JD Edwards EnterpriseOne Tools

Business Services Development Methodology Guide

Release 9.1.x

E24219-03

July 2014

ORACLE

JD Edwards EnterpriseOne Tools Business Services Development Methodology Guide, Release 9.1.x

E24219-03

Contents

5 Creating a Business Service

6 Creating Business Services That Call Database Operations

Preface

Welcome to the *JD Edwards EnterpriseOne Tools Business Services Development Methodology Guide*.

> **Note:** This guide has been updated for JD Edwards EnterpriseOne Tools Release 9.1 Update 2. For details on documentation updates, refer to the *JD Edwards EnterpriseOne Tools Net Change for Tools Documentation Library*.

Audience

This guide is intended for developers and technical consultants who are responsible for creating and customizing JD Edwards EnterpriseOne business services.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

You can access related documents from the JD Edwards EnterpriseOne Release Documentation Overview pages on My Oracle Support. Access the main documentation overview page by searching for the document ID, which is 876932.1, or by using this link:

https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&id=876932.1

To navigate to this page from the My Oracle Support home page, click the Knowledge tab, and then click the Tools and Training menu, JD Edwards EnterpriseOne, Welcome Center, Release Information Overview.

This guide contains references to server configuration settings that JD Edwards EnterpriseOne stores in configuration files (such as jde.ini, jas.ini, jdbj.ini, jdelog.properties, and so on). Beginning with the JD Edwards EnterpriseOne Tools Release 8.97, it is highly recommended that you only access and manage these settings for the supported server types using the Server Manager program. See the *Server Manager Guide*.

Conventions

The following text conventions are used in this document:

Convention	Meaning
Bold	Indicates field values.
Italics	Indicates emphasis and JD Edwards EnterpriseOne or other book-length publication titles.
`Monospace`	Indicates a JD Edwards EnterpriseOne program, other code example, or URL.

1

Introduction to JD Edwards EnterpriseOne Tools Business Services Development Methodology

This chapter contains the following topics:

1.1 JD Edwards EnterpriseOne Tools Business Services Development Methodology Overview

The guide provides rules, best practices, example code pieces, and steps that you can follow to create business services that enable interoperability between JD Edwards EnterpriseOne and other Oracle applications or third-party applications and systems. You create business services using the JD Edwards EnterpriseOne toolset and the Java programming language.

Rules are guidelines that you must follow when creating or customizing JD Edwards EnterpriseOne business services. Although the JD Edwards EnterpriseOne toolset does not enforce rules, these are mandatory guidelines that you must follow to accomplish the desired results and to meet specified standards.

Best practices are guidelines that you should follow when creating or customizing JD Edwards EnterpriseOne business services. These are guidelines, which are not mandatory, that help you make good design decisions.

This guide provides an overview of business services and information for creating and modifying business services.

This guide does not preclude the use of other standard development methodologies.

Important: Oracle reserves the right to reorganize the business services foundation packages (jar files) for tools release upgrades. If you are planning to upgrade your system, test your custom objects and modify them as appropriate to ensure your code will continue to work as intended. You cannot upgrade custom business service objects after you install a tools release upgrade.

1.2 JD Edwards EnterpriseOne Tools Business Services Development Methodology Implementation

The *JD Edwards EnterpriseOne Tools Business Services Development Guide* provides concepts and information for creating business services. The *JD Edwards EnterpriseOne Tools Business Services Development Methodology Guide* supports the *JD Edwards EnterpriseOne Tools Business Services Development Guide* by providing naming conventions, best practices, guidelines, and other information for developing business services. Use the *JD Edwards EnterpriseOne Tools Business Services Development Methodology Guide* in conjunction with the *JD Edwards EnterpriseOne Tools Business Services Development Guide* if you are developing business services.

See the *JD Edwards EnterpriseOne Tools Business Services Development Guide*

2

Understanding Business Services

This chapter contains the following topics:

2.1 JD Edwards EnterpriseOne Business Services

JD Edwards EnterpriseOne provides interoperability with other Oracle applications and third-party systems by natively producing and consuming web services. Web services enable software applications written in various programming languages and running on various platforms to exchange information. JD Edwards EnterpriseOne exposes business services as web services. A web service is a standardized way of integrating web-based applications, and in JD Edwards EnterpriseOne, web services are referred to as published business services. Business services enable JD Edwards EnterpriseOne to expose transactions as a basic service that can expose an XML document-based interface.

2.1.1 Published Business Services

A published business service is a JD Edwards EnterpriseOne Object Management Workbench (OMW) object that represents one Java class that publishes multiple business services. When you create a web service, you identify the Java class. The published business service also contains value object classes that make up the signature for the published business service.

2.1.2 Business Services

A business service is a JD Edwards EnterpriseOne OMW object that represents one or more classes that expose public methods. Each method performs a business process. A business service also contains internal value object classes that make up the signature for the business service methods. These public methods can be called from other business service classes and published business service classes.

2.2 Development Methodology

JD Edwards EnterpriseOne provides tools to help you create business services and published business services. You access Oracle's JDeveloper from JD Edwards

EnterpriseOne OMW. You should have one business service workspace based on the JD Edwards EnterpriseOne path code in JDeveloper. This workspace should have been created when JDeveloper was launched from OMW. Each business service and published business service has its own project under the business service workspace, where you can add and modify code for business services and published business services that were created using OMW. JDeveloper provides wizards that generate Java code to help you create business services and published business services. All business services and published business services are written in the Java programming language.

The JD Edwards EnterpriseOne business services framework provides a set of foundation packages. Each foundation package contains a set of interfaces and related classes that provide building blocks that you use to create the business service or published business service. Business service classes extend the BusinessService foundation class. Business service classes call business functions and database operations. The published business service class extends the PublishedBusinessService foundation class. This class exposes public methods that represent JD Edwards EnterpriseOne business processes as web services.

The business services framework also supports business service properties. Business service properties provide flexibility in the code by enabling users to set a value without changing the code. The business service framework includes wizards that provide building blocks to help you create business function calls and database operation calls. You also can access code templates. Code templates generate skeleton code that you modify and finalize. You can use code templates to generate skeleton code for creating public and private methods for a published business service, creating public methods for a business service, formatting data, calling a business service property, and testing a published business service.

JD Edwards EnterpriseOne business service and published business service classes use value object classes. A value object is an interface to a business service or a published business service. A value object is the high-level component that contains the business data that defines a business process. Business services use internal value objects, and published business services use published value objects. Internal value objects and published value objects and their components extend the ValueObject foundation class. Published response value objects, which are used by published business services, extend the MessageValueObject foundation class and contain warning messages that are returned from business function and database operation calls.

