the AgentProperty() constructor specifies a string description value as the third argument:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING,
        "User Id for logging into the database", true, false,
        ODKConstant.SINGLE_CARD, null, null);
```

Note: You can also specify a value for the agent property's description with the second form of the AgentProperty() constructor, using its sixth argument.

isHidden

Determines whether the value of the agent property should display as encrypted.

Type
```
public boolean isHidden;
```

Notes
The isHidden member variable is a boolean value that determines whether an agent property's value displays in Business Object Wizard. If isHidden is true, the agent property's value is encrypted when it displays; that is, the value appears as a

string of asterisk (*) characters. To indicate whether an agent property's value is encrypted, specify a boolean value as the fourth argument in the second form of the `AgentProperty()` constructor:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING, true, false, true,
        "User Id for logging into the database", true,
        ODKConstant.SINGLE_CARD, null, null);
```

isMultiple

Determines whether Business Object Wizard provides a means to enter multiple values for an agent property.

Type

```
public boolean isMultiple;
```

Notes

The `isMultiple` member variable is a boolean value that determines whether Business Object Wizard should provide a mechanism for allowing user entry of multiple values for an agent property:

- If `isMultiple` is `true`, Business Object Wizard displays a drop-down list with the list of values that the `allValidValues` member variable contains. From this list, the user clicks on the value to assign to the agent property. The value of the `cardinality` member variable determines how many of these values the user can choose from the drop-down list. If no `allValidValues` array is provided, Business Object Wizard provides a sub-grid of rows for the user to enter each value.
- If `isMultiple` is `false`, Business Object Wizard does *not* allow user entry of multiple values. Instead, it displays an empty field or the default value (if one is specified). In this field, the user enters the agent-property value. The value of the cardinality member variable should be `ODKConstant.SINGLE_CARD`.

Note: For more information, see "Choosing the type of display control" on page 142.

To initialize an agent property with a list of multiple values for the user to choose from, the following call to the third form of the `AgentProperty()` constructor specifies a boolean value of `true` as the fourth argument (the value of the `isMultiple` variable):

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING,
        "User Id for logging into the database", true, true,
        ODKConstant.SINGLE_CARD, null, null);
```

Note: You can also specify a value for `isMultiple` with the second form of the `AgentProperty()` constructor, using its seventh argument.

isReadOnly

Determines whether the user can specify a value in the agent property or can only view the value.

Type

```
public boolean isReadOnly;
```

Notes

The `isReadOnly` member variable is a boolean value that determines whether an agent property's value can be modified by the user when the property displays in Business Object Wizard. To indicate whether an agent property's value is required, specify a boolean value as the fifth argument in the second form of the `AgentProperty()` constructor:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING, true, false, true,
        "User Id for logging into the database", true,
        ODKConstant.SINGLE_CARD, null, null);
```

isRequired

Determines whether a value is required for the agent property.

Type

```
public boolean isRequired;
```

Notes

The `isRequired` member variable is a boolean value that determines whether a value must always be specified for the agent property or whether the user can leave the property's value empty. If `isRequired` is `true`, the user must provide a value for this property. To indicate that an agent property's value is required, the following call to the third form of the `AgentProperty()` constructor specifies a boolean value of `true` as the fourth argument:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING,
        "User Id for logging into the database", true, false,
        ODKConstant.SINGLE_CARD, null, null);
```

Note: You can also specify a value for `isRequired` with the second form of the `AgentProperty()` constructor, using its third argument.

propName

Specifies the name of the agent property.

Type

```
public java.lang.String propName;
```

Notes

The `propName` member variable contains a string with the name of the agent property—for example: Username, Password, DatabaseUrl. The value of the `propName` member variable displays in the Property column in the Configure Agent step of Business Object Wizard. To initialize an agent property's name, the following call to the third form of the `AgentProperty()` constructor specifies a name as the first argument:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING,
        "User Id for logging into the database", true, false,
        ODKConstant.SINGLE_CARD, null, null);
```

Note: All forms of the `AgentProperty()` constructor require that you specify a property name to initialize the `propName` member variable.

type

Specifies the type of the agent property.

Type

```
public int type;
```

Notes

The `type` member variable contains an integer value that represents the data type of the agent property. Table 67 on page 173 lists the property-type constants to use to represent valid property types. A string representation of the `type` member variable's value displays in the Type column in the Configure Agent step of Business Object Wizard. To initialize an agent property's data type, specify a property-type constant as the second argument in the `AgentProperty()` constructor:

```
AgentProperty agt = new AgentProperty("Username",
        AgentProperty.TYPE_STRING,
        "User Id for logging into the database", true, false,
        ODKConstant.SINGLE_CARD, null, null);
```

Methods

Table 69 summarizes the methods of the `AgentProperty` class.

Table 69. Member methods of the AgentProperty class

Member method	Description	Page
AgentProperty()	Creates an agent-property object.	179
copy()	Copies the specified property into the current AgentProperty object.	180

AgentProperty()

Creates an agent-property object.

Syntax

```
public AgentProperty(String name);

public AgentProperty(String name, int type, boolean isReqd, boolean isHid,
    boolean isRdOnly, String desc, boolean isMult, String cardinality,
    Object[] validValues, Object[] defaultValues);

public AgentProperty(String name, int type, String desc, boolean isReqd,
    boolean isMult, String cardinality, Object[] validValues,
    Object[] defaultValues);
```

Parameters

cardinality	Specifies whether the property can hold multiple values; the value of this parameter initializes the `cardinality` member variable of the agent-property object ("cardinality" on page 175).
defaultValues	Specifies default values for the property; the value of this parameter initializes the `allDefaultValues` member variable of the agent-property object ("allDefaultValues" on page 174).
desc	Provides a description of the property; the value of this parameter initializes the `description` member variable of the agent-property object ("description" on page 176).
isHid	Specifies whether the value of the property must be encrypted; the

	value of this parameter initializes the `isHidden` member variable of the agent-property object ("isHidden" on page 176).
isMult	Specifies whether the property can provide multiple values from which the user can choose; the value of this parameter initializes the `isMultiple` member variable of the agent-property object ("isMultiple" on page 177).
isRdOnly	Specifies whether a user can enter or can only view the value for the property; the value of this parameter initializes the `isReadOnly` member variable of the agent-property object ("isReadOnly" on page 177).
isReqd	Specifies whether a value is required for the property; the value of this parameter initializes the `isRequired` member variable of the agent-property object ("isRequired" on page 178).
name	Specifies the name of the property; the value of this parameter initializes the `propName` member variable of the agent-property object ("propName" on page 178).
type	Specifies the type of the property; the value of this parameter initializes the `type` member variable of the agent-property object ("type" on page 179).
validValues	Specifies the valid values for the property; the value of this parameter initializes the `allValidValues` member variable of the agent-property object ("allValidValues" on page 174).

Return values

A newly instantiated `AgentProperty` object.

Exceptions

`IllegalArgumentException`
> Thrown if the value of the *name* parameter is `null` or if the *type* parameter is not a valid property-type constant (see Table 67 on page 173).

Notes

The `AgentProperty()` method provides the following forms for instantiating a new agent-property object:

- The first form defines a new agent-property object and initializes it with *only* a property name. The type of this agent property defaults to `String`. The property is a single-cardinality property that does *not* display multiple values to the user.
- The second form defines a new agent-property object and initializes it with *all* member variables. You can customize the property's metadata by specifying the appropriate values for its member variables.
- The third form defines a new agent-property object and initializes it with all member variables *except* `isHidden` and `isReadOnly`. In this case, the `isHidden` and `isReadOnly` variables default to `false`.

copy()

Copies the specified property into the current AgentProperty object.

Syntax

```
public void copy(AgentProperty prop);
```

Parameters

prop   Specifies the name of the property to be copied.

Chapter 10. BusObjAttr class

The Object Discovery Agent Development Kit (ODK) API provides the `BusObjAttr` class to represent the attributes in a business object definition. A `BusObjAttr` instance represents an *attribute object*. This class defines the following:

- "Attribute constants"
- "Methods"

Note: A business object definition (`BusObjDef` object) automatically defines an attribute object for the `ObjectEventId` attribute. This attribute is automatically marked with the `BusObjAttr.OBJECT_EVENT_ID` constant to indicate its special purpose.

Attribute constants

The `BusObjAttr` class defines static member variables to represent attribute constants. Table 70 summarizes the attribute constants. All attribute constants are of type integer (`int`).

Table 70. Attribute constants of the BusObjAttr class.

Attribute constant	Description
Cardinality constants	
CARD_MULTIPLE	Indicates that the attribute represents an array of child business objects; that is, the attribute has multiple cardinality.
CARD_SINGLE	Indicates that the attribute represents one value or one child business object; that is, the attribute has single cardinality.
ObjectEventId constant	
OBJECT_EVENT_ID	Indicates that the attribute is the ObjectEventId.

Methods

Table 71 summarizes the member methods of the `BusObjAttr` class.

Table 71. Member methods of the BusObjAttr class

Member method	Description	Page
BusObjAttr()	Creates a business-object-attribute object.	185
getAppText()	Retrieves the application-specific information of an attribute.	185
getAttrType()	Retrieves the type of a simple attribute.	186
getAttrTypeName()	Retrieves the type of the child business object as the type of an attribute, for an attribute that represents a child business object or an array of child business objects.	186
getBOVersion()	Retrieves the version number of the business object definition, for an attribute that represents a child business object or an array of child business objects.	187

Table 71. Member methods of the BusObjAttr class (continued)

Member method	Description	Page
getCardinality()	Retrieves the cardinality of the attribute, for an attribute that represents a child business object or an array of child business objects.	187
getComments()	Retrieves the comments associated with the attribute.	188
getDefault()	Retrieves the default value for an attribute.	188
getMaxLength()	Retrieves the maximum length for this attribute.	188
getName()	Retrieves the name of an attribute.	188
getRelationType()	Retrieves the attribute's relationship type, which is containment for an attribute that represents a child business object or an array of child business objects.	189
isForeignKey()	Determines whether this attribute is part of the business object's foreign key.	189
isKey()	Determines whether this attribute is part of the business object's key.	189
isRequiredKey()	Determines whether this attribute is part of the business object's required key.	190
isRequiredServerBound()	Determines whether an attribute is required when the business object represents a triggering event.	190
isSimpleType()	Determines whether an attribute is of a simple type (such as String, Integer, or Float) or whether it represents a child business object or an array of child business objects.	190
setAppText()	Sets the application-specific information of an attribute.	191
setAttrType()	Sets the type of the attribute.	191
setBOVersion()	Sets the version of the child business object or objects that is represented by an attribute, for an attribute that represents a child business object or an array of child business objects.	192
setCardinality()	Sets the cardinality of the attribute, for an attribute that represents a child business object or an array of child business objects.	192
setComments()	Sets the comments associated with the attribute.	193
setDefault()	Sets the default value for an attribute.	193
setIsForeignKey()	Sets the attribute to a boolean value that indicates whether the attribute is part of a foreign key.	193
setIsKey()	Sets the attribute to a boolean value that indicates whether the attribute is part of a key.	193
setIsRequiredKey()	Sets the attribute to a boolean value that indicates whether the attribute is part of the business object's required key.	194
setMaxLength()	Sets the maximum length for an attribute.	194
setName()	Sets the name of an attribute.	194
setRelationType()	Sets the relationship type of an attribute to containment, for an attribute that represents a child business object or an array of child business objects.	195

BusObjAttr()

Creates a new business-object-attribute object.

Syntax

```
public BusObjAttr(String name, int type);

public BusObjAttr(String name, int type, String typeName);

public BusObjAttr(String name, int type, String typeName, boolean isKey,
    boolean isForeignKey, boolean isReqd, String appSpecInfo, int maxLen
    String defaultValue, String BOversion, String cardinality, String relType,
    boolean isReqdServerBound, String comments);
```

Parameters

appSpecInfo	Specifies the application-specific information for the attribute.
BOversion	Specifies the version of the child business object or objects, for an attribute that represents a child business object or an array of child business objects.
cardinality	Specifies the cardinality of the attribute, for an attribute that represents a child business object or an array of child business objects.
comments	Specifies the optional comments to associate with the attribute.
defaultValue	Specifies a default value for the attribute.
isForeignKey	Specifies whether the attribute is part of the business object's foreign key.
isKey	Specifies whether the attribute is part of the business object's key.
isReqd	Specifies whether a value is required for the attribute.
isReqdServerBound	
	Specifies whether a value is required for the attribute when the business object represents a triggering event.
maxLen	Specifies the maximum length of the attribute's value.
name	Specifies the name of the attribute.
relType	Specifies that the relationship type is containment, for an attribute that represents a child business object or an array of child business objects.
type	Specifies the type of the attribute.
typeName	Specifies type of the child business object as the type of the attribute, for an attribute that represents a child business object or an array of child business objects.

Return values

The newly instantiated `BusObjAttr` object.

getAppText()

Retrieves the application-specific information of an attribute.

Syntax

```
public String getAppText();
```

Parameters

None.

Return values

A `String` that contains the application-specific information of an attribute.

See also

setAppText()

getAttrType()

Retrieves the type of an attribute.

Syntax

```
public int getAttrType();
```

Parameters

None.

Return values

An integer that represents the type of the attribute. Compare this integer value with the one of the attribute-type constants:

`BusObjAttrType.BOOLEAN`
> The attribute has the Boolean data type.

`BusObjAttrType.CIPHERTEXT`
> The attribute has the Cipher Text data type.

`BusObjAttrType.DATE`
> The attribute has the Date data type.

`BusObjAttrType.DOUBLE`
> The attribute has the Double data type.

`BusObjAttrType.FLOAT`
> The attribute has the Float data type.

`BusObjAttrType.INTEGER`
> The attribute has the Integer data type.

`BusObjAttrType.INVALID_TYPE`
> The attribute has an invalid data type.

`BusObjAttrType.LONGTEXT`
> The attribute has the Long Text data type.

`BusObjAttrType.OBJECT`
> The attribute has the Object data type (it contains another business object).

`BusObjAttrType.STRING`
> The attribute has the String data type.

See also

getAttrTypeName(), setAttrType()

getAttrTypeName()

Retrieves the name of the attribute's data type.

Syntax

```
public String getAttrTypeName();
```

Parameters

None.

Return values

A `String` that contains the name of the business object definition that is the type of the child business object (when the attribute contains a child business object).

Notes

The `getAttrTypeName()` method retrieves the name of the attribute type for a child business object. When an attribute represents a child business object (or an array of child business objects), its attribute type is `BusObjAttrType.OBJECT` and its attribute type name is the name of the business object definition for the child business object.

See also

getAttrType(), setAttrType()

getBOVersion()

Retrieves the version number of the business object definition, for an attribute that represents a child business object or an array of child business objects.

Syntax

```
public String getBOVersion();
```

Parameters

None.

Return values

A `String` that contains the version number of the child business object definition represented by the attribute.

See also

setBOVersion()

getCardinality()

Retrieves the cardinality of the attribute, for an attribute that represents a child business object or an array of child business objects.

Syntax

```
public String getCardinality();
```

Parameters

None.