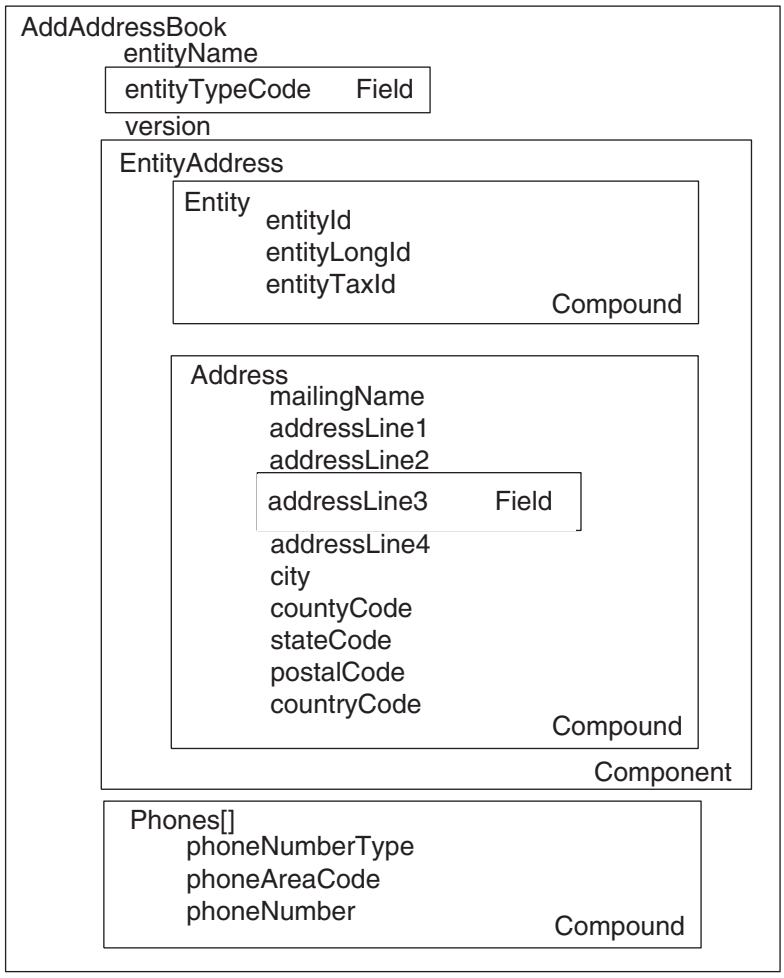
2.3 Value Objects

Value objects are a specific type of source file that holds input and output data, much like a data structure passes data. The input and output parameters of business service operations are called internal value objects. Business service internal value objects are not published interfaces. Business service operations use one internal value object for both input and output. Examples of internal value objects include InternalAddAddressBook, InternalProcessPurchaseOrder, InternalEntity, and so on.

The input and output parameters of the published business service business operations are called value objects. These parameters are the payload of the web service. A business operation defined in a published business service takes one value object as its input parameter and returns one value object as its output parameter. Examples of published business service value objects include AddAddressBook, ProcessSalesOrder, ProcessPurchaseOrder, GetCustomer, ConfirmProcessPurchaseOrder, and so on.

The structure of a value object is modeled after the business object document (BOD) defined by Open Applications Group, Inc. (OAGIS). The structure represents the hierarchy of a business process. The following example value object shows the hierarchy for AddAddressBook:

Figure 2–1 Value object structure



Value objects are made up of components, compounds, and fields.

2.3.1 Components

Components are extensible building blocks of a value object and consist of compounds and fields or just fields. Examples of components are PurchaseOrderHeader, PurchaseOrderDetail, and EntityAddress.

2.3.2 Compounds

Compounds are collections of related fields and are implemented as classes. Compounds are basic, shared building blocks. Examples of compounds are purchaseOrderKeys, supplier, and item.

2.3.3 Fields

Fields are the lowest-level elements that are defined. Components and compounds, if used, consist of fields.

2.4 Package Naming and Structure

You use JD Edwards EnterpriseOne OMW to create new JD Edwards EnterpriseOne business service and published business service objects and to access existing business service and published business service objects. When you name a business service or published business service, you must use naming conventions that are compatible with OMW. You create business service and published business service objects in OMW, and then you start JDeveloper from OMW. JDeveloper automatically creates a project for the last OMW object that you created; using JDeveloper and the Project wizard, you create projects for each OMW object that you created.

The Java package that is created for business services and published business services is determined when you create an OMW object. The following are examples of package names:

package oracle.e1.bssv.*JP010000*

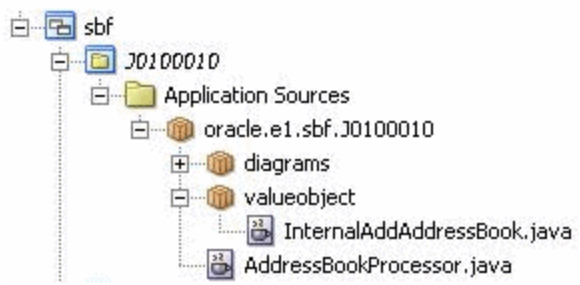
package oracle.e1.bssv.util.*J0100020*

A business service can be created in a utilities package (oracle.e1.bssv.util) if the business service provides a repeatable task that is consumed by multiple other business services. All other business services and published business services are created with the root package name (oracle.e1.bssv).

In the preceding examples, the portion of the name in italic font is the business service object name. To be compatible with OMW object names, this portion of the package name must be eight characters. The naming convention for the OMW object name is different for business service and published business service packages.

For a business service package, the OMW object name is J, system code, and numbers, where the numbers are a number that you assign to each business service; for example, J0100001, J0200002, J0100010, J0100020, J0100100, J0100110, and so on. The OMW object name must be eight characters. The following diagram shows the structure for a business service.

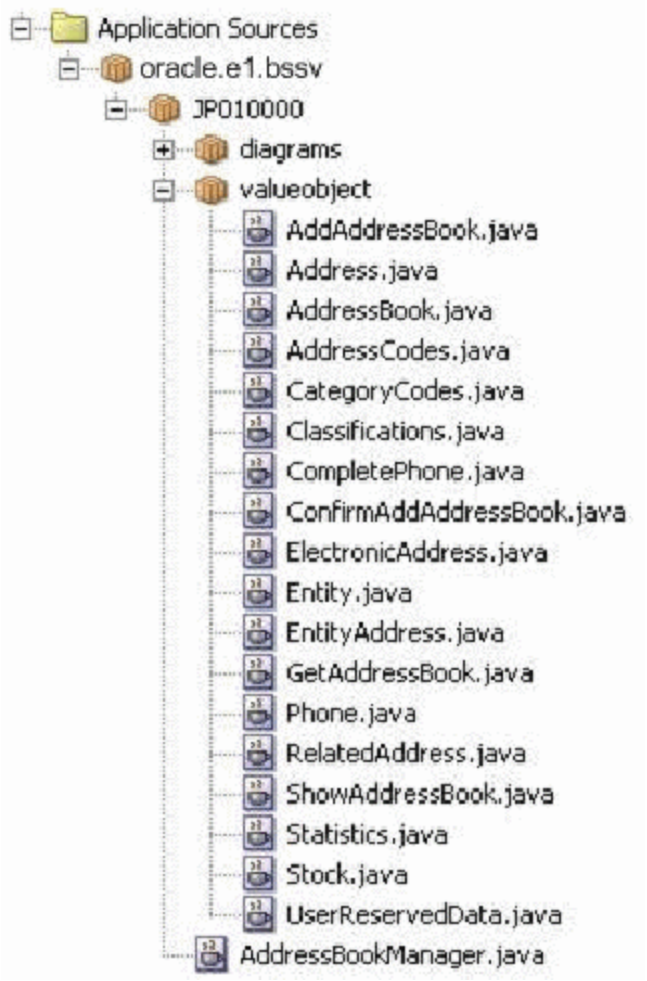
Figure 2–2 Business service package structure



For a published business service package, the OMW object name is JP, system code, and zeros (for example, JP010000). The OMW object name must be a total of eight characters. The naming standards do not preclude the creation of a second published business service per system code; however, our guideline is to create one service per system code. The naming convention for the OMW object is also part of the name of the package where the published business service class resides. Within the JDeveloper

tree structure, a published business service must be directly under the package name. For example, the published business service AddressBookManager.java can be under oracle.e1.bssv.JP010020 only; it cannot be under a subpackage of JP010020. The following diagram shows the structure for a published business service:

Figure 2–3 Published business service package structure



Each business service and published business service must have its own package name. You cannot include both a business service name and a published business service name together as one package. For example, the package name oracle.e1.bssv.JP010000.J0100020 is invalid.

2.5 Java Coding Standards

You use JDeveloper and the Java programming language to create JD Edwards EnterpriseOne business service and published business service classes that run in a J2EE environment. The business services foundation package provides classes that you extend when you write your code. The business services foundation and JDeveloper provide wizards that help you structure your code. JDeveloper enables you to set preferences for placing braces and then reformats the code to your desired style.

You use basic Java programming style conventions when you write your code. For example, instead of sprinkling literals and constant values throughout the code, you should define these values as private static final variables at the beginning of a method

or function or define them globally. Another convention is to use uppercase letters for each word. You should separate each pair of words with an underscore when naming constants, as illustrated in this code sample:

```
private static final String DEFAULT_ERROR = "c39f495121b...etc";
```

You should include meaningful comments consistently throughout your code. For easier readability when you create a Java class, order the elements in the following way:

1. Attributes
2. Constructors
3. Methods

The code that you write should check for null and empty strings, as illustrated in this example code:

```
if ((string != null) && (string.length() !=0))
                     or
if ((string == null) || (string.length() == 0))
                     or
if ((string == null) || (string.equals("")))
```

Your code should check for null objects. You can use this sample code to check for null objects:

```
if (object !=null)
{
  doSomething()
}
```

When you compare strings, use equals(). This code sample shows the correct way and the wrong way to compare strings:

```
String abc = "abc"; String def = "def";
// Correct way
if ( (abc + def).equals("abcdef") )
{
  .....
}

// Wrong way
if ( (abc + def) == "abcdef" )
{
   ......
}
```

When you create published value objects, the code should test for null objects in the set methods. This code sample shows how to test for null objects:

```
public void setCarrier(Entity carrier)
 {
   if (carrier != null)
     this.carrier = carrier;
   else
     this.carrier = new Entity();
 }
```

3

Understanding Media Object Business Services (Release 9.1 Update 2)

This chapter contains the following topics:

- Section 3.1, "JD Edwards EnterpriseOne Media Object Business Services"
- Section 3.2, "Development Methodology"

3.1 JD Edwards EnterpriseOne Media Object Business Services

JD Edwards EnterpriseOne Media Object business services provide a way to send and receive Media Object attachments to and from EnterpriseOne business services and third-party web services. Media Object business services leverage the Message Transmission Optimization Mechanism (MTOM) specification to transmit media objects as binary data in SOAP Messages. These Media Object business services are exposed in the JAX-WS web service that can be consumed by internal EnterpriseOne business services or external third-party systems. You can develop Media Object business services to perform an insert, select, or delete of media object files (PDF, documents, images, and so forth), text, and URLs in EnterpriseOne.

3.2 Development Methodology

The JD Edwards EnterpriseOne business services framework provides a set of foundation packages that can perform operations to insert, select, and delete media objects in EnterpriseOne. Media Object business service classes call Media Object operations. Media Object business services (published or internal) use the Media Object Value Object Class Wizard to create the value objects for calling the Media Object operations.

The following list contains the high level steps for developing business services that perform media object operations:

1. Create the internal media object value object using the Media Object Value Object Class Wizard in the internal business service project.

   See Creating Internal Media Object Value Objects (Release 9.1 Update 2) for more information.

2. Create a business service class using the Business Service Class Wizard in the internal business service project.

   See Creating a Business Service Class for more information.

3. Within the business service class, use the Create Media Object Call Wizard to generate the code that performs the Media Object operations.

See Calling Media Object Operations (Release 9.1 Update 2) for more information.

4. Create the published Media Object value object using the Media Object Value Object Class Wizard in the published business service project.

   See Creating a Media Object Published Value Object (Release 9.1 Update 2) for more information.

5. Create the published business service class using the Published Business Service Class Wizard in the published business service project.

   See Chapter 4, "Creating a Published Business Service" for more information.

6. Map the published value object fields to the corresponding fields in the internal value object.

   See Mappings for more information.

7. Within the published business service class, write the code to call the internal business service class created in step 2 above.

   See Calling a Media Object Business Service (Release 9.1 Update 2) for more information.

4

Creating a Published Business Service

This chapter contains the following topics:

Important: Oracle reserves the right to reorganize the business services foundation packages (jar files) for tools release upgrades. If you are planning to upgrade your system, test your custom objects and modify them as appropriate to ensure your code will continue to work as intended. You cannot upgrade custom business service objects after you install a tools release upgrade.

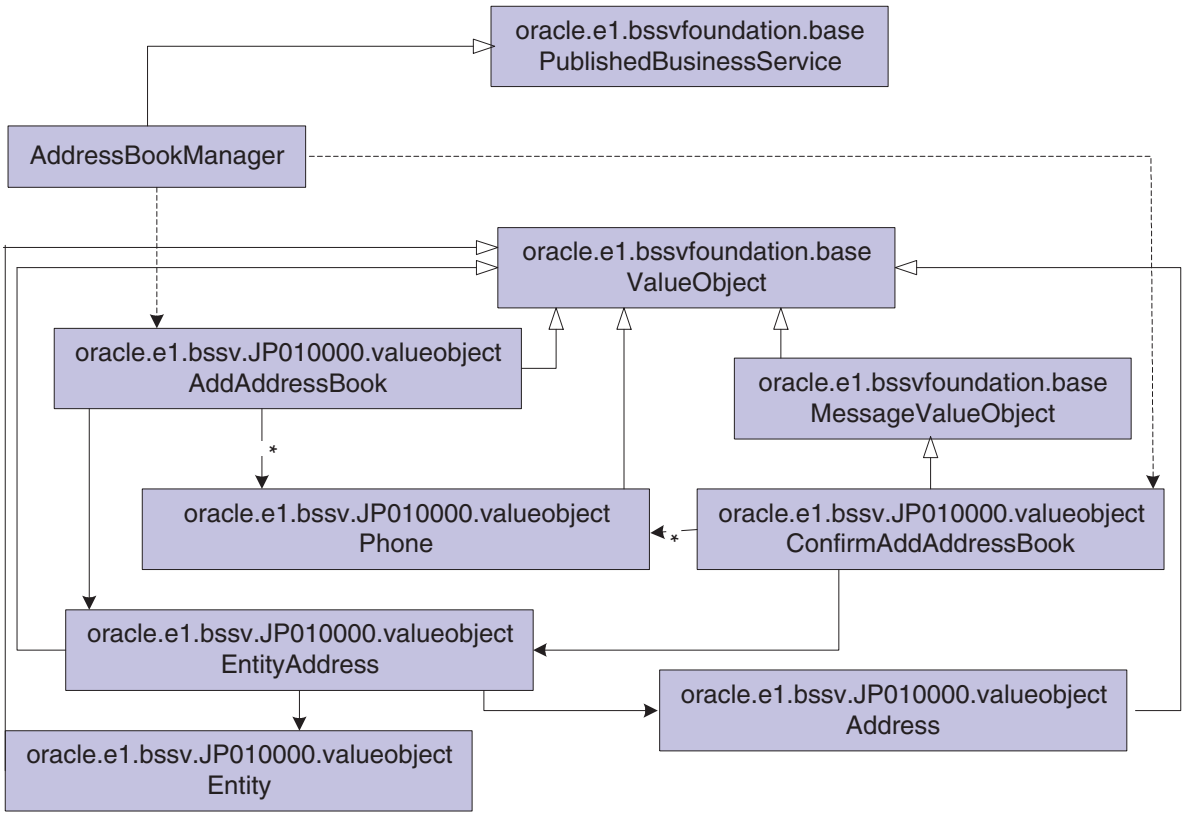
4.1 Understanding Published Business Services

A published business service gives exposure to one or more business services by providing an interface that is available to the public as a consumable web service. A published business service is a Java class that contains business service methods where the actual business logic is performed.