Return values

A `String` that contains the cardinality of an attribute that represents a child business object or array of child business objects. Compare this string value with the following cardinality constants:

BusObjAttr.CARD_SINGLE
> The attribute has single cardinality.

BusObjAttr.CARD_MULTIPLE
>The attribute has multiple cardinality.

See also
setCardinality()

getComments()

Retrieves the comments associated with the attribute.

Syntax
`public String getComments();`

Parameters
None.

Return values
A `String` that contains the comments for an attribute.

getDefault()

Retrieves the default value for an attribute.

Syntax
`public String getDefault();`

Parameters
None.

Return values
A `String` that contains the default value for an attribute.

See also
setDefault()

getMaxLength()

Retrieves the maximum length for this attribute.

Syntax
`public int getMaxLength();`

Parameters
None.

Return values
An integer that represents the maximum length of an attribute's value.

See also
setMaxLength()

getName()

Retrieves the name of an attribute.

Syntax

```
public String getName();
```

Parameters

None.

Return values

A String that contains the name of an attribute.

See also

setName()

getRelationType()

Retrieves the attribute's relationship type, which is containment for an attribute that represents a child business object or an array of child business objects.

Syntax

```
public String getRelationType();
```

Parameters

None.

Return values

A String that contains the relationship type ("containment") of an attribute that represents a child business object or an array of child business objects.

See also

setRelationType()

isForeignKey()

Determines whether this attribute is part of the business object's foreign key.

Syntax

```
public boolean isForeignKey();
```

Parameters

None.

Return values

Returns true, if the attribute is a foreign key or part of the foreign key; otherwise, returns false.

See also

setIsForeignKey()

isKey()

Determines whether this attribute is part of the business object's primary key.

Syntax

```
public boolean isKey();
```

Parameters

None.

Return values

Returns `true`, if the attribute is a key or part of the key; otherwise, returns `false`.

See also

setIsKey()

isRequiredKey()

Determines whether this attribute is part of the business object's required key.

Syntax

`public boolean isRequiredKey();`

Parameters

None.

Return values

Returns `true`, if the attribute is a required key or part of a required key; otherwise, returns `false`.

See also

setIsRequiredKey()

isRequiredServerBound()

Determines whether an attribute is required when the business object represents a triggering event.

Syntax

`public boolean isRequiredServerBound();`

Parameters

None.

Return values

Returns `true`, if the attribute is required when the business object represents a collaboration object request; otherwise, returns `false`.

isSimpleType()

Determines whether an attribute is of a simple type (such as String, Integer, or Float) or whether it represents a child business object or an array of child business objects.

Syntax

`public boolean isSimpleType();`

Parameters

None.

Return values

Returns `true`, if the attribute is of a simple type; otherwise, returns `false`.

See also

getAttrType(), setAttrType()

setAppText()

Sets the application-specific information of an attribute.

Syntax
```
public void setAppText(String appInfo);
```

Parameters

appInfo   Is the application-specific information to assign to the attribute.

Return values
None.

See also
getAppText()

setAttrType()

Sets the type of the attribute.

Syntax
```
public void setAttrType(int type);

public void setAttrType(int type, String typeName);
```

Parameters

type   Is the type of the attribute, represented as one of the attribute-type constants:

```
BusObjAttrType.BOOLEAN
BusObjAttrType.CIPHERTEXT
BusObjAttrType.DATE
BusObjAttrType.DOUBLE
BusObjAttrType.FLOAT
BusObjAttrType.INTEGER
BusObjAttrType.LONGTEXT
BusObjAttrType.OBJECT
BusObjAttrType.STRING
```

typeName   Is the name of the business object for an attribute that represents a child business object or array of child business objects; in this case, the type of the attribute is the same as the type of the child business object and the `type` value is `OBJECT`.

Return values
None.

Exceptions

BusObjInvalidAttrException

> Thrown if the *type* is invalid; that is, it is not one of the values represented by the attribute-type constants.

Notes
The `setAttrType()` method provides the following forms:

- The first form allows you to set the attribute type for a simple attribute, specified as an attribute-type constant that is defined in the `BusObjAttrType` class.

- The second form allows you to set the attribute type for a child business object or an array of child business objects. This form allows you to specify the attribute type (as the attribute-type constant `BusObjAttrType.OBJECT`) and the name of the business object definition for the child business object.

See also
getAttrType(), getAttrTypeName()

For related reference information, see Chapter 11, "BusObjAttrType interface," on page 197 and Chapter 23, "ODKException class," on page 253.

setBOVersion()

Sets the version number of the business object definition, for an attribute that represents a child business object or an array of child business objects.

Syntax
```
public void setBOVersion(String version);
```

Parameters

version   Is the version of the business object definition for the child business object or objects that this attribute represents.

Return values
None.

See also
getBOVersion()

setCardinality()

Sets the cardinality of the attribute, for an attribute that represents a child business object or an array of child business objects.

Syntax
```
public void setCardinality(String cardinality);
```

Parameters

cardinality   Is the cardinality to assign to this attribute. Cardinality is represented by one of the following cardinality constants:
```
BusObjAttr.CARD_SINGLE
BusObjAttr.CARD_MULTIPLE
```

Return values
None.

Exceptions

BusObjInvalidAttrException

Thrown if the *cardinality* is not a valid; that is, it does not contain a valid cardinality constant.

See also
getCardinality()

setComments()

Sets the comments associated with an attribute.

Syntax
```
public void setComments(String comment);
```

Parameters

comment   Is the comment string to provide additional information for the attribute.

Return values
None.

See also
getComments()

setDefault()

Sets the default value for an attribute.

Syntax
```
public void setDefault(String defaultValue);
```

Parameters

defaultValue   Is the default value to assign to the attribute.

Return values
None.

See also
getDefault()

setIsForeignKey()

Sets the attribute property that indicates whether the attribute is part of a foreign key.

Syntax
```
public void setIsForeignKey(boolean fKey);
```

Parameters

fKey   Indicates whether this attribute is part of a foreign key.

Return values
None.

See also
isForeignKey()

setIsKey()

Sets an attribute property that indicates whether the attribute is part of a primary key.

Syntax

```
public void setIsKey(boolean key);
```

Parameters

key   Indicates whether this attribute is part of a key.

Return values

None.

See also

isKey()

setIsRequiredKey()

Sets the attribute to a boolean value that indicates whether the attribute is part of the business object's required key.

Syntax

```
public void setIsRequiredKey(boolean isReqd);
```

Parameters

isReqd   Indicates whether this attribute is a required key.

Return values

None.

See also

isRequiredKey()

setMaxLength()

Sets the maximum length for an attribute.

Syntax

```
public void setMaxLength(int maxLength);
```

Parameters

maxLength   Is the maximum length to assign to the attribute.

Return values

None.

Exceptions

BusObjInvalidAttrException

Thrown if the maximum length is `maxLength < 0` or `maxLength > 2^31-1`

See also

getMaxLength()

setName()

Sets the name of an attribute.

Syntax

```
public void setName(String name);
```

Parameters

name   Is the name to assign to the attribute.

Return values

None.

See also

getName()

setRelationType()

Sets the relationship type of an attribute to containment, for an attribute that represents a child business object or an array of child business objects.

Syntax

```
public void setRelationType(String relType);
```

Parameters

relType   Is the relationship type to assign to this attribute.

Return values

None.

See also

getRelationType()

Chapter 11. BusObjAttrType interface

The Object Discovery Agent Development Kit (ODK) API provides the `BusObjAttrType` class to represent the valid data types for attributes in a business object definition. Any class that implements the `BusObjAttrType` interface can access its defined constants directly. For example, if the `ODKAgentBase2` class implements the `BusObjAttrType` interface, its methods can access the `BOOLEAN` constant as follows:

```
int bool_type = BOOLEAN;
```

The `BusObjAttrType` class defines the following:

- "Attribute-type constants"
- "Static member variable"

Attribute-type constants

The `BusObjAttrType` class defines static member variables to represent attribute-type constants. Table 72 summarizes these attribute-type constants. All property-type constants are of type integer (`int`).

Table 72. Attribute-type constants of the BusObjAttrType class.

Attribute-type constant	Description
BOOLEAN	Represents an attribute type of Boolean.
CIPHERTEXT	Represents an attribute type of CipherText.
DATE	Represents an attribute type of Date.
DOUBLE	Represents an attribute type of Double.
FLOAT	Represents an attribute type of Float.
INTEGER	Represents an attribute type of Integer.
INVALID_TYPE	Represents an invalid attribute type.
LONGTEXT	Represents an attribute type of Long Text.
OBJECT	Represents an attribute of type of Object.
STRING	Represents an attribute type of String.

Static member variable

In addition to the attribute-type constants (which are defined as static member variables), the `BusObjAttrType` class defines the static member variable in Table 73.

Table 73. Static member variable of the BusObjAttrType class.

Static member variable	Description
AttrTypes	A `String` array that contains the names for the different attribute types. This array can be indexed by the attribute type; for example, the following code retrieves the type name for the Integer attribute type: `BusObjAttrType.AttrTypes[BusObjAttrType.INTEGER]`

Chapter 12. BusObjDef class

The Object Discovery Agent Development Kit (ODK) API provides the
`BusObjDef`class to represent a business object definition that the Object Discovery
Agent (ODA) generates. Table 74 summarizes the methods in the `BusObjDef` class.

Table 74. Member methods of the BusObjDef class.

Member method	Description	Page
`BusObjDef()`	Creates a business-object-definition object.	199
`addDefaultVerbs()`	Adds the default verbs (Create, Retrieve, Update, and Delete) to the list of supported verbs.	200
`getAppInfo()`	Retrieves the application-specific information for the business object definition.	200
`getAttrCount()`	Retrieves the number of attributes, including ObjectEventId, in the attribute list of the business object definition.	201
`getAttribute()`	Retrieves the attribute by its name or by its specified position in the business object definition.	201
`getAttributeIndex()`	Retrieves the ordinal position of the attribute in the business object definition, given its attribute name.	202
`getAttributeList()`	Retrieves a vector that contains the list of attributes in the business object definition.	202
`getName()`	Retrieves the name of the business object definition.	203
`getVerb()`	Retrieves the verb object for the specified verb name.	203
`getVerbCount()`	Retrieves the number of verbs in the verb list.	204
`getVerbList()`	Retrieves a vector that contains the list of verbs in the business object definition.	204
`getVersion()`	Retrieves the version of the business object definition.	204
`insertAttribute()`	Inserts the specified attribute in the business object's attribute list.	205
`insertVerb()`	Inserts the specified verb into the business object's verb list.	205
`removeAttribute()`	Removes the attribute at the specified position in the attribute list.	206
`removeVerb()`	Removes the verb with the specified name in the verb list.	207
`setAppInfo()`	Sets the application-specific information for the business object definition.	207
`setAttributeList()`	Sets the list of attributes for the business object definition.	208
`setVerbList()`	Sets the list of verbs for the business object definition.	208

BusObjDef()

Creates a business-object-definition object.

Syntax

```
public BusObjDef(String name);
public BusObjDef(String name, Vector attrList, String[] verbNames,
    String appSpecInfo);
public BusObjDef(String name, Vector attrList, Vector verbList,
    String appSpecInfo);
```

Parameters

appSpecInfo	Specifies the business-object-level application-specific information.
attrList	Specifies a Java `Vector` obtain that contains the business object definition's attribute list.
name	Specifies the name of the business object definition.
verbList	Specifies a vector of the business object's verbs.
verbNames	Specifies a String array of the business object's verb names.

Return values

A newly instantiated `BusObjDef` object.

Exceptions

BusObjInvalidDefException
Definition

BusObjInvalidVerbException
Definition

addDefaultVerbs()

Adds the default verbs (Create, Retrieve, Update, and Delete) to the business object definition's verb list.

Syntax

```
public void addDefaultVerbs();
```

Parameters

None.

Return values

None.

getAppInfo()

Retrieves the application-specific information for the business object definition.

Syntax

```
public String getAppInfo();
```

Parameters

None.

Return values

A `String` that contains the business-object-level application-specific information.

See also

setAppInfo()

---

getAttrCount()

Retrieves the number of attributes in the attribute list of the business object definition.

Syntax

```
public int getAttrCount();
```

Parameters

None.

Return values

The number of attributes in the business object definition (including the ObjectEventId attribute).

---

getAttribute()

Retrieves the attribute by its name or by its specified position in the business object definition's attribute list.

Syntax

```
public BusObjAttr getAttribute(String attrName);
public BusObjAttr getAttribute(int pos);
```

Parameters

attrName	Is the name of the attribute to retrieve from the business object definition's attribute list.
pos	Is an integer that specifies the ordinal position of the attribute in the business object definition's attribute list.

Return values

The attribute (`BusObjAttr`) object for the specified attribute in the business object definition.

Exceptions

BusObjNoSuchAttrException
Thrown if the specified attribute does not exist or the position within the attribute list is not valid.

Notes

The getAttribute() method retrieves an attribute from the business object definition's attribute list. It returns this attribute as an attribute object (BusObjAttr). You can use methods of the BusObjAttr class to obtain information about the attribute.

See also

getAttributeIndex(), getAttributeList()

---

getAttributeIndex()

Retrieves the ordinal position of the attribute in the business object definition, given its attribute name.

Syntax

```
public int getAttributeIndex(String attrName);
```

Parameters

attrName   Is the name of the attribute whose ordinal position is retrieved.

Return values

The integer position of the attribute in the attribute list of the business object definition

Exceptions

BusObjNoSuchAttrException
Thrown if the specified attribute does not exist in the business object definition.

See also

getAttribute()

---

getAttributeList()

Retrieves the list of attributes in the business object definition.

Syntax

```
public Vector getAttributeList();
```

Parameters

None.

Return values

A java.util.Vector object that contains one attribute (BusObjAttr) object for each attribute in the business object definition.

Notes

The getAttributeList() method returns the business object definition's attribute list as a Java Vector of attribute objects. You can use methods of the java.util.Vector class to retrieve attribute objects from this Vector object. You can use methods of the BusObjAttr class to obtain information from the attribute object.

See also

setAttributeList()

getName()

Retrieves the name of the business object definition.

Syntax

public String getName();

Parameters

None.

Return values

A String that contains the name of the business object definition

getVerb()

Retrieves the specified verb from the business object definition's verb list.

Syntax

public BusObjVerb getVerb(String *verb*);

Parameters

verb   Is the name of the verb to retrieve from the business object definition's verb list.

Return values

The verb (BusObjVerb) object for the specified verb in the business object definition's verb list.

Exceptions

BusObjNoSuchVerbException
        Thrown if the specified verb does not exist.

Notes

The getVerb() method retrieves a verb from the business object definition's verb list. It returns this verb as a verb object (BusObjVerb). You can use methods of the BusObjVerb class to obtain information about the verb.

See also

getVerbCount(), getVerbList()

getVerbCount()

Retrieves the number of verbs in the business object definition's verb list.

Syntax

```
public int getVerbCount();
```

Parameters

None.

Return values

The integer number of verbs in the business object definition's verb list.