You use JDeveloper, JD Edwards EnterpriseOne business services framework, and the Java programming language to create published business services. The business service framework provides a set of foundation packages that helps you create published business services. Each foundation package contains a set of interfaces and related classes. All published business service classes extend from the PublishedBusinessService foundation class. Code samples are provided throughout this chapter to demonstrate the general concepts for creating a published business service. Rules and best practices are discussed for each topic, if appropriate.

The following class diagram shows the main published business service class (AddressBookManager) and the value object class (AddAddressBook) and its components:

Figure 4–1 Published business service class diagram



These features are illustrated in the published business service class diagram:

- AddressBookManager extends foundation class PublishedBusinessService.
- AddAddressBook extends ValueObject.
- ConfirmAddAddressBook extends MessageValueObject.
- All components of AddAddressBook and ConfirmAddAddressBook extend ValueObject.

4.2 Developing a Published Business Service

A published business service contains multiple Java classes, including a published business service class and value object classes. The published business service class contains public methods that are exposed to the public. These public Java methods are wrappers for business services where the actual business logic is performed.

After a business service is published, you cannot change the name and signature of the business service without affecting the consumers of that service. If you change an underlying business service that the published method exposes, then you change the signature and contract of the published business service. Because JD Edwards EnterpriseOne is not providing a merge of new and existing software, when you update or upgrade your system, any business services that you have changed will be overwritten by new JD Edwards EnterpriseOne code. If you need to change an underlying business service, copy the existing business service into a new Object Management Workbench (OMW) object and name the OMW object as a version of the

original business service. You also create a new published business service method that includes the versioned business service.

4.2.1 Creating a Transaction in a Published Business Service

A published business service class has a public method and a protected method that work together to expose a web service operation. The public method is exposed as the web service and acts as a wrapper method that passes a null to the context and connection parameters of the protected method. By passing null for these objects, the wrapper method identifies that this is the outermost call; that is, this is the web service. When a null context is passed, the protected method creates a context object that contains either a default manual connection or an auto commit connection for processing a transaction. Two methods with the same context name but different parameters exist. The context object that is used depends on whether you initiate a manual commit or auto commit connection. After the context object is created, the protected method starts processing by calling startPublishedMethod. All calls after startPublishedMethod are tied together by the context object. By passing null for the connection object, the wrapper method indicates that the default connection should be used for all operations. If a JD Edwards EnterpriseOne customer needs to extend a published business service by creating their own published business service and calling an existing JD Edwards EnterpriseOne published business service, the connection must be passed and it would not be null.

See Auto Commit.

The context object and the connection object are passed to the business service method where the business function call is made. After returning from the business service, the context object is sent to finishPublishedMethod to commit the default transaction in the case of manual commit, and then to the close method to close and clean up all outstanding connections.

This code sample shows creating and passing the context object:

```
 public ConfirmAddAddressBook addAddressBook(AddAddressBook vo)
throws BusinessServiceException {
       return (addAddressBook(null,null, vo));
   }
    protected ConfirmAddAddressBook addAddressBook
                                (IContext context,IConnection
                                connection, AddAddressBook vo) throws
                                BusinessServiceException{
       //perform all work within try block, finally will clean up any
       //connections
       try {
           // call start published method, passing null,
           //will return context object so BSFN can be called later
           //used to indicate transaction boundary as well as used for
           //logging
           //RI: Start Implicit Transaction
           context = startPublishedMethod(context,
                "addAddressBook");
           // create a new internal vo based on the external vo passed
           InternalAddAddressBook internalVO= new
              InternalAddAddressBook();
           messages.addMessages(vo.mapFromPublished(context,
              internalVO));
          // start business service addAddressBook passing context
            and internal VO
           //RI: Published Business Service Calling Business Service
```

```
        E1MessageList messages = AddressBookProcessor.addAddressBook
        (context,connection,internalVO);
        // Published Business Service will send either warnings in
           the Confirm Value Object or throw a published business
           service Exception.
        //a return status of 2 is an error, throw the exception
        if (messages.hasErrors()) {
            // get the string representation of all the messages
             //RI: Error Handling
            String error = messages.getMessagesAsString());
            // Throw new BusinessServiceException(error);
            throw new BusinessServiceException(error,context);
        }
        // exception was not thrown, so create the confirm VO from
           internal VO
        ConfirmAddAddressBook confirmVO = new ConfirmAddAddressBook
          (internalVO);
        confirmVO.setE1MessageList(messages);
        // call finish published method, passing the context
        //to commit transaction(if no exceptions), as well as use
        //in logging
        finishPublishedMethod(context, "addAddressBook");
        // return confirm VO, filled with return values and messages
        return confirmVO;
    } finally {
        //clean up any remaining connections and resources.
        close(context,"addAddressBook");
    }
```

4.3 Managing Published Business Service Components

Naming conventions and concepts for creating published business service classes, methods, value objects, and fields are discussed in the following sections. Code samples are provided as examples for you to follow. Rules and best practices are also discussed where appropriate.

4.3.1 Published Business Service Class Names

The naming convention for a published business service class is the description name of the system code with Manager added to the end of the name; for example, AddressBookManager. Other examples of published business service class names are ProcurementManager and SalesOrderManager.

This code sample shows the naming convention for a published business service class:

```
public class AddressBookManager extends
PublishedBusinessService {
....
}
```

4.3.2 Published Business Service Method Names

The naming convention for a published business service method is to use a functional description prefaced by an action verb that describes the processing that will occur. For example, for a published business service method that adds an address book record to the database, an appropriate published business service method name is

addAddressBook. The business service public method uses the same name as the published business service method.

This code sample shows the naming convention for a published business service method:

```
public ConfirmAddAddressBook addAddressBook
(AddAddressBook vo) throws BusinessServiceException{
...
}
```

4.3.3 Published Business Service Value Object Names

The input and output parameters of the published business service are called published value objects. The published business service method takes one value object as its input parameter and returns one value object as its output parameter.

This code sample shows the naming convention for published value objects:

```
    public ConfirmAddAddressBook addAddressBook
(AddAddressBook vo) throws BusinessServiceException {
...
}
```

4.3.3.1 Published Business Service Variable Names

The variable name should clarify the type of data in the field or compound. For example, if multiple entity type objects exist, the class called Entity would be the data type, but ProcessPurchaseOrder would contain objects of type Entity called supplier and shipTo. In this example, the Entity class can be reused from the EntityProcessor utility business service.

In the following code sample, the AddAddressBook value object has three top-level field names and contains an entityAddress, which is subsequently made up of an entity with three fields and an address with ten fields:

```
public class AddAddressBook extends ValueObject implements
Serializable{
       private EntityAddress entityAddress = new EntityAddress();
       private String entityName;
       private String entityTypeCode;
       private String version;
....
}
public class EntityAddress extends ValueObject implements
Serializable {
    private Entity entity = new Entity();
    private Address address = new Address();
....
}
public class Address extends ValueObject implements Serializable{
    private String mailingName;
    private String addressLine1;
    private String addressLine2;
    private String addressLine3;
    private String addressLine4;
    private String city;
    private String countyCode;
    private String stateCode;
    private String postalCode;
```

```
    private String countryCode;
....
}
public class Entity extends ValueObject implements Serializable{
   private Integer entityId;
   private String entityLongId;
   private String entityTaxId;
....
}
```

4.3.4 Creating a Published Business Service Class

The business service foundation provides a Published Business Service wizard that helps you create published business service classes. The wizard prompts you for a published business service name, an input value object name, an output value object name, and a method name. The wizard creates a Java code structure for a published business service class that can be published as a web service. This structure contains comments and TODO: tags to help you add the code to call mapping methods and business service methods.

See "Understanding Business Services" in the *JD Edwards EnterpriseOne Tools Business Services Development Methodology Guide.*

4.3.4.1 Rules

The published business service class extends the PublishedBusinessService foundation class, and the constructor must be public. This extension provides access to the transaction methods (startPublishedMethod and finishPublishedMethod) that are used in all of the published methods of a published business service class.

This code sample shows how to extend the published business service foundation class:

```
public class AddressBookManager extends PublishedBusinessService {
     public AddressBookManager() {
     }
 ....
 }
```

4.3.5 Declaring Public Methods for a Published Business Service

Published business service classes expose public, nonstatic methods. Declaring a public method exposes it to third-party systems.

This code sample shows a published business service declaring a business service method:

```
public ConfirmAddAddressBook addAddressBook
(AddAddressBook vo) throws BusinessServiceException{
 ...
 }
```

When you use the Published Business Service wizard to create the published business service class, the wizard also creates a public and protected method. For additional methods, you can use code templates to generate Java code. The E1PM – EnterpriseOne Published Business Service Method code template generates code for both public and protected methods of a published business service class. You use a code template in the source code. After you generate code using the code template,

you press the Tab key to move through the highlighted fields to complete the generated code. The generated code contains TODO: tags that help you.

4.3.6 Creating a Published Value Object

The business service foundation provides value object wizards that help you create value object classes that follow methodology rules for published value objects. The Value Object wizard creates objects based on database tables and business views for database operations or from the data structures defined within a business function.

When the wizard generates member variables for the published value object class, it uses the description that comes from the data dictionary item in the business function data structure or from table or business view columns as the variable name. If these are not the names that you want to use in your published interface, you can change them.

This code sample shows a generated variable:

```
/**
 * Business Unit
 * An alphanumeric code that identifies a separate entity within a
 * business for which you want to track costs. For example, a
 * business unit might be a warehouse location, job, project, work
 * center, branch, or plant.

 * EnterpriseOne Key Field: false
 * EnterpriseOne Alias: MCU
 * EnterpriseOne field length: 12
 */
 private String businessUnit = null;
```

You use the standard JDeveloper wizard to generate the getter and setter methods for the variables because the Value Object wizard does not generate these methods. For web services to be generated and deployed successfully, you must use J2EE standards for naming the getter and setter methods. J2EE standards for writing a field such as private String description would be:

```
public String getDescription(){
    return description;
 }
 public void setDescription(String description){
    this.description = description;
 }
```

For Boolean fields, the pattern is slightly different. J2EE standards for writing a field such as *private Boolean isCreditExempt*; would be:

```
public Boolean isIsCreditExempt(){
    return isCreditExempt;
 }
 public void setIsCreditExempt(Boolean isCreditExempt){
    this.isCreditExempt = isCreditExempt;
 }
```

4.3.6.1 Published Value Object Structure and Data Types

The published input value object must extend the ValueObject foundation class. The published confirm or response value object contains warning messages that were

returned from the business processing and must extend the MessageValueObject foundation class. All published value objects must have a default constructor.

This table lists the valid data types for published value objects:

Valid Data Type	Usage
java.lang.String	Use for string or char fields in JD Edwards EnterpriseOne.
java.util.Calendar	Use for JDEDate or UTIME fields in JD Edwards EnterpriseOne.
java.lang.Integer	Use for MathNumeric fields defined with 0 decimals, for example, mnAddressNumber, mnShortItemNumber, and so on.
java.lang.BigDecimal	Use for MathNumeric fields defined with >0 decimals, for example, mnPurchaseUnitPrice.
java.lang.Boolean	Use for char fields specified only as true/false or 0/1 Boolean fields.

Value object classes can be reused when a business service calls a utility or for calls between business services that depend on one another—such as AddressBook and Supplier. For example, you can reuse the Entity class from the EntityProcessor utility business service by importing the class from the utility's package.

4.3.6.2 Web Service Considerations for Data Types and Variable Names

A published business service class is the foundation for creating a web service. The web services description language (WSDL) is an XML-based language that describes a web service. The WSDL describes all methods of the published business service as well as the input and output value objects for these methods. All classes that make up the highest-level value object are included in the WSDL description. For example, for the Procurement Manager web service, the operations that the WSDL exposes are processPurchaseOrder, getPurchaseOrder, and processPurchaseOrderAcknowledge. All value object classes that are associated with these operations are defined in the WSDL as well.

All classes that are used within a published business service must have a unique name, which you should consider when you reuse value objects across published business services. Member variable names within the published business service value object class must be unique if they are of different object types. For example, the hierarchy of ProcessPurchaseOrder contains two classes representing financial data—one at the header level and one at the detail level. The header and detail are represented by unique classes because they are structured differently. Because both header and detail belong under the interface ProcessPurchaseOrder, the variable name referencing these object types must be unique; for example, financial and financialDetail.

The requirement for using unique variable names applies only to classes that have the same parent value object. You are not required to use unique variable names across value object classes. For example, both ProcessPurchaseOrder and ProcessPurchaseOrderAcknowledge have a header class, but the header classes are structured differently. Both of the member variables representing these classes can use the name header because they belong to different parent value objects. Classes that can be reused, such as PurchaseOrderKey, can have the same variable name across value objects.

The following examples show uniquely named classes that have member variables that are named the same:

Type	Member Variable Name
ProcessPurchaseOrder	
PurchaseOrderHeader	header
PurchaseOrderKey	purchaseOrderKey
Integer	documentNumber
String	documentCompany
String	documentType
UserReservedData	userReservedData
String	userReservedCode
Integer	userReservedNumber
BigDecimal	userReservedAmount
Calendar	userReservedDate
PurchaseOrderFinancial	financial
PurchaseOrderDetail	detail
PurchaseOrderFinancialDetail	financialDetail

Type	Member Variable Name
ConfirmProcessPurchaseOrder	
ConfirmPurchaseOrderHeader	header
PurchaseOrderKey	purchaseOrderKey
Integer	documentNumber
String	documentCompany
String	documentType
UserReservedData	userReservedData
String	userReservedCode
Integer	userReservedNumber
BigDecimal	userReservedAmount
Calendar	userReservedDate
ConfirmPurchaseOrderFinancial	financial
ConfirmPurchaseOrderDetail	detail
ConfirmPurchaseOrderFinancialDetail	financialDetail

Type	Member Variable Name
ProcessPurchaseOrderAcknowledge	header
PurchaseOrderAcknowledgeHeader	purchaseOrderKey
PurchaseOrderKey	documentNumber
Integer	documentCompany
String	documentType
String	userReservedData
UserReservedData	userReservedCode
String	userReservedNumber
Integer	userReservedAmount
BigDecimal	userReservedDate
Calendar	financial
PurchaseOrderAcknowledgeFinancial	detail
PurchaseOrderAcknowledgeDetail	financialDetail
PurchaseOrderAcknowledgeFinancialDetail	

Type	Member Variable Name
GetPurchaseOrder	
PurchaseOrderGetHeader	purchaseOrderGetHeader
PurchaseOrderKey	purchaseOrderKey
Integer	documentNumber
String	documentCompany
String	documentType
UserReservedData	userReservedData
String	userReservedCode
Integer	userReservedNumber
BigDecimal	userReservedAmount
Calendar	userReservedDate
PurchaseOrderGetFinancial	financial
PurchaseOrderGetDetail	detail
PurchaseOrderGetFinancialDetail	financialDetail

Type	Member Variable Name
ShowPurchaseOrder	
PurchaseOrderShowHeader	header
PurchaseOrderKey	purchaseOrderKey
Integer	documentNumber
String	documentCompany
String	documentType
UserReservedData	userReservedData
String	userReservedCode
Integer	userReservedNumber
BigDecimal	userReservedAmount
Calendar	userReservedDate
PurchaseOrderShowFinancial	financial
PurchaseOrderShowDetail	detail
PurchaseOrderShowFinancialDetail	financialDetail

4.3.6.3 Rules

Follow these rules when you develop published business service value object classes:

- Implement the serialize interface for all published value objects. This facilitates exposing the published business service as a web service.
- Initialize published business service value object compound attributes. This is to prevent null pointer exceptions when the method calls accessors.
- Expose published business service value object compound collections as arrays. Collection objects such as an ArrayList cannot be exposed from a web service at this time.
- Do not change published value objects, because the change breaks the contract that was created by the original value object. This is to support backwards compatibility.
- Do not add a new field, because this breaks the original contract that was set by the value object. You must create a new version of the value object and method.
- Create response value objects that contain a complete message (more than just keys).