See also

getVerb()

getVerbList()

Retrieves the list of verbs in the business object definition.

Syntax

```
public Vector getVerbList();
```

Parameters

None.

Return values

A `java.util.Vector` object that contains one verb (`BusObjVerb`) object for each
supported verb in the business object definition.

Notes

The `getVerbList()` method returns the business object definition's verb list as a
Java `Vector` of verb objects. You can use methods of the `java.util.Vector` class to
retrieve verb objects from this `Vector` object. You can use methods of the
`BusObjVerb` class to obtain information from the verb object.

See also

setVerbList()

getVersion()

Retrieves the version of the business object definition.

Syntax

```
public String getVersion();
```

Parameters

None.

Return values

A `String` that contains the version of the business object definition.

---

insertAttribute()

Inserts the specified attribute in the business object definition's attribute list.

Syntax

```
public void insertAttribute(BusObjAttr attrObj);
public void insertAttribute(BusObjAttr attrObj, int pos);
```

Parameters

attrObj	Is the attribute object be added to the attribute list of the business object definition.
pos	Is the ordinal position at which the attribute is to be added to the attribute list.

Return values

None.

Exceptions

BusObjInvalidAttrException
Thrown if the attribute that an attribute object describes is invalid.

Notes

The `insertAttribute()` method provides the following forms:

- The first form specifies the attribute to add by its attribute name. When you use this form, `insertAttribute()` inserts the specified attribute at the position immediately *above* the `ObjectEventId` attribute in the business object's attribute list.
- The second form specifies the attribute to add and the ordinal position within the attribute list at which to add this attribute. When you specify an ordinal position, `insertAttribute()` inserts the specified attribute at the specified *pos* position in the business object definition's attribute list, and moves down by one position every attribute that follows in the list.

Important: If you specify an ordinal position, make sure that the specified position is *above* the `ObjectEventId` attribute.

See also

removeAttribute()

---

insertVerb()

Inserts the specified verb into the business object definition's verb list.

Syntax

```
public void insertVerb(BusObjVerb verbObj);
public void insertVerb(String verbStrng, String appSpecInfo);
```

Parameters

appSpecInfo   Is the application-specific information for the verb to be added to the verb list.

verbObj   Is the verb object to be added to the verb list.

verbStrng   Is the name of the verb to be added to the verb list.

Exceptions

BusObjInvalidVerbException
> Thrown if the verb that the verb object describes is a duplicate.

Notes

The `insertVerb()` method provides the following forms:to insert a verb object into the verb list of the business object definition in either of the following ways:

- The first form specifies the verb to add as an initialized verb object (a `BusObjVerb` instance). You can use methods of the `BusObjVerb` class to initalize the verb object.
- The second form specifies the verb information, including the name and application-specific information for the verb.

See also

removeVerb()

---

removeAttribute()

Removes a specified attribute from the business object definition's attribute list.

Syntax

```
public BusObjAttr removeAttribute(int pos);
public BusObjAttr removeAttribute(String attrName);
```

Parameters

attrName   Is the name of attribute to remove from the business object definition's attribute list.

pos   Is the ordinal position at which to remove the attribute.

Return values

An attribute (`BusObjAttr`) object that contains the removed attribute.

Exceptions

BusObjNoSuchAttrException
> Thrown if the specified attribute does not exist.

BusObjInvalidAttrException
> Thrown if the attribute to be removed is one that cannot be removed, such as the `ObjectEventId` attribute.

Notes

The `removeAttribute()` method provides the following forms:

- The first form specifies the attribute to remove by its ordinal position within the business object definition's attribute list.
- The second form specifies the attribute to remove by its attribute name. and the ordinal position within the attribute list at which to add this attribute.

Important: If you specify an ordinal position, make sure that the specified position is *not* the ObjectEventId attribute.

See also

insertAttribute()

removeVerb()

Removes the specified verb from the business object definition's verb list.

Syntax

```
public BusObjVerb removeVerb(String verb);
```

Parameters

verb　　　　　Is the name of the verb whose verb object is to be removed from the business object definition's verb list.

Return values

A verb (BusObjVerb) object that contains the removed verb.

Exceptions

BusObjNoSuchVerbException
　　　　Thrown if the specified verb does not exist.

See also

insertVerb()

setAppInfo()

Sets the application-specific information for the business object definition.

Syntax

```
public void setAppInfo(String appSpecInfo);
```

Parameters

appSpecInfo　　Is the business-object-level application-specific information.

Return values

None.

See also

getAppInfo()

setAttributeList()

Sets the list of attributes for the business object definition.

Syntax

```
public void setAttributeList(Vector attrList);
```

Parameters

attrList   Is a `java.util.Vector` object that contains attribute objects to store
             in the business object definition's attribute list.

Exceptions

BusObjInvalidAttrException
> Thrown if an attribute object in *attrList* contains an attribute that is
> duplicate or is null.

Notes

The `setAttributeList()` method passes the *attrList* attribute list as a Java `Vector` of
attribute objects. You can use methods of the `BusObjAttr` class to store information
in the attribute object. You can use methods of the `java.util.Vector` class to store
attribute objects in this `Vector` object.

See also

getAttributeList()

---

setVerbList()

Sets the list of verbs for the business object definition.

Syntax

```
public void setVerbList(Vector verbList);
```

Parameters

verbList   Is a `java.util.Vector` object that contains verb objects to store in
             the business object definition's verb list.

Return values

None.

Exceptions

BusObjInvalidVerbException
> Thrown if a verb object in *verbList* contains a verb that is duplicate or is
> null.

Notes

The `setVerbList()` method passes the *verbList* verb list as a Java `Vector` of verb
objects. You can use methods of the `BusObjVerb` class to store information in the
verb object. You can use methods of the `java.util.Vector` class to store verb
objects in this `Vector` object.

See also

getVerbList()

Chapter 13. BusObjVerb class

The Object Discovery Agent Development Kit (ODK) API provides the BusObjVerb class to represent the verbs in a business object definition. A BusObjVerb instance represents a *verb object*. Table 75 summarizes the methods of the BusObjVerb class.

Table 75. Member methods of the BusObjVerb class.

Member method	Description	Page
BusObjVerb()	Creates a business-object-verb object.	211
clone()	Clones a verb object.	211
getAppInfo()	Retrieves the application-specific information of the verb.	212
getName()	Retrieves the name of the verb.	212
setAppInfo()	Sets the application-specific information of the verb.	212
setName()	Sets the name of the verb.	213

BusObjVerb()

Creates a business-object-verb object.

Syntax

```
public BusObjVerb(String verb, String appSpecInfo);
```

Parameters

appSpecInfo   Specifies the application-specific information for the verb.

verb   Specifies a verb that is supported by the business object definition.

Return values

The newly instantiated BusObjVerb object.

Exceptions

BusObjInvalidVerbException
    Thrown if the specified verb is not valid.

clone()

Clones a verb object.

Syntax

```
public Object clone();
```

Parameters

None.

Return values

None.

Notes

This `clone()` method overrides the `clone()` method in the `java.lang.Object` class.

---

getAppInfo()

Retrieves the application-specific information of the verb.

Syntax

```
public String getAppInfo();
```

Parameters

None.

Return values

A `String` that contains the application-specific information of the verb

See also

setAppInfo()

---

getName()

Retrieves the name of the verb.

Syntax

```
public String getName();
```

Parameters

None.

Return values

A `String` that contains the name of the verb.

See also

setName()

---

setAppInfo()

Sets the application-specific information of the verb.

Syntax

```
public void setAppInfo(String appSpecInfo);
```

Parameters

appSpecInfo
> Is the verb-level application-specific information to store in the verb object.

Return values

None.

See also

getAppInfo()

---

setName()

Sets the name of the verb.

Syntax

```
public void setName(String verb);
```

Parameters

verb   Is the name of the verb to store in the verb object.

Return values

None.

Exceptions

BusObjInvalidVerbException
Thrown if the specified verb is not valid.

See also

getName()

Chapter 14. CompleteCondition class

The Object Discovery Agent Development Kit (ODK) API provides the `CompleteCondition` class to represent a conditions on the value of an agent property (represented by an `AgentProperty` object). A condition consists of two kinds of subconditions, input conditions and dependent conditions. An agent property stores all its conditions in its `allDependencies` member variable.

Note: For information on input conditions, see Chapter 20, "InputCondition class," on page 241. For information on dependent conditions, see Chapter 17, "DependentCondition class," on page 227.

The `CompleteCondition` class defines the following:
- "Operator constants"
- "Member variables"
- "Methods" on page 216

Operator constants

The `CompleteCondition` class defines static member variables to represent operator constants. Table 76 summarizes these operator constants, which represent valid operators to use in conditions. All operator constants are of type `String`.

Table 76. Operator constants of the CompleteCondition class.

Operator constant	Description
OP_EQUAL	Contains a `String` that represents the Equals (=) operator.
OP_EXISTS	Contains a `String` that represents the Exists operator.
OP_GREATER_THAN	Contains a `String` that represents the Greater Than (>) operator.
OP_GREATER_THAN_EQUAL	Contains a `String` that represents the Greater Than or Equal To (>=) operator.
OP_LESS_THAN	Contains a `String` that represents the Less Than (<) operator.
OP_LESS_THAN_EQUAL	Contains a `String` that represents the Less Than or Equal To (<=) operator.
OP_NOT_EQUAL	Contains a `String` that represents the Not Equal (!=) operator.

Member variables

Table 77 summarizes the member variables in the `CompleteCondition` class.

Table 77. Member variables of the CompleteCondition class.

Member variable	Description	Page
allDependentConditions	Specifies all dependent conditions for the property.	216
allInputConditions	Specifies all input conditions for the property.	216

allDependentConditions

Specifies an array of all dependent conditions in the current complete condition.

Type

```
public DependentCondition[] allDependentConditions
```

Notes

The `allDependentConditions` member variable contains a list of dependent conditions in the dependent-condition array, which is an array of `DependentCondition` objects. Each `DependentCondition` object contains one dependent condition, which restricts the value of the dependent property when the associated input conditions evaluate to true. For more information, see "Setting conditions on the property value" on page 145.

allInputConditions

Specifies an array of all input conditions in the current complete condition.

Type

```
public InputCondition[] allInputConditions
```

Notes

The `allInputConditions` member variable contains a list of conditions in the input-condition array, which is an array of `InputCondition` objects. Each `InputCondition` object contains one input condition, which specifies a comparison to make on the current agent property's value. For more information, see "Setting conditions on the property value" on page 145.

Methods

Table 78 summarizes the methods in the `CompleteCondition` class.

Table 78. Member methods of the CompleteCondition class.

Member method	Description	Page
`CompleteCondition()`	Creates a complete-condition object.	216
`copy()`	Copies the current complete condition into the specified complete-condition object.	217

CompleteCondition()

Creates a complete-condition object.

Syntax

```
public CompleteCondition();
public CompleteCondition(InputCondition[] allInputConds,
    DependentCondition[] allDepConds);
```

Parameters

allDepConds Specifies an array of dependent conditions; the value of this parameter initializes the `allDependentConditions` member variable ("allDependentConditions").

allInputConds Specifies an array of input conditions; the value of this parameter initializes the `allInputConditions` member variable ("allInputConditions").

Return values

A newly instantiated `CompleteCondition` object.

copy()

Copies the current complete condition into a specified complete-condition object.

Syntax

```
public void copy(CompleteCondition completeCond);
```

Parameters

completeCond Specifies the name of the complete-condition object into which the current complete condition is copied.

Return values

None.

Chapter 15. ContentMetaData class

The Object Discovery Agent Development Kit (ODK) API provides the
`ContentMetaData` class to contain the metadata for the generated content of the
Object Discovery Agent (ODA). Member variables of this class represent the ODA's
content metadata. When the ODA generates its content, it must return a
content-metadata object to describe the generated content. The method that returns
the content metadata depends on the content protocol that the ODA supports, as
follows:

- If the ODA supports an on-request protocol for a particular content type
  (business object definitions or files), the appropriate content-generation method
  returns the content metadata to Business Object Wizard.

- If the ODA supports a callback protocol (for file content only), a user-defined
  method returns the content metadata to Business Object Wizard through the
  `ODKUtility.contentComplete()` method.

Note: For more information, see "Providing generated content" on page 94.

Business Object Designer Express uses the content-metadata object to obtain
information about the generated content for each of the content types that the
ODA supports. To determine the supported generation protocols, Business Object
Designer Express calls the ODA's `getContentProtocol()` method (from its
`IGeneratesContent` class).

The `ContentMetaData` class defines the following:
- "Member variables"
- "Methods" on page 220

Member variables

Table 79 summarizes the member variables of the `ContentMetaData` class.

Table 79. Member variables of the ContentMetaData class.

Member variable	Description	Page
`contentType`	Indicates the content type for the generated content.	219
`count`	Specifies the total number of content elements in the requested content.	220
`length`	Specifies the total length, in bytes, of the requested content.	220

contentType

Indicates the content type of the generated content.

Type
`public ContentType contentType`

Notes

The `contentType` member variable is a `ContentType` object that indicates the content type of the generated content that this content metadata describes. It must be set to the content type appropriate for the generated content, as Table 80 shows.

Table 80. Content-type values

Content type	Value of contentType member variable
Business object definitions	`ContentType.BusinessObject`
Binary files	`ContentType.BinaryFile`

For example, when an ODA completes content generation, it must return a content-metadata object whose `contentType` member variable corresponds to the type of content generated.

count

Specifies the total number of content elements in the requested content. This count value must be greater than zero (0).

Type
`public long count`

length

Specifies the total size of the requested content, in bytes. If the content's length is unknown, assign a length of zero (0).

> **Important**
> Business Object Wizard does not currently use the `length` member variable. Therefore, this member variable should be initialized to a "null" value, such as zero (0) or -1.

Type
`public long length`

Methods

Table 81 summarizes the methods of the `ContentMetaData` class.

Table 81. Member methods of the ContentMetaData class

Member method	Description	Page
`ContentMetaData()`	Creates a content-metadata object.	221
`badContent()`	Returns a content-metadata object that indicates the ODA is unable to generate the specified content type.	221
`contentNotReady()`	Returns a content-metadata object that indicates the ODA is not yet finished with the content generation.	221
`contentUnavailable()`	Returns a content-metadata object that indicates the ODA is not generating the specified content, even though it implements the corresponding interface.	222

ContentMetaData()

Creates a content-metadata object.

Syntax

```
public ContentMetaData(ContentType contentType, long length, long count);
```

Parameters

contentType   Is a `ContentType` object that indicates the content type of the generated content that the content-metadata object describes; the value of this parameter initializes the `contentType` member variable of the content-metadata object ("contentType" on page 219).

count   Specifies number of content elements in the requested content; the value of this parameter initializes the `count` member variable in the content-metadata object ("count" on page 220).

length   Specifies the total size of the requested content, in bytes; the value of this parameter initializes the `length` member variable in the content-metadata object ("length" on page 220). Business Object Wizard does *not* currently use the length member variable.

Return values

A newly instantiated `ContentMetaData` object.

badContent()

Notifies Business Object Wizard that the content that the ODA has generated is incomplete or in some other way has an error.

Syntax

```
public static ContentMetaData badContent(ContentType contentType);
```

Parameters

contentType   Is the `ContentType` object that identifies the content type of the bad generated content.

Return values

A `ContentMetaData` object that describes the unsuccessfully generated content.

contentNotReady()

Notifies Business Object Wizard that the ODA is not yet finished generating the specified content.

Syntax

```
public static ContentMetaData contentNotReady(ContentType contentType);
```

Parameters

contentType   Is the `ContentType` object that identifies the content type of the incomplete generated content.

Return values

A `ContentMetaData` object that describes the incompletely generated content.

contentUnavailable()

Notifies Business Object Wizard that the ODA does not support generation of the specified content, even though it implements the corresponding interface.

Syntax

```
public static ContentMetaData contentUnavailable(ContentType contentType);
```

Parameters

contentType   Is the `ContentType` object that identifies the content type of the unavailable generated content.

Return values

A `ContentMetaData` object that describes the unavailable generated content.

Notes

The `contentUnavailable()` method indicates that the ODA does not generate content of the *contentType* content type. For example, if an ODA supports *only* a callback content protocol for a particular content type, Business Object Wizard never calls its content-generation method (`generateBoDefs()` for business-object-definition content or `generateBinFiles()` for binary-file content). Therefore, the content-generation method can call `contentUnavailable()` as its return value to Business Object Wizard.

Chapter 16. ContentType class

The Object Discovery Agent Development Kit (ODK) API provides the `ContentType` class to represent the valid content types that an Object Discovery Agent (ODA) can generate. The `ContentType` class defines the following:

- "Member variables"
- "Methods" on page 224

Member variables

Table 82 summarizes the member variables of the `ContentType` class.

Table 82. Member variables of the ContentType class.

Member variable	Description	Page
`BinaryFile`	Indicates that the ODA generates binary files as its content.	223
`BusinessObject`	Indicates that the ODA generates business object definitions as its content.	223

BinaryFile

Indicates that the ODA generates binary files as its content.

Type
`public static final ContentType BinaryFile`

Notes
The `contentType` member variable indicates that the ODA supports generation of binary files as content. Therefore, the ODA implements the `IGeneratesBinFiles` interface. File content can be generated using either of the content protocols:

- The on-request content protocol requires that the ODA implement the `generateBinFiles()` method to handle generation of the files.
- The callback content protocol requires that the ODA implement some user-defined method to handle generation of the files.

For more information, see "Generating binary files as content" on page 133.

BusinessObject

Indicates that the ODA generates business object definitions as its content.

Type
`public static final ContentType BusinessObject`

Notes
The `contentType` member variable indicates that the ODA supports generation of business object definitions as content. Therefore, the ODA implements the `IGeneratesBoDefs` interface. Business-object-definition content must be generated using the on-request content protocol, which requires that the ODA implement the

generateBoDefs() method to handle generation of the business object definitions. For more information, see "Generating business object definitions as content" on page 110.

Methods

Table 83 summarizes the methods of the ContentType class.

Table 83. Member methods of the ContentType class

Member method	Description	Page
ContentType()	Creates a content-type object.	224
equals()	Compares two content-type objects.	224
from_int()	Generates a content-type object for a specified ordinal value.	225
toString()	Returns a literal representation of the current content-type object.	225
value()	Returns an ordinal value for the current content type.	225
xmlObject()	Generates an XML object that represents the current content-type object.	225

ContentType()

Creates a content-type object.

Syntax

public ContentType(int *contTypeOrdValue*);

Parameters

contTypeOrdValue
> Is the ordinal value that represents the content type.

Return values

A newly instantiated ContentType object.

equals()

Compares two content-type objects.

Syntax

public boolean equals(Object *contentTypeObj*);

Parameters

contentTypeObj
> Is a reference to the ContentType object to compare with the current ContentType object.

Return values

A boolean value that indicates whether the two content-type objects are equal.

Notes

The equals() method overrides the equals() method in the java.lang.Object class.

from_int()

Generates a content-type object for the specified ordinal value.

Syntax

```
public static ContentMetaData from_int(int contTypeOrdValue);
```

Parameters

contTypeOrdValue

Is the ordinal value that represents the current content type.

Return values

A `ContentMetaData` object that represents the content type of the specified ordinal value.

See also

value()

toString()

Returns a literal representation of the current content-type object.

Syntax

```
public String toString();
```

Parameters

None.

Return values

A `String` object that contains the literal representation of the current content-type object.

Notes

The `toString()` method overrides the `toString()` method in the `java.lang.Object` class.

value()

Returns an ordinal value for the current content type.

Syntax

```
public int value();
```

Parameters

None.

Return values

An integer ordinal value that represents the current content type.

See also

from_int()

xmlObject()

Generates an XML object that represents the current content-type object.

Syntax

```
public XMLObject xmlObject();
```

Parameters

None.

Return values

An `com.crossworlds.ODK.XMLObject` object that represents the current content-type object.

Chapter 17. DependentCondition class

The Object Discovery Agent Development Kit (ODK) API uses the
DependentCondition class to represent dependent conditions, which define
conditions that restrict the value of a dependent agent property. When the
associated input condition evaluates to true, the dependent condition is applied to
the dependent property. Dependent conditions and their associated input condition
(or conditions) are stored in a complete-condition (CompleteCondition) object.

Note: For information on complete conditions, see Chapter 14,
"CompleteCondition class," on page 215.

The DependentCondition class defines the following:
- "Member variables"
- "Methods" on page 229

Member variables

Table 84 summarizes the member variables in the DependentCondition class.

Table 84. Member variables of the DependentCondition class.

Member variable	Description	Page
isDynamic	Specifies whether Business Object Wizard should check the value of the specific-value property before it makes the dependent condition's comparison	227
operatorType	Specifies the operator type for the dependent condition.	227
propertyName	Specifies the name of the dependent property to be displayed.	228
specificValue	Specifies the value to compare with the dependent property's value.	228
typeOfSpecificValue	Specifies the data type of the dependent condition's specific value.	228

isDynamic

Specifies whether Business Object Wizard should check the value of the
specific-value property before making the dependent condition's comparison.

Type
```
public boolean isDynamic
```

Notes
When the isDynamic member variable is true, Business Object Wizard obtains the
value of the property that the specificValue member variable specifies *before* it
performs the comparison with the dependent property's value. If specificValue
contains a constant, isDynamic should be set to false.

operatorType

Specifies the operator type for the dependent condition.

Type

```
public String operatorType
```

Notes

The `operatorType` specifies the kind of comparison that Business Object Wizard makes between the value of the dependent property (which the `propertyName` member variable specifies) and the `specificValue`. Valid values for the `operatorType` variable are the operator constants, which are defined in the `CompleteCondition` class. For more information, see Table 76 on page 215.

propertyName

Specifies the name of the dependent property.

Type

```
public String propertyName
```

Notes

The `propertyName` member variable contains the name of the dependent property. It is the value of the dependent property that the dependent condition restricts (when the associated input conditions evaluate to true).

specificValue

Specifies the value to compare with the dependent property's value.

Type

```
public String specificValue
```

Notes

The `specificValue` holds the dependent condition's value, which Business Object Wizard compares with the value of the dependent property (which the `propertyName` member variable specifies). The kind of comparison is determined by the `operatorType` variable. The specific value can be either of the following:

- A constant (of the same type as the dependent property)

  For example, if a dependent condition specifies the Less Than operator (`CompleteCondition.OP_LESS_THAN`) as its `operatorType` and specifies a value of 5 as its `specificValue`, the dependent property's value must be less than 5 when the associated input conditions evaluate to true.

- The name of another agent property

  For example, if a dependent condition specifies the Greater Than operator (`CompleteCondition.OP_GREATER_THAN`) as its `operatorType` and specifies the name of the "Property1" property as its `specificValue`, the dependent property's value must be greater than the value of Property1 agent property when the associated input conditions evaluate to true.

The `specificValue` variable is declared of type `String` so that it can hold any kind of value. However, to make comparisons properly, Business Object Wizard needs to know the actual data type of the specific value, which the `typeOfSpecificValue` member variable contains.

typeOfSpecificValue

Specifies the data type of the dependent condition's specific value.

Type

```
public int typeOfSpecificValue
```

Notes

The `typeOfSpecificValue` holds the data type for the dependent condition's specific value. The `specificValue` variable is declared of type `String` so that it can hold any kind of value. However, to make comparisons properly, Business Object Wizard needs to know the actual data type of the specific value. Valid values for the `typeOfSpecificValue` variable are the property-type constants, which are defined in the `AgentProperty` class. For more information, see Table 67 on page 173.

For example, if the dependent condition's specific value is an integer constant of 5:

- The `specificValue` variable holds the string "5".
- The `typeOfSpecificValue` variable holds the `AgentProperty.TYPE_INTEGER` property-type constant.

Methods

Table 85 summarizes the methods in the `DependentCondition` class.

Table 85. Member methods of the DependentCondition class.

Member Method	Description	Page
`DependentCondition()`	Creates a dependent-condition object.	229
`copy()`	Copies the current dependent condition into the specified `DependentCondition` object.	230

DependentCondition()

Creates a dependent-condition object.

Syntax

```
public DependentCondition();

public DependentCondition(String name, String op,
   boolean isDyn, int type, String specificVal);
```

Parameters

isDyn   Indicates whether to obtain the value of the specific-value property dynamically; the value of this parameter initializes the `isDynamic` member variable ("isDynamic" on page 227).

name   is the name of the dependent property; the value of this parameter initializes the `propertyName` member variable ("propertyName" on page 228)

op   Is the operator that specifies the kind of comparison to make; the value of this parameter initializes the `operatorType` member variable ("operatorType" on page 227).

specificVal   Specifies the specific value of the dependent condition; the value of this parameter initializes the `specificValue` member variable ("specificValue" on page 228).

type   Specifies the data type of the specific value; the value of this parameter initializes the `typeOfSpecificValue` member variable ("typeOfSpecificValue" on page 228).

Return values

A newly instantiated `DependentCondition` object.

copy()

Copies the current dependent condition into a specified dependent-condition object.

Syntax

```
public void copy(DependentCondition depCond);
```

Parameters

depCond   Is a reference to the dependent-condition object into which the current dependent condition is copied.

Return values

None.

Chapter 18. IGeneratesBinFiles interface

The Object Discovery Agent Development Kit (ODK) API uses the IGeneratesBinFiles interface to define the functionality required by the Object Discovery Agent (ODA) for the generation of binary files as its content. This interface defines the set of methods that the ODA developer must implement to enable the ODA to generate binary files. Business Object Wizard calls the methods of the IGeneratesBinFiles interface to generate and access content that is file objects. A file object is a Java File object, which represents a binary operating-system file.

Note: An ODA must also support the generation of business object definitions as its content. To enable an ODA to generate business object definitions from source data, you must implement the IGeneratesBoDefs interface. For more information, see Chapter 19, "IGeneratesBoDefs interface," on page 235.

To provide the ODA with the ability to generate file objects, the ODA developer must take the following steps:

- In the definition of the ODA class (which is an extension of the ODKAgentBase2 class), include IGeneratesBinFiles as an interface that the ODA implements.
- Within the ODA class, implement the methods of the IGeneratesBinFiles interface. Because IGeneratesBinFiles is an interface, ODA developers must implement *all* methods in Table 86..

Table 86. Member methods of the IGeneratesBinFiles interface

Member method	Description	Page
generateBinFiles()	Generates file objects for the source nodes chosen from the data source.	231
getBinFile()	Retrieves generated file objects.	232
getContentProtocol()	Indicates the content protocol supported for this binary-file content type.	233

generateBinFiles()

Generates files objects.

Syntax

```
public ContentMetaData generateBinDefs(String[] strNames);
```

Parameters

strNames [ ]   Is an array of String objects. This argument is not currently used.

Return values

A ContentMetaData object, which describes the generated file objects.

Exceptions

ODKException   Thrown if the generation of the binary files fails.

Notes

The purpose of the `generateBinFiles()` method depends on the content protocol
that the ODA uses for generation of file (`ContentType.BinaryFile`) content, as
follows:

- If the ODA generates files "on request", Business Object Wizard explicitly calls
  the `generateBinFiles()` method to generate the files.
- If the ODA generates files through callbacks, Business Object Wizard *never*
  explicitly calls the `generateBinFiles()` method. Instead, the ODA uses some
  other way to generate the files, which Business Object Wizard can then access.

If the ODA generates files "on request", the `generateBinFiles()` method is the
content-generation method for the `IGeneratesBinFiles` interface. It can create file
objects that contain information about the business-object-definition-generation
process. Business Object Wizard calls the `generateBinFiles()` method to generate
content (if the ODA supports generation of file content). It calls this method in
Step 5, Generating Business Objects, of its start.

For the on-request protocol, this method does not actually return the generated
content. Instead, it returns a content-metadata (`ContentMetaData`) object, which
contains information that describes the generated content. From this returned
content-metadata object, Business Object Wizard can determine whether the
content-generation process is complete. When generation is complete, Business
Object Wizard obtains the generated file objects with the `getBinFiles()` method.
For more information on how to implement `generateBinFiles()`, see "Generating
files" on page 135..

See also

generateBoDefs(), getBinFile()

---

getBinFile()

Retrieves the generated file objects from the generated-content structure.

Syntax

```
public File[] getBinFile(long index);
```

Parameters

index   Specifies the file object to retrieve from the generated-content
                 structure.

Exceptions

ODKException   Thrown if Business Object Wizard encounters a problem getting
                   the generated file objects from the generated-content structure.

Notes

The `getBinFile()` method is the content-retrieval method for the
`IGeneratesBinFiles` interface. It retrieves generated file objects from the ODA's
generated-content structure, which is the structure that the ODA populated with
the generated file objects. The method that populated the generated-content
structure depends on the content protocol that the ODA supports for file
generation, as follows:

- If the ODA generates files "on request", the generateBinFiles() method has populates the generated-content structure.
- If the ODA generates files through callbacks, some user-defined method populates the generated-content structure.

The value of the *index* argument determines whether to getBinFile() returns one or all generated file objects, as Table 87 shows.

Table 87. Specifying the file objects to return

Value of *index* argument	Action of getBinFile()
In the range 0 to *count* (where *count* is the member variable in the content-metadata object that specifies the number of file objects in the generated-content structure)	Return an array that contains *one* file (Java File) object, the File object at the specified *index* position in the generated-content structure.
ODKConstant.GET_ALL_OBJECTS	Return an array of *all* generated file objects in the generated-content structure.

For more information on how to implement getBinFile(), see "Providing access to generated files" on page 139.

See also

generateBinFiles(), getBoDefs()

getContentProtocol()

Indicates the content protocol that the ODA supports for a specified content type.

Syntax

```
public long getContentProtocol(ContentType contentType);
```

Parameters

contentType   Indicates the content type for which the method obtains the supported content protocol.