- Place mappings between published and internal value objects in a method in the published value object.

4.3.6.4 Published Input Value Object

This code sample illustrates the code for a published input value object class:

```
public class AddAddressBook extends ValueObject implements
Serializable{
       private EntityAddress entityAddress = new EntityAddress();
       // Compound attribute is initialized
       private String entityName; //Leaf attribute not initialized
       private String entityTypeCode;
       private String version;
   ....
   }
   public class EntityAddress extends ValueObject implements
Serializable {
       private Entity entity = new Entity();
       private Address address = new Address();
   ....
   }
   public class Address extends ValueObject implements
Serializable{
       private String mailingName;
       private String addressLine1;
       private String addressLine2;
       private String addressLine3;
       private String addressLine4;
       private String city;
       private String countyCode;
       private String stateCode;
       private String postalCode;
       private String countryCode;
   ....
   }
   public class Entity extends ValueObject implements
Serializable{
      private Integer entityId;
      private String entityLongId;
      private String entityTaxId;
   ....
   }
```

4.3.6.5 Published Response Value Object

This code sample illustrates the code for a published response value object class:

```
public class ConfirmAddAddressBook extends MessageValueObject implements
Serializable{
       private EntityAddress entityAddress = new EntityAddress();
       // Compound attribute is initialized
       private String entityName;
       //Leaf attribute not initialized
       private String entityTypeCode;
       private String version;
   ....
   }
   public class EntityAddress extends ValueObject implements Serializable {
       private Entity entity = new Entity();
```

```
    private Address address = new Address();
....
}
public class Address extends ValueObject implements Serializable{
    private String mailingName;
    private String addressLine1;
    private String addressLine2;
    private String addressLine3;
    private String addressLine4;
    private String city;
    private String countyCode;
    private String stateCode;
    private String postalCode;
    private String countryCode;
....
}
public class Entity extends ValueObject implements Serializable{
   private Integer entityId;
   private String entityLongId;
   private String entityTaxId;
....
}
```

4.3.6.6 Mappings

The mapping between the published value object and the internal value object takes place in the published value object. You create a method for mapping fields from the published value object to the corresponding fields of the internal value object.

If you call the Formatter utility or a business service utility when mapping data from published to internal value objects, Oracle recommends that you create a method named mapFromPublished that returns an E1MessageList. The mapFromPublished method takes at a minimum the internal value object as a parameter. This method holds all of the mappings between the published value object and the internal value object. If a message could be returned to the published business service, you should create a method for mappings. You should always create a method to return messages when you call a business service utility or the Formatter utility during mapping. If no messages would be returned from mappings, you can have the method return void.

This code sample uses the mapFromPublished method and returns an E1MessageList:

```
  public E1MessageList mapFromPublished(IContext context, RI_InternalAdd
AddressBook vo){
     E1MessageList messages = new E1MessageList();
     //set all internal VO attributes based on external VO passed in

     vo.setSzMailingName(this.getEntityAddress().getAddress().
getMailingName());
     vo.setSzAddressLine1(this.getEntityAddress().getAddress().
getAddressLine1());
     vo.setSzAddressLine2(this.getEntityAddress().getAddress().
getAddressLine2());
     vo.setSzAddressLine3(this.getEntityAddress().getAddress().
getAddressLine3());
     vo.setSzAddressLine4(this.getEntityAddress().getAddress().
getAddressLine4());
     vo.setSzCity(this.getEntityAddress().getAddress().getCity());
     vo.setSzState(this.getEntityAddress().getAddress().getStateCode());
     vo.setSzCountry(this.getEntityAddress().getAddress().getCountryCode());
     vo.setSzCounty(this.getEntityAddress().getAddress().getCountyCode());
```

```
     vo.setSzPostalCode(this.getEntityAddress().getAddress().
getPostalCode());
     vo.setMnAddressBookNumber(this.getEntityAddress().getEntity().
getEntityId());
     vo.setSzLongAddressNumber(this.getEntityAddress().getEntity().
getEntityLongId());
     vo.setSzTaxId(this.getEntityAddress().getEntity().getEntityTaxId());
     vo.setSzAlphaName(this.getEntityName());
     vo.setSzSearchType(this.getEntityTypeCode());
     vo.setSzVersion(this.getVersion());
     vo.setJdDateEffective(this.getEffectiveDate());
     //format business unit coming from published vo.
     String formattedMCU = null;
     String bu = this.getBusinessUnit();
     if(bu!=null && !bu.equals("")){
        try {
          formattedMCU = context.getBSSVDataFormatter().format(this.
getBusinessUnit(),"MCU");
          vo.setSzBusinessUnit(formattedMCU);
        }
        catch (BSSVDataFormatterException e) {
          context.getBSSVLogger().app(context,"Error when formatting Business
 Unit.",null,vo,e);
          //Create new E1 Message with error from exception
          messages.addMessage(new E1Message(context, "002FIS",this.
getBusinessUnit()));
        }
      }

        //phones loop through array
        //new arraylist
        RI_Phone phones[] = this.getPhones();
       if (this.getPhones()!=null){
           ArrayList phonesList = new ArrayList();
           for(int i=0; i<phones.length; i++){
              //create internal phone and add to array list
```

If an E1MessageList would never be returned, and the mappings are from internal to published response value objects, you can use an overloaded constructor for the internal value object mappings. If you have no calls to utilities or formatters, mapping can be done in the constructor. If the mappings are from published to internal value objects and no messages are being returned, you should create a mapFromPublished method that returns void.

This code sample uses an overloaded constructor for mapping:

```
     public ShowAddressBook(InternalGetAddressBook internalVO){
       if(internalVO.getQueryResults()!=null){
          this.setNumberRowsReturned(internalVO.getQueryResults().size());
          this.addressBook = new AddressBook[internalVO.getQueryResults().
size()];
          for(int i = 0;i<internalVO.getQueryResults().size();i++){
             AddressBook ab = new AddressBook(internalVO.getQueryResults(i));
             this.setAddressBook(i,ab);
          }
       }
   }
```

4.3.6.7 Data Type Transformation

When you map data between published and internal value objects, data type transformations may be required. The business service foundation provides methods and constructors that format data and transform data types. Data type transformations that are done in the mappings are:

- Integer to and from MathNumeric
- BigDecimal to and from MathNumeric
- Boolean to and from String

4.3.6.8 Integer to and from MathNumeric and BigDecimal to and from MathNumeric

Mapping between published integer fields and internal math numeric fields requires a data type transformation. You use the set methods of the internal value object to make these transformations. An overloaded method takes either an integer or a math numeric data type when setting the field value.

The same rule applies to mapping between big decimal and math numeric fields. The business service foundation provides multiple math numeric constructors. The null check is performed because the constructor throws an error if a null parameter is passed.