Return values

A long-integer (long) value that indicates the content protocol that the ODA implements. Compare this long value with the following content-protocol constants:

ODKConstant.CONTENT_PROTOCOL_CALLBACK
Indicates that the ODA supports a callback protocol; that is the ODK initiates generation of the specified content and notifies Business Object Wizard when generation is complete.

ODKConstant.CONTENT_PROTOCOL_ONREQUEST
Indicates that the ODA supports an on-demand protocol; that is, Business Object Wizard initiates generation of the specified content type.

Notes

The getContentProtocol() method is the single method defined in the IGeneratesContent interface, which the IGeneratesBoDefs interface extends.

Business Object Wizard calls `getContentProtocol()` to determine the content protocol that the ODA supports fort the *contentType* content type. For more information, see "Indicating the implemented content protocols" on page 109.

Chapter 19. IGeneratesBoDefs interface

The Object Discovery Agent Development Kit (ODK) API uses the IGeneratesBoDefs interface to define the functionality required by the Object Discovery Agent (ODA) for the generation of business object definitions as its content. This interface defines the set of methods that the ODA developer must implement to enable the ODA to generate business object definitions from source data. Business Object Wizard calls the methods of the IGeneratesBoDefs interface to obtain source nodes, as well as generate and access content that is business object definitions.

Note: An ODA can also support the generation of file objects as its content. To enable an ODA to generate binary files from source data, you must implement the IGeneratesBinFiles interface. For more information, see Chapter 18, "IGeneratesBinFiles interface," on page 231.

To provide the ODA with the ability to generate business object definitions objects from source data, the ODA developer must take the following steps:

- In the definition of the ODA class (which is an extension of the ODKAgentBase2 class), include IGeneratesBoDefs as an interface that the ODA implements.
- Within the ODA class, implement the methods of the IGeneratesBoDefs interface. Because IGeneratesBoDefs is an interface, ODA developers must implement *all* methods in Table 88..

Table 88. Member methods of the IGeneratesBoDefs interface

Member method	Description	Page
generateBoDefs()	Generates business object definitions for the specified source nodes from the data source.	235
getBoDefs()	Retrieves generated business object definitions.	236
getContentProtocol()	Indicates the content protocol supported for this business-object-definition content type.	237
getTreeNodes()	Constructs an array of tree nodes that represent the hierarchy of source nodes.	238

generateBoDefs()

Generates business object definitions for the specified source nodes.

Syntax

```
public ContentMetaData generateBoDefs(String[] srcNodeNames);
```

Parameters

srcNodeNames [ ]
       Is an array that contains the names of source nodes that the user has selected.

Return values

A `ContentMetaData` object, which describes the generated business object definitions for the source nodes named in the *srcNodeNames* argument.

Exceptions

ODKException   Thrown if the generation of business object definitions fails.

Notes

The `generateBoDefs()` method is the content-generation method for the `IGeneratesBoDefs` interface. It creates business object definitions for each of the source nodes named in the *srcNodeNames* array. The user has selected these source nodes in the Select Source dialog box of Business Object Wizard. Once the user has finished selecting source nodes, Business Object Wizard calls the `generateBinFiles()` method to generate content. It calls this method in Step 5, Generating Business Objects, of its start.

Note: Business Object Wizard always calls `generateBoDefs()` because the ODA must support an on-request content protocol for generation of business object definitions. For more information on content protocols, see "Choosing the ODA content protocol" on page 108.

The goal of the `generateBoDefs()` method is to generate a business object definition (`BusObjDef` object) for each user-selected source node, store it in the generated-content structure, and return a content-metadata (`ContentMetaData`) object that describes the generated content. This method does *not* actually return the generated content to Business Object Wizard. From this returned content-metadata object, Business Object Wizard can determine whether the content-generation process is complete. When generation is complete, Business Object Wizard obtains the generated business object definitions with the `getBoDefs()` method. For more information on how to implement `generateBoDefs()`, see "Generating business object definitions" on page 118.

See also

generateBinFiles(), getBoDefs()

---

getBoDefs()

Retrieves the generated business object definitions.

Syntax

```
public BusObjDef[] getBoDefs(long index);
```

Parameters

index   Specifies the business object definition to retrieve from the generated-content structure.

Exceptions

ODKException   Thrown if Business Object Wizard encounters a problem getting the generated business object definitions from the generated-content structure.

Notes

The getBoDefs() method is the content-retrieval method for the IGeneratesBoDefs interface. It retrieves generated business object definitions from the ODA's generated-content structure, which is the structure that the generateBoDefs() method populated with the generated business object definitions. The value of the *index* argument determines whether to getBoDefs() returns one or all generated business object definitions, as Table 89 shows.

Table 89. Specifying the business object definitions to return

Value of *index* argument	Action of getBoDefs()
In the range: 0 to *count* (where *count* is the member variable in the content-metadata object that specifies the number of business object definitions in the generated-content structure)	Return an array that contains *one* business-object-definition (BusObjDef) object, the BusObjDef object at the specified *index* position in the generated-content structure.
ODKConstant.GET_ALL_OBJECTS	Return an array of *all* generated business object definitions in the generated-content structure.

For more information on how to implement getBoDefs(), see "Providing access to generated business object definitions" on page 131.

See also

generateBoDefs(), getBinFile()

---

getContentProtocol()

Indicates the content protocol that the ODA supports for a specified content type.

Syntax

```
public long getContentProtocol(ContentType contentType);
```

Parameters

contentType   Indicates the content type for which the method determines the supported content protocol.

Return values

A long-integer (long) value that indicates the content protocol that the ODA implements. Compare this long value with the following content-protocol constants:

ODKConstant.CONTENT_PROTOCOL_CALLBACK
> Indicates that the ODA supports a callback protocol; that is the ODK initiates generation of the specified content and notifies Business Object Wizard when generation is complete.

ODKConstant.CONTENT_PROTOCOL_ONREQUEST
> Indicates that the ODA supports an on-demand protocol; that is, Business Object Wizard initiates generation of the specified content type.

Notes

The getContentProtocol() method is the single method defined in the IGeneratesContent interface, which the IGeneratesBoDefs interface extends. Business Object Wizard calls getContentProtocol() to determine the content protocol that the ODA supports fort the *contentType* content type. For more information, see "Indicating the implemented content protocols" on page 109.

getTreeNodes()

Constructs an array of tree nodes that represents one level in the hierarchy of source nodes.

Syntax

```
public TreeNode[] getTreeNodes(String parentNodePath, String searchPattern);
```

Parameters

parentNodePath

Is a fully qualified path from the top-level node to the source node whose children are to be returned to Business Object Wizard; each node in the path is separated by a colon (:).

searchPattern Is the user-specified search pattern for the child nodes in the expandable *parentNodePath* node.

Return values

An array of TreeNode objects, which tree-node object is a child node in the hierarchy of specified objects.

Exceptions

ODKException Thrown if the Object Discovery Agent encounters a problem getting the tree nodes.

Notes

The getTreeNodes() method is the source-node-generation method for the IGeneratesBoDefs interface. Business Object Wizard invokes getTreeNodes() to obtain the array of tree nodes that initializes its Select Source (Step 3) dialog box. From this dialog box, the user selects specific source nodes for business-object-definition generation. Within the getTreeNodes() method, you must construct tree nodes to represent the hierarchy of source nodes in the data source. The getTreeNode() method returns this source-node hierarchy as an array of TreeNode objects to its caller, Business Object Wizard.

The tree-node array that getTreeNodes() returns provides the source nodes at one particular level of the source-node hierarchy. At any given level, some source nodes might be expandable (have child nodes) and some might be leaf (terminating) nodes. The user can traverse the hierarchy by expanding any source node that displays a plus (+) sign to its left. When the user expands a node, Business Object Wizard calls getTreeNodes() again, providing as its *parentNodePath* argument the name of the node the user wants to expand. This node name consists of the names of each of the nodes in the path, separated by a colon (:).

Note: An ODA uses the colon rather than a slash or backslash to keep the path operating-system-independent.

The getTreeNodes() method performs the following basic tasks to generate the source nodes:

1. Parse the *parentNodePath* to identify the parent object to search for in the data source.
2. Discover the child objects for the specified data-source parent object.

   If Business Object Wizard provides a *searchPattern* argument to getTreeNodes(), the user has specified search criteria. Therefore, getTreeNodes() must return only those child nodes of the *parentNodePath* node that match *searchPattern* search criteria. The ability to apply a search pattern to source nodes requires that the following conditions are true:

   • The user-specified *searchPattern* must match the search criteria that the ODA supports.

     The searchPatternDesc member variable in the ODA's metadata (AgentMetaData) object provides a description to the user of the supported search criteria. However, the getTreeNodes() method must parse the user-specified *searchPattern* to ensure that it matches the supported search criteria.

   • The ODA supports search patterns.

     The searchableNodes member variable in the ODA's metadata (AgentMetaData) object is true. If searchableNodes is false, the "Search for items" menu item (which initiates the user's entry of search criteria) is not available. Therefore, the user cannot enter search criteria.

3. Construct tree nodes for the child objects and put these nodes into the tree-node array.

For more information on how to implement getTreeNodes(), see "Generating source nodes" on page 111.

See also

For related reference information, see Chapter 25, "TreeNode class," on page 267.

Chapter 20. InputCondition class

The Object Discovery Agent Development Kit (ODK) API provides the `InputCondition` class to represent input conditions, which specify conditions on the value of an agent property. When an input condition evaluates to true, the associated dependent condition is applied to the dependent property. An input condition and its associated dependent condition (or conditions) are stored in a complete-condition (`CompleteCondition`) object.

Note: For information on complete conditions, see Chapter 14, "CompleteCondition class," on page 215.

The `InputCondition` class defines the following:
- "Member variables"
- "Methods" on page 243

Member variables

Table 90 summarizes the member variables in the `InputCondition` class.

Table 90. Member variables of the InputCondition class.

Member variable	Description	Page
isDynamic	Specifies whether Business Object Wizard should check the value of the specific-value property before it makes the input condition's comparison.	241
operatorType	Specifies the operator type for the input condition.	241
specificValue	Specifies the value to compare with the agent property's value.	242
typeOfSpecificValue	Specifies the data type of the input condition's specific value.	242

isDynamic

Specifies whether Business Object Wizard should check the value of the specific-value property before making the input condition's comparison.

Type
```
public boolean isDynamic
```

Notes
When the `isDynamic` member variable is `true`, Business Object Wizard obtains the value of the property that the `specificValue` member variable specifies *before* it performs the comparison with the agent property's value. If `specificValue` contains a constant, `isDynamic` should be set to `false`.

operatorType

Specifies the operator type for the input condition.

Type

```
public String operatorType
```

Notes

The `operatorType` specifies the kind of comparison that Business Object Wizard makes between the agent property's value and the `specificValue`. Valid values for the `operatorType` variable are the operator constants, which are defined in the `CompleteCondition` class. For more information, see Table 76 on page 215.

specificValue

Specifies the value to compare with the agent property value.

Type

```
public String specificValue
```

Notes

The `specificValue` holds the input condition's value, which Business Object Wizard compares with the agent property's value. The kind of comparison is determined by the `operatorType` variable. The specific value can be either of the following:

- A constant (of the same type as the agent property)

  For example, if an input condition specifies the Less Than operator (`CompleteCondition.OP_LESS_THAN`) as its `operatorType` and specifies a value of 5 as its `specificValue`, the associated dependent conditions apply when the agent property's value is less than 5.

- The name of another agent property

  For example, if an input condition specifies the Greater Than operator (`CompleteCondition.OP_GREATER_THAN`) as its `operatorType` and specifies the name of the "Property1" property as its `specificValue`, the associated dependent conditions apply when the agent property's value is greater than the value of `Property1` agent property.

The `specificValue` variable is declared of type `String` so that it can hold any kind of value. However, to make comparisons properly, Business Object Wizard needs to know the actual data type of the specific value, which the `typeOfSpecificValue` member variable contains.

typeOfSpecificValue

Specifies the data type of the input condition's specific value.

Type

```
public int typeOfSpecificValue
```

Notes

The `typeOfSpecificValue` holds the data type for the input condition's specific value. The `specificValue` variable is declared of type `String` so that it can hold any kind of value. However, to make comparisons properly, Business Object Wizard needs to know the actual data type of the specific value. Valid values for the `typeOfSpecificValue` variable are the property-type constants, which are defined in the `AgentProperty` class. For more information, see Table 67 on page 173.

For example, if the input condition's specific value is an integer constant of 5:

- The `specificValue` variable holds the string "5".

- The `typeOfSpecificValue` variable holds the `AgentProperty.TYPE_INTEGER` property-type constant.

Methods

Table 91 summarizes the methods in the `InputCondition` class.

Table 91. Member methods of the InputCondition class.

Member method	Description	Page
InputCondition()	Creates an input-condition object.	243
copy()	Copies the current input condition into the specified input-condition object.	243

InputCondition()

Creates an input-condition object.

Syntax
```
public InputCondition();
public InputCondition(String operator, boolean isDyn, int type,
    String specificVal);
```

Parameters

isDyn   Indicates whether to obtain the value of the specific-value property dynamically; the value of this parameter initializes the `isDynamic` member variable ("isDynamic" on page 241).

operator   Is the operator that specifies the kind of comparison to make; the value of this parameter initializes the `operatorType` member variable ("operatorType" on page 227).

specificVal Is the specific value of the input condition; the value of this parameter initializes the `specificValue` member variable ("specificValue" on page 242).

type   Specifies the data type of the specific value; the value of this parameter initializes the `typeOfSpecificValue` member variable ("typeOfSpecificValue" on page 242).

Return values
A newly instantiated `InputCondition` object.

copy()

Copies the current input condition into the specified input-condition object.

Syntax
```
public void copy(InputCondition inputCond);
```

Parameters

inputCond   is a reference to the `InputCondition` object into which the current input condition is copied.

Return values
None.

Chapter 21. ODKAgentBase2 class

The Object Discovery Agent Development Kit (ODK) API provides the `ODKAgentBase2` class as the base class for an Object Discovery Agent (ODA). From this class, an ODA developer must derive an *ODA class* and implement the abstract methods for the ODA.

Note: The `ODKAgentBase2` class extends the `ODKAgentBase` class of the low-level ODA library. It inherits the `getAgentProperties()`, `getVersion()`, `init()`, and `terminate()` methods of this class. It also "disables" the `getTreeNodes()` and `generateDefs()` methods of this class because they are now replaced with functionality defined in the `getTreeNodes()` and `generateBoDefs()` methods of the `IGeneratesBoDefs` interface.

> **Important**
>
> All ODAs *must* extend this ODA base class and provide implementations for all its methods *except* `getVersion()`.

Table 92 summarizes the methods of the `ODKAgentBase2` class.

Table 92. Member methods of the ODKAgentBase2 class

Member method	Description	Page
`getAgentProperties()`	Sends an array of ODA configuration properties to Business Object Wizard.	245
`getMetaData()`	Sends the ODA metadata to Business Object Wizard.	246
`getVersion()`	Retrieves the version of the ODA.	247
`init()`	Initializes the ODA.	247
`terminate()`	Terminates the ODA, performing any required clean-up tasks.	247

getAgentProperties()

Sends an array of ODA configuration properties to Business Object Wizard.

Syntax

```
public abstract AgentProperty[] getAgentProperties();
```

Parameters

None.

Return values

An array of `AgentProperty` objects, one object for each ODA configuration property.

Exceptions

ODKException   Thrown if the ODA fails to get the configuration properties.

Notes

Business Object Wizard invokes the `getAgentProperties()` method to get the array of ODA configuration properties that initializes its Configure Agent (Step 2) dialog box. From this dialog box, you can enter or change these property values.

Important: The `getAgentProperties()` method is an abstract method that has no default implementation.Therefore, the ODA class *must* implement this method.

Within the `getAgentProperties()` method, you must instantiate and initialize agent-property (`AgentProperty`) objects for each ODA configuration property and store each property in the configuration-property array. The `getAgentProperties()` method returns this configuration-property array to its caller, Business Object Wizard. Once the user has set the configuration properties from the Configure Agent dialog box, Business Object Wizard reads these user-initialized properties in the ODA-runtime memory. You can obtain the user-initialized property with either the `getAgentProperty()` or `getAllAgentProperties()` method in the `ODKUtility` class. For more information on how to implement `getAgentProperties()`, see "Obtaining configuration properties" on page 102.

See also

getAgentProperty(), getAllAgentProperties()

---

getMetaData()

Sends the ODA metadata to Business Object Wizard.

Syntax

```
public abstract AgentMetaData getMetaData();
```

Parameters

None.

Return values

An `AgentMetaData` object that contains the metadata for the ODA.

Notes

Business Object Wizard invokes the `getMetaData()` method to get the ODA's metadata. It calls `getMetaData()` after its call to the `getAgentProperties()` method completes. The `getMetaData()` method returns the `AgentMetaData` object for the ODA. This `AgentMetaData` object contains metadata for the ODA, such as the version and the generated content it supports. Within this method, call the `AgentMetaData()` constructor and return the instantiated metadata object.

Important: The `getMetaData()` method is an abstract method that has no default implementation.Therefore, the ODA class *must* implement this method.

For more information on how to implement `getMetaData()`, see "Initializing ODA metadata" on page 104.

getVersion()

Retrieves the version of the ODA runtime.

Syntax

```
public String getVersion();
```

Parameters

None.

Return values

An `String` that contains the version of the ODA runtime.

init()

Performs the ODA initialization tasks.

Syntax

```
public abstract void init();
```

Parameters

None.

Return values

None.

Exceptions

ODKException   Thrown if the ODA initialization fails.

Notes

Business Object Designer Express invokes the `init()` method to initialize the Object Discovery Agent. It calls `init()` after its calls to the `getAgentProperties()` and `getMetaData()` methods complete. Typically, ODA initialization includes obtaining values for the ODA's configuration properties, establishing a connection to the data source, (which could be an application, database, XML file, or other business object source), and allocating any resources that the ODA requires.

Important: The `init()` method is an abstract method that has no default implementation.Therefore, the ODA class *must* implement this method.

For more information on how to implement `init()`, see "Initializing the ODA start" on page 105.

terminate()

Terminates the ODA, performing any required clean-up tasks.

Syntax

```
public abstract void terminate();
```

Parameters

None.

Return values

None.

Notes

Business Object Designer Express calls the `terminate()` method when it shuts down the ODA. In your implementation of this method, it is good practice to free all the memory and disconnect from the data source.

Important: The `terminate()` method is an abstract method that has no default implementation. Therefore, the ODA class *must* implement this method.

Deprecated Methods

Some methods in the `ODKAgentBase2` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

lists the deprecated methods for the `ODKAgentBase2` class. If you are writing a new ODA (not modifying an existing ODA), you can ignore this section.

Table 93. Deprecated methods of the ODKAgentBase2 class

Deprecated method	Replacement
`getTreeNodes()` (inherited from `ODKAgentBase`)	`getTreeNodes()` in the `IGeneratesBoDefs` interface
`generateDefs()` (inherited from `ODKAgentBase`)	`generateBoDefs()` in the `IGeneratesBoDefs` interface (to generate business object definitions) **Note:** You can also generate files with the `generateBinFiles()` method in the `IGeneratesBinFiles` interface.

Chapter 22. ODKConstant interface

The Object Discovery Agent Development Kit (ODK) API uses the `ODKConstant` interface to provide general constants to the Object Discovery Agent (ODA). Any class that implements the `ODKConstant` interface can access its defined constants directly. For example, if the `TreeNode` class implements the `ODKConstant` interface, its methods can access the `MSG_QUESTION` constant as follows:

```
int message_icon = MSG_QUESTION;
```

The `ODKConstant` interface defines static member variables to represent the following kinds of constants:

- "String-value constants"
- "User-response-dialog constants"
- "Cardinality constants" on page 250
- "Trace-level constants" on page 251
- "Message-type constants" on page 251
- "Node-nature constants" on page 251
- "Content-protocol constants" on page 252
- "Content-index constant" on page 252

String-value constants

Table 94 summarizes the string-value constants in the `ODKConstant` interface. These constants represent special attribute values of Blank and Ignore. All string-value constants are of type `String`.

Table 94. String-value constants of the ODKConstant interface.

String-value constant	Description
CW_EMPTY_STRING	Specifies the defined constant for an empty String ("")
CW_NULL_STRING	Specifies the defined constant for a `null` value

User-response-dialog constants

The `sendMsg()` method, defined in the `ODKUtility` class, provides the ODA developer with a means to display a user-response dialog box. To provide support for `sendMsg()`, the `ODKConstant` interface provides the user-response-dialog constants listed in Table 95. All user-response-dialog constants are of type integer (`int`).

Table 95. User-response-dialog constants of the ODKConstant interface.

User-response-dialog constant	Description
dialog-button constants	
MSG_OK	Specifies the display of the OK button on the user-response dialog box
MSG_OKCANCEL	Specifies the display of the OK and Cancel buttons on the user-response dialog box
MSG_RETRYCANCEL	Specifies the display of the Retry and Cancel buttons on the user-response dialog box

User-response-dialog constant	Description
MSG_ABORTRETRYIGNORE	Specifies the display of the Retry, Ignore, and Abort buttons on the user-response dialog box
MSG_YESNO	Specifies the display of the Yes and No buttons on the user-response dialog box
MSG_YESNOCANCEL	Specifies the display of the Yes, No, and Cancel buttons on the user-response dialog box
dialog-icon constants	
MSG_ERROR	Specifies that the user-response dialog box displays the error icon
MSG_CRITIALERROR	Specifies that the user-response dialog box displays the critical-error icon
MSG_WARNING	Specifies that the user-response dialog box displays the warning icon
MSG_INFORMATION	Specifies that the user-response dialog box displays the information icon
MSG_QUESTION	Specifies that the user-response dialog box displays the question-mark icon
User-response constants	
ODK_OK	Specifies the OK button on the user-response dialog box
ODK_CANCEL	Specifies the Cancel button on the user-response dialog box
ODK_RETRY	Specifies the Retry button on the user-response dialog box
ODK_IGNORE	Specifies the Ignore button on the user-response dialog box
ODK_ABORT	Specifies the Abort button on the user-response dialog box
ODK_YES	Specifies the Yes button on the user-response dialog box
ODK_NO	Specifies the No button on the user-response dialog box
ODK_CLOSE	Specifies the Close button on the user-response dialog box
ODK_HELP	Specifies the Help button on the user-response dialog box

Cardinality constants

Table 96 summarizes the cardinality constants in the ODKConstant interface. These constants represent valid values for the cardinality of an agent property. All cardinality constants are of type String.

Table 96. Cardinality constants of the ODKConstant interface.

Cardinality constant	Description
MULTIPLE_CARD	Specifies that an agent property can have *more than one* value; that is, the user can specify more than one value for the property.
SINGLE_CARD	Specifies that an agent property can have *only one* value; that is, the user can only specify one value for the property.

Trace-level constants

Table 97 summarizes the trace-level constants in the `ODKConstant` interface. These constants represent valid trace levels for the tracing method, `trace()` (defined in the `ODKUtility` class) . All trace-level constants are of type integer (`int`).

Table 97. Trace-level constants of the ODKConstant interface.

Trace-level constant	Description
TRACELEVEL0	Represents a trace level 0 (error logging on; tracing off)
TRACELEVEL1	Represents a trace level 1
TRACELEVEL2	Represents a trace level 2
TRACELEVEL3	Represents a trace level 3
TRACELEVEL4	Represents a trace level 4
TRACELEVEL5	Represents a trace level 5

For a description of the expected content at each trace level, see Table 15 on page 76.

Message-type constants

Table 98 summarizes the message-type constants in the `ODKConstant` interface. These constants indicate the severity level for a message that the `trace()` method (defined in the `ODKUtility` class) outputs. All message-type constants are of type integer (`int`).

Table 98. Message-type constants of the ODKConstant interface.

Message-type constant	Description
XRD_FATAL	Represents a fatal error
XRD_ERROR	Represents an error
XRD_URGENTWARNING	Represents an urgent warning
XRD_WARNING	Represents an warning
XRD_INFO	Represents an informational message
XRD_TRACE	Represents a trace message

Important: The `ODKConstant` interface also provides message-type constants of the form XRD_INT_*messageType*. In addition, it defines the XRD_UNKNOWN constant to represent an undefined message type. These message-type constants are for internal use only. Do *not* use these message-type constants in your ODA.

Node-nature constants

Table 99 summarizes the node-nature constants in the `ODKConstant` interface. These constants indicate the actions that the user can take on a tree node when it displays within the Select Source dialog box of Business Object Wizard. All node-nature constants are of type integer (`int`).

Table 99. Node-nature constants of the ODKConstant interface.

Node-nature constant	Description
NODE_NATURE_NORMAL	Specifies that the tree node is "normal"; that is, the user can either expand the node or select the node (if it is a leaf node).

Node-nature constant	Description
NODE_NATURE_FILE	Specifies that the tree node can be associated with a file; that is, Business Object Wizard enables the Associate File menu item on the context-menu of the node name to allow the user to locate the file to associate with the node.

Content-protocol constants

Table 100 summarizes the content-protocol constants in the ODKConstant interface. These constants represent the content protocol for the ODA. All content-protocol constants are of type byte.

Table 100. Content-protocol constants of the ODKConstant interface.

Content-protocol constant	Description
CONTENT_PROTOCOL_ONREQUEST	Specifies that the ODA generates its content "on request"; that is, Business Object Wizard explicitly request content generation by the ODA. When such content is ready, the ODA runtime notifies Business Object Wizard, which can then retrieve the content at its convenience.
CONTENT_PROTOCOL_CALLBACK	Specifies that the ODA generates its content "spontaneously"; that is, it cannot predict or guarantee when its content will be generated. When such content is ready, the ODA must notify Business Object Wizard, which can then retrieve the content at its convenience.

Content-index constant

Table 101 summarizes the content-index constant in the ODKConstant interface. This constant represents a special value to the content-retrieval methods that indicates to return all generated content. The content-index constant is of type long.

Table 101. Content-index constant of the ODKConstant interface.

Content-index constant	Description
GET_ALL_OBJECTS	Passed as an argument to the content-retrieval method of the ODA. It specifies that the content-retrieval method should return *all* generated content.

For more information, see "Providing access to generated business object definitions" on page 131 and "Providing access to generated files" on page 139.

Chapter 23. ODKException class

The `ODKException` class is the base class for exceptions in the Object Discovery Agent Development Kit (ODK) API. The ODK API extends the Java `Exception` class to create its own exception class called:

`com.crossworlds.ODK.ODKException`

This class represents an *exception object*, which methods of the `ODK API` can throw.

Note: The reference description for each ODK API method lists the exceptions thrown by that method.

The `ODKException` class defines the following:
- "Methods"
- "Exception subclasses" on page 254

Methods

Table 102 summarizes the methods in the `ODKException` class.

Table 102. Member methods of the ODKException class.

Member method	Description	Page
`ODKException()`	Creates an ODK exception object.	253
`getMsg()`	Retrieves the exception message from the exception object.	253

ODKException()

Creates an exception object.

Syntax
`public ODKException(String `*`msg`*`);`

Parameters

msg   is the exception message for the exception object.

Return values
A newly instantiated `ODKException` exception object.

getMsg()

Retrieves the exception message from the exception object.

Syntax
`public String getMsg();`

Parameters
None.

Return values

A `String` that contains the exception message.

Exception subclasses

Within this `ODKException` class are subclasses that identify particular exceptions possible in the methods of the ODK API. Table 103 lists the subclassed exceptions.

Table 103. ODKException subclasses.

Exception subclass	Definition
`BusObjInvalidAttrException`	Thrown when an attribute is invalid.
`BusObjInvalidDefException`	Thrown when a business object definition is invalid.
`BusObjInvalidVerbException`	Thrown when a verb is invalid.
`BusObjNoSuchAttrException`	Thrown when the attribute does not exist in the business object definition.
`BusObjNoSuchVerbException`	Thrown when the verb is not supported by the business object definition.
`ODKInvalidNodeException`	Thrown to indicate a tree-node exception.
`ODKInvalidPropException`	Thrown to indicate exceptions caused by an invalid property.
`UnsupportedContentException`	Thrown if the ODA cannot supported the requested generated content.

Chapter 24. ODKUtility class

The Object Discovery Agent Development Kit (ODK) API provides the `ODKUtility` class to provide an Object Discovery Agent (ODA) with various utility methods. By obtaining a handle to the singleton `ODKUtility` object, the ODA can access the following functionality:

- Information in the memory of the ODA runtime:
  - User-specified values for the ODA configuration properties
  - User-specified values for the business-object properties
- Utility methods that provide the following features:
  - Ability to send messages that require user input
  - Ability to send non-blocking status messages
  - Ability to perform tracing

Note: Use the `getODKUtility()` method in this class to obtain a handle to the `ODKUtility` object.

Table 104 summarizes the methods in the `ODKUtility` class.

Table 104. Member methods of the ODKUtility class.

Member method	Description	Page
`contentComplete()`	Notifies Business Object Wizard that the ODA has completed content generation when using the callback protocol.	255
`getAgentProperty()`	Retrieves the specified ODA configuration property.	256
`getAllAgentProperties()`	Retrieves all ODA configuration properties.	257
`getAllBOSpecificProperties()`	Retrieves all business-object properties.	257
`getBOSpecificProperty()`	Retrieves the specified business-object property.	258
`getBOSpecificProps()`	Sends the specified business-object properties to the BO Properties dialog box for user input.	258
`getClientFile()`	Requests that Business Object Wizard retrieve a specified file.	259
`getMsg()`	Returns a message from the ODA message file.	260
`getODKUtility()`	Returns a handle to an ODKUtility object.	261
`sendMsg()`	Displays a user-response dialog box,which includes a message and button, and requires a response from the user.	262
`sendStatusMsg()`	Displays a message to the user.	264
`trace()`	Writes a message to the trace file.	264

contentComplete()

Notifies Business Object Wizard that the ODA has completed its generation of content for the callback content protocol.