This code sample shows set methods where a new math numeric data type is created by passing an integer type value or a big decimal type value:

```
---------------Integer to MathNumeric----------------------
public void setNumberField(Integer numberField){
   if(numberField!=null)
      this.numberField= new MathNumeric(numberField);
}
---------------BigDecimal to MathNumeric-------------------
public void setNumberField(BigDecimal numberField){
   if(numberField!=null)
      this.numberField= new MathNumeric(numberField);
}
---------------MathNumeric to BigDecimal-------------------
public void setNumberField(MathNumeric numberField){
   if(numberField!= null)
      this.numberField= numberField.asBigDecimal();
}
---------------MathNumeric to Integer----------------------
public void setNumberField(MathNumeric numberField){
   if(numberField!= null)
      this.numberField= new Integer(numberField.intValue());
}
```

4.3.6.9 Boolean to and from String

A published Boolean field must be translated to an internal String type field. The business service foundation provides three ValueObject methods to assist you with this transformation. Because these methods are in the ValueObject class, they are available from all value objects. The methods are:

Method	Usage
transformBooleanYN(Boolean)	Returns a string of Y for passed value of true. Returns N for passed value of false. Returns null string for null Boolean.

Method	Usage
transformBoolean01(Boolean)	Returns a string of 1 for passed value of true. Returns 0 for passed value of false. Returns null string for null Boolean.
transformToBoolean(String)	Returns a Boolean value that takes a string. A string of 1,Y,y returns true. A string of 0,N,n returns false. A null or incorrect string returns null.

This code sample shows the structure for each of the methods:

```
-----------------String to Boolean---------------------
//Use ValueObject (tools provided method) transformToBoolean.
//Tools method will account for both Y,y,N,n,0,1 values, null values
//set Boolean to null
public void setIsSomething(String isSomething){
   this.isSomething= transformToBoolean(isSomething);
}
-----------------Boolean to String----------------------
//E1 needs to be researched to determine what values are valid for
//true and false values
//Use ValueObject (tools provided methods) transformBooleanYN or
//transform Boolean01.
//Tools method will provide proper Boolean value for either Y/N or
//0/1, null will result in null String
public void setIsSomething(Boolean isSomething){
   this.isSomething = transformBooleanYN(isSomething);
}
                OR
public void setIsSomething(Boolean isSomething){
   this.isSomething = transformBoolean01(isSomething);
```

4.3.6.10 Data Formatter

In addition to mappings, you might need to format data coming from the published value object. For example, the JD Edwards EnterpriseOne database stores fields such as company (CO) and business unit (MCU) with preceding spaces or zeros. These fields should be formatted so that the preceding spaces and zeros are hidden from the published business service. The business service foundation utilities package provides formatting methods that enable you to pass in a value, and based on the data dictionary rules for the data dictionary item being passed in, formats the value accordingly.

You can use the code template E1DF – EnterpriseOne Data Formatter to generate code for data that requires formatting. The formatter code template generates the code and highlights variable names that you must change.

This sample code is generated by the EnterpriseOne Data formatter code template:

```
//format business unit coming from published vo.
        String formattedMCU = null;
        String bu = this.getBusinessUnit();
        if(bu!=null && !bu.equals("")){
           try {
             formattedMCU = context.getBSSVDataFormatter().format(
this.getBusinesUnit(),"MCU");
             vo.setSzBusinessUnit(formattedMCU);
           }
           catch (BSSVDataFormatterException e) {
             context.getBSSVLogger().app(context,"Error when
```

```
formatting BusinessUnit.",null,vo,e);
             //Create new E1 Message with error from exception
             messages.addMessage(new E1Message(context,
"002FIS",this.getBusinessUnit()));
          }
        }
```

4.3.7 Creating a Media Object Published Value Object (Release 9.1 Update 2)

The business service development tools provide a value object wizard that helps you create Media Object value object classes that follow the methodology rules for published value objects. The Media Object Value Object Class Wizard creates objects based on Media Object data structures. When the wizard generates member variables for the published value object class, it uses the metadata of the data dictionary item in the Media Object data structure.

You use the standard JDeveloper wizard to generate the getter and setter methods for the variables because the Media Object Value Object Class Wizard does not generate these methods. To successfully generate and deploy web services, you must use J2EE standards for naming the getter and setter methods.

The Media Object Value Object Class Wizard generates two java classes: one is the actual value object and the other is a value object that by default is named MOItem_Publish.java. The actual value object contains the properties from the Media Object data structure and the reference to the array of default value objects in order to hold multiple media objects.

The default value object, MOItem_Publish.java, contains the Media Object properties such as moname, seqno, and motype, as well as the attachment. Do not change the name of the default value object.

Below is the sample code for the actual value object created for Media Object data structure ABGT using the Media Object Value Object Class Wizard:

```
public class ABGT extends ValueObject implements Serializable {
    /**
     * Media Object Array <br>
     */
    private MOItem_Publish[] moItems = null;

    /**
     * Address Number
     * <p>
     * TODO: Description using Glossary Text from EnterpriseOne if appropriate.
     * </p>
     * EnterpriseOne Key Field: false <br>
     * EnterpriseOne Alias: AN8 <br>
     * EnterpriseOne field length:  8 <br>
     * EnterpriseOne decimal places: 0 <br>
     */
    private Integer mnAddressNumber = null;

    /**
     * TODO: Default public constructor for instantiating: ABGT
     */
    public ABGT() {
    }
}
```

```
Below is the sample code for Default value Object (MOItem_Publish.java) created
using the Media Object valueobject Wizard.

public class MOItem_Publish extends ValueObject implements Serializable {
    /**
     * Media Object Attachment Type <br>
     */
    private String szMoType = null;

    /**
     * Media Object Attachment File Name <br>
     */
    private String szItemName = null;

    /**
     * Media Object Sequence Number <br>
     */
    private int moSeqNo = 0;

    /**
     * Media Object Data <br>
     */
    private DataHandler szData = null;

    /**
     * TODO: Default public constructor for instantiating: MOItem_Publish
     */
    public MOItem_Publish() {
    }
}
```

4.4 Calling a Business Service

The published business service class exposes a public method as a web service operation. The business service method that the published business service class calls acts as a controller to the business logic.

Starting with Release 9.1 Update 2, published business services can also call Media Object business services. For more information, see Calling a Media Object Business Service (Release 9.1 Update 2).

4.4.1 Rules

These are the rules for a published business service method calling a business service method:

- The signature for the business service static method must contain an IContext object, an IConnection object, and an internal value object.
- The published business service method passes the IContext and IConnection objects to the business service, enabling the published business service to keep track of transaction information throughout the entire processing of the published business service.
- The published business service method creates a new internal value object that is based on the external value object.

- The business service static method returns an E1MessageList object, which contains an array of all error, warning, and information messages that occurred during processing and were set by the business function.
- If the array contains an error message, the published business service must throw an exception using the text from the E1MessageList
- If no error messages exist in the array, the business service returns a confirm value object to the published business service method caller.

  The confirm object is created when the business service passes the internal value object to the constructor for the published confirm value. All warnings and information messages that are returned from calling the business service are mapped to the confirm object.