Syntax

```
public void contentComplete(ContentMetaData contentMetaData);
```

Parameters

contentMetaData

Is a content-metadata object that describes the current state of the generated content.

Return values

None.

Notes

The contentComplete() method indicates that the ODA has completed generation of its content. In the callback protocol, Business Object Wizard does *not* initiate content generation by calling the appropriate content-generation method. Instead, the ODA initiates content generation and Business Object Wizard waits for the ODA to notify it when this content generation is complete. The ODA performs this notification by calling contentComplete(). Once Business Object Wizard is notified, it calls the appropriate content-retrieval method to obtain the generated content.

Important

If the ODA uses the callback protocol for a particular content generation, it *must* call the contentComplete() method to notify Business Object Wizard that the content is available. Otherwise, Business Object Wizard does not know that the generated content is available for retrieval.

The *contentMetaData* object must indicate the type of generated content as well as the number of items in the generated content.

getAgentProperty()

Retrieves the specified ODA configuration property.

Syntax

```
public AgentProperty getAgentProperty(String propName);
```

Parameters

propName   Is the name of the configuration property to retrieve.

Return values

An AgentProperty object that contains the specified configuration property, or null if no configuration property exists of that name exists.

Notes

The getAgentProperty() method retrieves the *propName* configuration property from the ODA-runtime memory. Business Object Wizard reads configuration properties into the ODA-runtime memory after the user specifies configuration-property values in the Configure Agent dialog box. This method

returns the specified configuration property as an `AgentProperty` object. You can obtain information about the property by accessing the object's member variables.

See also

getAgentProperties(), getAllAgentProperties()

getAllAgentProperties()

Retrieves all ODA configuration properties.

Syntax

```
public Hashtable getAllAgentProperties();
```

Parameters

None.

Return values

A reference to the `java.util.Hashtable` object that contains the ODA configuration properties (represented as `AgentProperty` objects), keyed on the property name.

Notes

The `getAllAgentProperties()` method retrieves all ODA configuration properties from the ODA-runtime memory. Business Object Wizard reads configuration properties into ODA-runtime memory after the user specifies configuration-property values in the Configure Agent dialog box. This method retrieves the configuration properties as a `Hashtable` object, which maps keys to values. The keys are the names of the properties and values are the associated property values. Use methods of the `Hashtable` class (such as `keys()` and `elements()`) to obtain the information from this structure.

See also

getAgentProperties(), getAgentProperty()

getAllBOSpecificProperties()

Retrieves all business-object properties from the BO Properties dialog box.

Syntax

```
public Hashtable getAllBOSpecificProperties();
```

Parameters

None.

Return values

A reference to a `java.util.Hashtable` object that contains the business-object properties (represented as `AgentProperty` objects), keyed on the property name.

Notes

The `getAllBOSpecificProperties()` method retrieves all business-object properties from the ODA-runtime memory. Business Object Wizard saves these properties into

memory after the user specifies their values in the BO Properties dialog box (part of Step 5). This method returns the business-object properties as a `Hashtable` object, which maps keys to values. The keys are the names of the business-object properties and values are the associated property values. Use methods of the `Hashtable` class (such as `keys()` and `elements()`) to obtain the information from this structure.

See also

getBOSpecificProperty(), getBOSpecificProps()

---

getBOSpecificProperty()

Retrieves the specified business-object property.

Syntax

```
public AgentProperty getBOSpecificProperty(String propName);
```

Parameters

propName   Is the name of the business-object property to retrieve.

Return values

An `AgentProperty` object that contains the specified business-object property, or `null` if no business-object property of that name exists.

Notes

The `getBOSpecificProperty()` method retrieves the *propName* business-object property from the ODA-runtime memory. Business Object Wizard saves these properties into memory after the user specifies their values in the BO Properties dialog box (part of Step 5). This method returns the specified business-object property as an `AgentProperty` object. You can obtain information about the property by accessing the object's member variables.

See also

getAllBOSpecificProperties(), getBOSpecificProps()

---

getBOSpecificProps()

Sends the specified business-object properties to the BO Properties dialog box for user input.

Syntax

```
public Hashtable getBOSpecificProps(AgentProperty[] properties,
   String titleBarText);
public Hashtable getBOSpecificProps(AgentProperty[] properties,
   String titleBarText, String propGridText);
```

Parameters

properties   Is an array of business-object properties, each property in an `AgentProperty` object.

titleBarText   Is text to display in the title bar of the BO Properties dialog box.

| *propGridText* | Is text to display in a text area above the property grid of the BO Properties dialog box. |

Return values

A Java `Hashtable` object of business-object properties (as `AgentProperty` objects) keyed on the property name.

Exceptions

ODKInvalidPropException
Thrown if the property is invalid—for example, if it does not have a name.

XMLException Thrown if the XML conversion of the properties failed.

Notes

The `getBOSpecificProps()` method sends the *properties* array of business-object properties to Business Object Wizard, who displays them in the BO Properties dialog box. From this dialog box, the user can enter or change these property values. Before calling the `getBOSpecificProps()` method, you must instantiate and initialize agent-property (`AgentProperty`) objects for each business-object property and store each property in the *properties* business-object-property array. The `getBOSpecificProps()` method passes this business-object-property array to its caller, Business Object Wizard.

Once the user has set the business-object properties from the BO Properties dialog box, Business Object Wizard saves these user-specified properties in a `java.util.Hashtable` object and the ODA-runtime memory. Within the ODA, you can obtain the user-initialized properties in either of the following ways:

*   From ODA-runtime memory

    Use the `getBOSpecificProperty()` or `getAllBOSpecificProperties()` method in the `ODKUtility` class. The user-initialized values for the property are in the `allValues` member variable of its agent-property (`AgentProperty`) object.
*   From the `Hashtable` object that `getBOSpecificProps()` returns

    Use the methods of the `Hashtable` object to obtain the agent properties.

For more information on how to use `getBOSpecificProps()`, see "Requesting business-object properties" on page 119.

See also

getAllBOSpecificProperties(), getBOSpecificProperty()

---

getClientFile()

Requests that Business Object Wizard retrieve a specified file.

Syntax

```
public byte[] getClientFile(String srcNodePath, ODKAgentBase2 ODAobj);
```

Parameters

| *srcNodePath* | Is the source-node path of the file to request from Business Object Wizard. |

	Is the ODA (`ODKAgentBase2`) object, which is used to verify that the
ODAobj	ODA is authorized to perform the operation; that is, that the ODA generates file content.

Return values

The contents of the specified operating-system file, as a byte array.

Exceptions

UnsupportedContentException
> Thrown if the ODA does *not* support generation of file content; that is, it does not implement the `IGeneratesBinFiles` interface.

Java.io.IOException
> Thrown if an error occurs during file retrieval, for example the file was not found.

Notes

The `getClientFile()` method requests that Business Object Wizard return the contents of the operating-system file that *srcNodePath* identifies. This *srcNodePath* path takes the following form:

`fileNodePath:fileLocation`

where:

- *fileNodePath* is the colon (:) separated list of source-node names for the node that is associated with the file. For example, `Apollo:Vulso:Flavius.xml`
- *fileLocation* is the full operating-system path to the file. For example, `C:\temp\XMLFiles\Flavius.xml`

Use the `getClientFile()` method to access an associated file for objects that source nodes represent. If a source node can have a file associated with it, then the ability to interpret the file's source-node path and to read the contents of this file is needed at *both* of the following points:

- During source-node generation, the `getTreeNodes()` method must be able to "discover" a child node that is in a file.
- During content generation, the method that generates content must be able to access information in nodes that are in a file.

For more information, see "Reading files for source data" on page 134.

---

getMsg()

Retrieves a message from the ODA message file.

Syntax

```
public String getMsg(int msgNum, int msgType);
public String getMsg(int msgNum, int msgType, msgParameters);
public String getMsg(int msgNum, int msgType, Vector paramArray);
```

Parameters

msgNum	Specifies the message number from the message file.
msgParameters	Is an optional list of up to three `String` parameter values, each corresponding to a parameter in the message list.

msgType	Is the type of message, specified as one of the following message-type constants:

ODKConstant.XRD_FATAL
ODKConstant.XRD_ERROR
ODKConstant.XRD_URGENTWARNING
ODKConstant.XRD_WARNING
ODKConstant.XRD_INFO

paramArray	Is an optional list of parameters, as a Java `Vector`, to be inserted in the message's parameters.

Exceptions

`IllegalArgumentException`
    Thrown if the *msgType* argument is not valid.

Return values

A `String` that contains the text associated with the specified message number. If message parameters have been provided, these values have been inserted as appropriate into the message. If *msgNum* is not valid, the method returns `null`.

Notes

The `getMsg()` method retrieves a message from a message file. It identifies the name of this file from the `MessageFile` startup property, which the ODK automatically includes with the ODA startup properties. The `getMsg()` method provides the following forms:

- The first form retrieves a message with the specified message number (*msgNum*) from the ODA message file.
- The second form also retrieves the message with the specified message number (*msgNum*) from the ODA message file. It also provides the ability to specify up to three `String` message parameters (*msgParameters*) to be inserted in the message before retrieving it.
- The third form also sends a message from the ODA message file and provides message parameters. However, with this form you can send the message parameters as elements in a Java `Vector`, *paramArray*.

For information on ODA message files, see "Message files" on page 153. For information on message parameters, see "Using parameter values" on page 156.

See also

trace()

---

getODKUtility()

Returns a handle to the singleton `ODKUtility` object.

Syntax

```
public static ODKUtility getODKUtility();
```

Parameters

None.

Return values

A handle to an `ODKUtility` object.

Notes

The `getODKUtility()` method provides access within the ODA code to the utilities in the `ODKUtility` class. You must use `getODKUtility()` to obtain a handle to the singleton object of this class *before* you access the `ODKUtility` methods.

Note: The call to `getODKUtility()` is often performed in the `getAgentProperties()` method. For more information, see "Obtaining the handle to the ODKUtility object" on page 102.

See also

getAgentProperties()

---

sendMsg()

Displays a user-response dialog box, which includes a message and requires a response from the user.

Syntax

```
public int sendMsg(String msg, int dialogFlags);
```

Parameters

msg   Is the message to display in the user-response dialog box.

dialogFlags   Is a set of flags to indicate the buttons and icons to display as part of the user-response dialog box. Indicate these buttons and icons as a mask of the user-response-dialog constants shown in Table 105.

Return values

An integer that indicates the button that the user has clicked to terminate the user-reponse dialog box. Compare this integer value with the following user-response constants:

`ODKConstant.ODK_OK`
              The user selected the OK button.

`ODKConstant.ODK_CANCEL`
              The user selected the Cancel button.

`ODKConstant.ODK_RETRY`
              The user selected the Retry button.

`ODKConstant.ODK_IGNORE`
              The user selected the Ignore button.

`ODKConstant.ODK_ABORT`
              The user selected the Abort button.

`ODKConstant.ODK_YES`
              The user selected the Yes button.

`ODKConstant.ODK_NO`
              The user selected the No button.

ODKConstant.ODK_CLOSE
> The user selected the Close button.

ODKConstant.ODK_HELP
> The user selected the Help button.

Notes

The `sendMsg()` method sends a request to Business Object Wizard to display a user-response dialog box to the user. You specify the following components of this user-response dialog box:

- The *msg* string contains text to indicate the condition, question, or information you need the user to see.
- A *dialogFlags* mask contains features that describe the appearance of the user-response dialog box, as follows:
  - The buttons to display

    The user clicks one of these buttons to terminate the user-response dialog box. You specify these buttons with the dialog-button constants, in the "Buttons to display" section of Table 105.
  - The icon to display

    The icon determines the type of user-response dialog box to display. You specify the dialog box type with one of the dialog-icon constants, in the "dialog box icon to display" section of Table 105.

Table 105. Display appearance of the user-response dialog box

Appearance of user-response dialog box	ODKConstant user-response-dialog constant
Buttons to display:	
OK	MSG_OK
OK, CANCEL	MSG_OKCANCEL
RETRY, CANCEL	MSG_RETRYCANCEL
RETRY, IGNORE, ABORT	MSG_ABORTRETRYIGNORE
YES, NO	MSG_YESNO
YES, NO, CANCEL	MSG_YESNOCANCEL
dialog box icon to display:	
Error icon	MSG_ERROR
Critical-error icon	MSG_CRITICALERROR
Warning icon	MSG_WARNING
Information icon	MSG_INFORMATION
Question-mark icon	MSG_QUESTION

Note: All user-response-dialog constants in Table 105 are defined in the `ODKConstant` interface.

To specify the *dialogFlags* argument, create a mask of the user-response-dialog constants that describe the appearance of your user-response dialog box. For example, the following call to `sendMsg()` creates a user-response dialog box that displays an error icon as well as the buttons Retry and Cancel:

```
String msg = new String(bdkUtil.getMsg(1002, ODKConstant.XRD_ERROR, params));
bdkUtil.sendMsg(msg, ODKConstant.MSG_RETRYCANCEL | ODKConstant.MSG_ERROR);
```

See also

sendStatusMsg()

sendStatusMsg()

Displays a message to the user.

Syntax

```
public void sendStatusMsg(String msg);
```

Parameters

msg   Is the message to send to the user.

Return values

None.

See also

sendMsg()

---

trace()

Writes a message to the trace file.

Syntax

```
public void trace(int level, int msgType, String message);

public void trace(int level, int msgNum, int msgType);
public void trace(int level, int msgNum, int msgType, msgParameters);
public void trace(int level, int msgNum, int msgType, Vector paramArray);

public void trace(int level, int msgType, BusObjDef boDef);

public void trace(int level, int msgType, AgentProperty[] properties,
    String foreword);
```

Parameters

boDef	Is the business object definition to be written to the trace file.
foreword	Is a `String` that clarifies the message before the *properties* property array—for example, "These are the properties for the Object Discovery Agent".
level	Is the trace level, specified as one of the following trace-level constants:

```
ODKConstant.TRACELEVEL0
ODKConstant.TRACELEVEL1
ODKConstant.TRACELEVEL2
ODKConstant.TRACELEVEL3
ODKConstant.TRACELEVEL4
ODKConstant.TRACELEVEL5
```

message	Is the `String` message to be written to the trace file.
msgNum	Specifies the message number in the message file.
msgParameters	Is an optional list of up to three `String` parameter values, each corresponding to a parameter in the message list.
msgType	Is the type of message, specified as one of the following message-type constants:

```
                    ODKConstant.XRD_FATAL
                    ODKConstant.XRD_ERROR
                    ODKConstant.XRD_URGENTWARNING
                    ODKConstant.XRD_WARNING
                    ODKConstant.XRD_INFO
                    ODKConstant.XRD_TRACE
```

paramArray   A vector of parameters to be inserted in the message.

properties   Is an array of agent-property (`AgentProperty`) objects to be written to the trace file.

Return values

None.

Exceptions

`IllegalArgumentException`

Thrown if the *properties* argument is `null` or the *msgType* argument is invalid.

Notes

The `trace()` method sends the specified information to the trace file when the trace *level* is less than or equal to the system trace level. The system trace level is set through the `TraceLevel` configuration property, which Business Object Wizard automatically includes in the ODA configuration properties. A trace *level* of zero (0) activates error logging; that is, `trace()` sends an error message to the trace file. The non-zero trace levels, shown in Table 106, activate tracing; that is, `trace()` sends a trace message to the trace file.

Table 106. Trace levels for an ODA

Trace level	Description	Trace-level constant
0	Log an error message.	TRACELEVEL0
1	Trace whenever a method is entered. Usually provides status messages and key information for each business object definition.	TRACELEVEL1
2	Trace the agent properties and the values received.	TRACELEVEL2
3	Trace the name of the business object definition. Usually provides the business-object properties and the values received.	TRACELEVEL3
4	Trace a message whenever a method is entered and exited. Record the spawning of all threads.	TRACELEVEL4
5	Indicate the ODA initialization. Provide the values for all agent properties retrieved, a detailed status of each thread that the ODA has spawned, and a dump of the business object definition.	TRACELEVEL5

The user establishes the name of the ODA's trace destination through the `TraceFileName` configuration property, which the ODK automatically includes in the ODA startup properties. Therefore, tracing cannot begin until after the `init()` method (which receives initialized startup properties) starts.

The `trace()` method provides the following forms:

- The first four forms send a text message to the trace file:

- The first form sends the specified text *message* to the trace file.
- The second form sends the message with the specified message number (*msgNum*) from the ODA message file.
- The third form also sends the message with the specified message number (*msgNum*) from the ODA message file. It also provides the ability to send up to three `String` message parameters (*msgParameters*) to be inserted in the message before sending the message to the trace destination.
- The fourth form also sends a message from the ODA message file and provides message parameters. However, with this form you can send the message parameters as elements in a Java `Vector`, *paramArray*.

For information on ODA message files, see "Message files" on page 153. For information on message parameters, see "Using parameter values" on page 156.

- The fifth form sends a dump of a business object definition to the trace file. This dump is formatted in the format of the `repos_copy` utility and has the following basic format:

```
[BusinessObjectDefinition]
Name=busObjName
AppSpecificInfo=business-object-level application-specific information
[Attribute]
Name=attribute1
Type=attribute type
Cardinality=n or 1
AppSpecificInfo=attribute-level application-specific information
other attribute properties
[End]
...
```

- The sixth form sends a dump of the specified agent *properties* to the trace file. This dump. The *forward* argument provides introductory text that clarifies the message.

See also

getMsg()

Deprecated Methods

Some methods in the `ODKUtility` class were supported in earlier versions but are no longer supported. These *deprecated methods* will not generate errors, but IBM recommends that you avoid their use and migrate existing code to the new methods. The deprecated methods might be removed in a future release.

Table 107 lists the deprecated methods for the `ODKUtility` class. If you are writing a new ODA (not modifying an existing ODA), you can ignore this section.

Table 107. Deprecated methods of the ODKUtility class

Deprecated method	Replacement
All methods that support filtering: • filterData() • getFilter() • setFilter()	An ODA still supports filtering at the user level, but not at the programmatic level. For more information, see "Using a filter" on page 78. At the programmatic level, the search-pattern feature provides the ability to reduce the number of child nodes for a particular parent node. For more information, see "Implementing the search-pattern feature" on page 114.

Chapter 25. TreeNode class

The Object Discovery Agent Development Kit (ODK) API provides the `TreeNode` class to represent tree nodes. The Object Discovery Agent (ODA) generates an array of tree nodes so that Business Object Wizard can display a hierarchy of source nodes to the user. The user navigates the nodes of this source-node hierarchy to select the objects whose business object definition the ODA is to generate.

The `TreeNode` class defines the following:
- "Member variables"
- "Method" on page 269

The `TreeNode` class implements the `ODKConstant` interface. Therefore, all constants defined in `ODKConstant` are available to a `TreeNode` object. For a list of constants the `ODKConstant` interface defines, see Chapter 22, "ODKConstant interface," on page 249.

Member variables

Table 108 summarizes the member variables of the `TreeNode` class.

Table 108. Member variables of the TreeNode class.

Member variable	Description	Page
description	Contains a description of the tree node.	267
isExpandable	Specifies whether the tree node is expandable; that is, whether there are elements below the current level.	268
isGeneratable	Specifies whether the tree node is generatable; that is, whether the node can be converted to a business object definition.	268
name	Contains the name of the tree node.	268
nodes	Contains the expanded hierarchy of tree nodes.	268
polymorphicNature	Defines the node's nature; that is whether it is "normal" (expandable or a leaf) or "file".	269

description

Contains a description of the tree node.

Type
`public String description`

Notes
The `description` member variable displays in the Description column of the Select Source dialog box.

isExpandable

Specifies whether the tree node is expandable; that is, whether there are nodes below the current level.

Type
```
public boolean isExpandable
```

Notes
The `isExpandable` member variable indicates whether a node is expandable, as Table 109 shows.

Table 109. Types of nodes

Type of node	Description	Value of isExpandable
Expandable	Node has child nodes	`true`
Leaf (terminating)	Node does not have child nodes but is the terminating point of a branch of the source-node hierarchy	`false`

Only normal-nature nodes (nodes with their `polymorphicNature` member variable set to `NODE_NATURE_NORMAL`) can have `isExpandable` set to `true`.

isGeneratable

Specifies whether the tree node is generatable; that is, whether the user can select this node as one for which the ODA generates content.

Type
```
public boolean isGeneratable
```

name

Contains the name of the tree node.

Type
```
public String name
```

Notes
The `name` member variable displays in the Name column of the Select Source dialog box.

nodes

Contains the expanded hierarchy of child tree nodes.

Type
```
public TreeNode[] nodes
```

Notes
The `nodes` member variable contains an array of `TreeNode` objects, one object for each of this parent node's children. A child node can, in turn, contain child nodes (grandchildren of this parent node). This member variable is only used if the node is expandable (not a leaf); that is, if the `isExpandable` member variable is `true`.

polymorphicNature

Indicates the valid actions the user can take on the tree node.

Type
```
public int polymorphicNature
```

Notes
The `polymorphicNature` member variable determines what actions the user can take on the node when it displays in the Select Source dialog box of Business Object Wizard. This variable contains an integer node-nature constant to indicate the nature of the tree node. These node-nature constants are defined in the `ODKConstant` interface, as Table 110 shows.

Table 110. Nature of tree nodes

Nature of tree node	Description	Node-nature constant
Normal	The user can take either of the following actions: • The user can select the node, if the node is a leaf (terminating) node. Only leaf nodes can be selected for generation into content. • The user can expand the node to see more nodes. Business Object Wizard displays a plus (+) to the left of an expandable node name.	NODE_NATURE_NORMAL
File	The user can associate a file from the local file system with the node. Business Object Wizard activates the Associate files menu item in the context menu that displays when the user right-clicks on the node name. This menu item opens a window for browsing system files. From this window, the user can select which file to associate with the node. For a tree node that has a file node nature, the ODA can use the `getClientFile()` method (defined in the `ODKUtility` class) to obtain the user-selected file's contents.	NODE_NATURE_FILE

Note: Because the `TreeNode` class implements the `ODKConstant` interface, the node-nature constants are available to the `polymorphicNature` member variable without being qualified with the `ODKConstant` name.

For more information on node natures, see "Constructing the tree nodes" on page 115.

Method

Table 111 summarizes the method of the `TreeNode` class.

Table 111. Member method of the TreeNode class.

Member method	Description	Page
TreeNode()	Creates a tree-node object.	page 267

TreeNode()

Creates a tree-node object.

Syntax

```
public TreeNode(String name, String desc, boolean isGen, boolean isExp);

public TreeNode(String name, String desc, boolean isGen, boolean isExp,
    TreeNode[] treeNodes);

public TreeNode(String name, String desc, boolean isGen, boolean isExp,
    TreeNode[] treeNodes, int nodeNature);
```

Parameters

desc	Specifies the description of the node; the value of this parameter initializes the `description` member variable ("description" on page 267).
isGen	Specifies whether the node is "generatable" (that is, whether the node can be converted to a business object definition); the value of this parameter initializes the `isGeneratable` member variable ("isGeneratable" on page 268).
isExp	Specifies whether the node is expandable (that is, whether the node is or is not a leaf); the value of this parameter initializes the `isExpandable` member variable ("isExpandable" on page 268).
name	Specifies the name of the node; the value of this parameter initializes the `name` member variable ("name" on page 268).

nodeNature	Indicates the nature of the node, as one of the following node-nature constants:
	`ODKConstant.NODE_NATURE_FILE`
	`ODKConstant.NODE_NATURE_NORMAL`

treeNodes	Specifies the fully expanded hierarchy of nodes; the value of this parameter initializes the `nodes` member variable ("nodes" on page 268).

Return values

A newly instantiated `TreeNode` object.

Notes

The `TreeNode()` method provides the following forms to instantiate a tree node:

- The first form of the constructor allows you to specify the name and description of the tree node, as well as whether it is generatable or expandable. In this form, the child-nodes array (the `nodes` member variable) is initialized to null and the node nature (the `polymorphicNature` member variable) is initialized to "normal". Use this form to initialize a leaf node.

- The second form of the constructor allows you to specify the child-nodes array (in addition to the values that the first form specifies). In this form, the node nature is initialized to "normal". Use this form to initialize an expandable node.

- The third form of the constructor allows you to specify the node nature (in addition to the values that the first and second forms specify). Use this form to initialize a file-nature node.

For more information, see "Constructing the tree nodes" on page 115.

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Burlingame Laboratory Director
IBM Burlingame Laboratory
577 Airport Blvd., Suite 800

Burlingame, CA 94010
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not necessarily tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

IBM
the IBM logo
AIX
CrossWorlds
DB2
DB2 Universal Database
Domino
Lotus
Lotus Notes
MQIntegrator
MQSeries
Tivoli
WebSphere

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

MMX, Pentium, and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

System Manager includes software developed by the Eclipse Project (http://www.eclipse.org/) IBM WebSphere InterChange Server Express v4.3, IBM WebSphere Business Integration Toolset v4.3, IBM WebSphere Business Integration Adapters v2.4, IBM WebSphere Business Integration Collaborations v4.2.

IBM WebSphere InterChange Server Express v4.3, IBM WebSphere Business Integration Toolset Toolset v4.3

WebSphere Business Integration Adapter Framework v2.4.0

Index

A

Adapter   3, 64
Adapter Development Kit (ADK)   96
Adapter framework   95
addDefaultVerbs() method   200
Agent property   140, 149
   array of   120
   cardinality   141, 142, 175
   class for   173
   conditions   174
   conditions on   141, 145, 215
   contents of   141
   creating   141, 179
   default value   141, 144, 174
   dependent   146, 227
   dependent condition   146, 147, 227
   description   141, 142, 176
   determining if value is required   141, 178
   encrypted   141, 176
   hidden   141, 176
   input condition   146, 241
   multiple values   141, 175, 177
   name   141, 178
   read-only   141, 177
   single value   145
   type   141, 142, 173, 179
AgentMetaData class   104, 165, 167, 173
   agentVersion   167
   constructor   170
   member variables   167
   method summary   169
   searchableNodes   167
   searchPatternDesc   168
   supportedContent   168
   toXml()   171
AgentMetaData() constructor   104, 246
AgentMetaData() method   170
AgentProperty class   140, 165, 173, 183
   allDefaultValues   174
   allDependencies   174
   allValidValues   174
   allValues   175
   cardinality   175
   constructor   120, 179
   copy()   180
   description   176
   isHidden   176
   isMultiple   177
   isReadOnly   177
   isRequired   178
   member variables   173
   method summary   179
   property-type constants   173
   propName   178
   type   179
   TYPE_BOOLEAN   173
   TYPE_DOUBLE   173
   TYPE_FLOAT   173
   TYPE_INTEGER   173
   TYPE_STRING   173
AgentProperty() method   120, 141, 179

agentVersion member variable (AgentMetaData)   104, 167
allDefaultValues member variable (AgentProperty)   121, 141, 174
allDependencies member variable (AgentProperty)   141, 146, 174
allDependentConditions member variable (CompleteCondition)   146, 216
allInputConditions member variable (CompleteCondition)   146, 216
allValidValues member variable (AgentProperty)   121, 141, 142, 174
allValues member variable (AgentProperty)   106, 141, 175
Application-specific business object   3, 17, 36
   application-specific information   32
   attributes in   31
   comparing to generic business objects   41
   default values in   6
   designing   29, 37, 41
   foreign key   5
   generating definitions for   64
   structure   30
Application-specific information   7, 10, 32
   example of processing   34
   for a business object   8, 30, 125, 200, 207
   for a verb   10, 212
   for an attribute   9, 125, 129
   metadata and   7, 32
   storage of   8
   suggested format   33
Attribute   4, 31
   adding   57, 125, 205
   application-specific information   9, 125, 129, 185, 191
   as part of foreign key   5, 126, 189, 193
   as part of primary key   5, 126, 189, 190, 193, 194
   cardinality   5, 126, 187, 192
   changing order of   59
   class for   125, 183
   comment for   6, 126, 188, 193
   creating   125, 185
   default value   6, 58, 126, 188, 193
   defining   125
   determining number of   201
   maximum length   6, 58, 126, 188, 194
   name of   5, 57, 125, 188, 194
   ordinal position of   202
   properties   4, 30
   relationship type   126, 189, 195
   removing from attribute list   206
   required   6
   required in triggering event   190
   retrieving   201, 202
   type   5, 57, 126, 186, 190, 191, 197
AttrTypes member variable (BusObjAttrTypes)   127, 197

B

badContent() method   221
BinaryFile member variable (ContentType)   107, 136, 220, 223, 232
BOOLEAN attribute-type constant   186, 191, 197
Business object   3

© Copyright IBM Corp. 2004

277

TYPE_BOOLEAN property-type constant   173
TYPE_DOUBLE property-type constant   173
TYPE_FLOAT property-type constant   173
TYPE_INTEGER property-type constant   173
TYPE_STRING property-type constant   173
typeOfSpecificValue member variable
  (DependentCondition)   148, 228
typeOfSpecificValue member variable (InputCondition)   147,
  242
typographic conventions   ix

U

UnsupportedContentException exception   254
UseDefaults connector configuration property   6

V

value() method   225
Verb   4, 7, 129
    adding   60, 129, 205
    application-specific information   10, 129, 212
    class for   129, 211
    creating   129, 211
    default   60, 200

Verb *(continued)*
    deleting   60, 207
    determining number of   204
    name of   60, 129, 212, 213
    retrieving   203, 204

W

Warnings   151, 251

X

XML format
    converting content type to   225
    converting ODA metadata to   171
xmlObject() method   225
XRD_ERROR message-type constant   152, 155, 251, 261, 265
XRD_FATAL message-type constant   152, 155, 251, 261, 265
XRD_INFO message-type constant   152, 155, 251, 261, 265
XRD_TRACE message-type constant   153, 155, 251, 265
XRD_UNKNOWN message-type constant   251
XRD_URGENTWARNING message-type constant   152, 155,
    251, 261, 265
XRD_WARNING message-type constant   152, 155, 251, 261,
    265