This code sample shows implementation of these rules:

```
public ConfirmAddAddressBook addAddressBook(AddAddressBook vo) throws
BusinessServiceException {
       return (addAddressBook(null, null, vo));
   }
   protected ConfirmAddAddressBook addAddressBook(IContext context,
                                IConnection connection,
                                AddAddressBook vo) throws
BusinessServiceException {
        //perform all work within try block, finally will clean up any
connections
       try {
           //Call start published method, passing context of null
           //will return context object so BSFN or DB operation can
           //be called later.
           //Context will be used to indicate default transaction
           //boundary, as well as access to formatting and logging
           //operations.
           context = startPublishedMethod(context, "addAddressBook",
vo);
           //Create new published business service messages object for
           //holding errors and warnings that occur during processing.
           E1MessageList messages = new E1MessageList();
           // Create a new internal value object.
           InternalAddAddressBook internalVO =
               new InternalAddAddressBook();
           vo.mapFromPublished(context, internalVO);
           //Call business service passing context, connection and
           //internal VO
           E1MessageList bssvMessages = AddressBookProcessor.addAddressBook
(context, connection, internalVO);
          //Add messages returned from business service to message list
          //for published business service.
           messages.addMessages(bssvMessages);
         //Published Business Service will send either warnings in the
         //Confirm Value Object or throw a published business service
         //Exception.
           //If messages contains errors, throw the exception
           if (messages.hasErrors()) {
               //Get the string representation of all the messages.
               String error = messages.getMessagesAsString();
               //Throw new BusinessServiceException
               throw new BusinessServiceException(error, context);
           }
           //Exception was not thrown, so create the confirm VO from
```

```
            //internal VO
            ConfirmAddAddressBook confirmVO =
                new ConfirmAddAddressBook(internalVO);
            confirmVO.setE1MessageList(messages);
            finishPublishedMethod(context, "addAddressBook");
            //return outVO, filled with return values and messages
            return confirmVO;
        } finally {
            //Call close to clean up all remaining connections and
            //resources.
            close(context, "addAddressBook");
        }
    }
```

4.5 Calling a Media Object Business Service (Release 9.1 Update 2)

The published business service class exposes a public method as a web service operation. The business service method that the Media Object published business service class calls acts as a controller to the business logic that can perform operations on the media objects.

The rules for calling a business service also apply to calling a Media Object business service. See Calling a Business Service for more information.

You may need to call multiple business services from a published business service method to perform the following tasks:

1. Call a normal internal business service, such as a call to the AddAddressBook business service, which does not perform a Media Object operation.
2. Call the Media Object internal business service for adding the Media Object attachments.

For example, you could change the existing RI_AddressBook business service (JPR01000) so that it performs the add Media Object operation after creating an Address Book record. The following sample code shows this example:

```
    protected RI_ConfirmAddAddressBook addAddressBookMO(IContext context,
                          IConnection connection,
                          RI_AddAddressBook vo) throws BusinessServiceException {
    //perform all work within try block, finally will clean up any connections.
        try {
         //Call start published method, passing context of null
         //will return context object so BSFN or DB operation can be called later.
         //Context will be used to indicate default transaction boundary, as well
as access
         //to formatting and logging operations.
         context = startPublishedMethod(context, "addAddressBookMO", vo);

         //Create new PublishedBusinessService messages object for holding errors
and warnings that occur during processing.
         E1MessageList messages = new E1MessageList();
         //TODO: Create a new internal value object.
         RI_InternalAddAddressBook internalVO =
             new RI_InternalAddAddressBook();
         messages.addMessages(vo.mapFromPublished(context, internalVO));
         //Call BSSV passing context, connection and internal VO
         E1MessageList bssvMessages =
             RI_AddressBookProcessor.addAddressBook(context, connection,
                                                    internalVO);
```

```
        //Add messages returned from BSSV to message list for Published Business
Service.
        messages.addMessages(bssvMessages);

        //PublishedBusinessService will send either warnings in the Confirm Value
Object or throw a BusinessServiceException.
        //If messages contains errors, throw the exception
        if (messages.hasErrors()) {
            //Get the string representation of all the messages.
            String error = messages.getMessagesAsString();
            //Throw new BusinessServiceException
            throw new BusinessServiceException(error, context);
        }

        //MO Code to call Internal BSSV - Start
        ABGT_Internal inputVO = new ABGT_Internal();
        messages.addMessages(vo.mapFromPublished(context, inputVO));
        inputVO.setMnAddressNumber(internalVO.getMnAddressBookNumber());

        E1MessageList moMessages =
            RI_AddressBookMediaObjectProcessor.addAddressBookMO(context,
                                                                connection,
                                                                inputVO);
        messages.addMessages(moMessages);
        //MO Code to call Internal BSSV End
        if (messages.hasErrors()) {
            //Get the string representation of all the messages.
            String error = messages.getMessagesAsString();
            //Throw new BusinessServiceException
            throw new BusinessServiceException(error, context);
        }

        //Exception was not thrown, so create the confirm VO from internal VO
        RI_ConfirmAddAddressBook confirmVO =
            new RI_ConfirmAddAddressBook(internalVO);
        confirmVO.setE1MessageList(messages);
        finishPublishedMethod(context, "addAddressBookMO");
        //return outVO, filled with return values and messages
        return confirmVO;
    } finally {
        //Call close to clean up all remaining connections and resources.
        close(context, "addAddressBookMO");
    }
}
```

See Calling Media Object Operations (Release 9.1 Update 2) in this guide for details on how to call the Media Object operations within the internal business service.

4.6 Handling Errors in the Published Business Service

The published business service class is the JD Edwards EnterpriseOne object that is exposed as a web service. Upon invocation, the published business service returns either a value object that contains data and warning messages, or it throws a BusinessServiceException that contains all errors and warnings that occurred during business processing. The published business service throws BusinessServiceException if any messages of the type error occur in the collection of messages that are returned from the call to the business service method. System errors and database failures are thrown as runtime exceptions. A runtime exception is not handled, but it will cause the published business service to fail and return to the original caller. Throwing an

exception causes any database operations that were performed between the default transaction boundaries to roll back, and an error message is sent to the log files.

This code sample shows how to handle errors in the published business service:

```
               E1MessageList messages = AddressBookProcessor.addAddress
Book(context, connection, internalVO);
          //published business service will send either warnings in the
            Confirm Value Object or throw a published business service
            exception.
          //a return status of 2 is an error, throw the exception
          if (messages.hasErrors()) {
              //get the string representation of all the messages
              //RI: Error Handling
              String error = messages.getMessagesAsString();
              //Throw new BusinessServiceException(error);
              throw new BusinessServiceException(error, context);
          }
     //exception was not thrown, so create the confirm VO from internal VO
           ConfirmAddAddressBook confirmVO = new ConfirmAddAddressBook
(internalVO);
         confirmVO.setE1MessageList(messages);
         //return confirm VO, filled with return values and messages
          return confirmVO;
```

4.7 Testing a Published Business Service

You must perform unit testing for the published business service (and business service) that you develop to ensure that the service works as intended. Because published business services depend on the JD Edwards EnterpriseOne system, most of the testing is actually integrated testing. Unit testing should include scenarios that test all decision points in the code. Here are some possible unit tests:

- Test for each action code that is passed, for example, add, change, cancel.
- Test 1 line, 5 lines, 0 lines.
- Perform negative tests.

You can use any of the following methods to test objects in your code:

- Create a test harness class to test the different functions of the published business service.

  If you create a test harness, you must call business service foundation methods at the start and finish of the test to shut down the process within JDeveloper. You can use the code template E1Test – EnterpriseOne Test Harness Class to generate the framework for your test harness application. You can use this code sample as a model for creating a test harness:

```
public static void main(String[] args) throws BusinessServiceException{
      try{
        //call required prior to starting test from application (main())
        TestBusinessService.startTest();
        //call test method
         testAddNoPhone();
      }
      finally{
        //call required after completing test from application (main())
        TestBusinessService.finishTest();
      }
```

```
}
```

- Use the JUnit extension for JDeveloper and create test cases that test the functionality of the published business service.

  JUnit provides a way of running all tests in a suite and can write assertions to determine whether a test passed or failed.

- Test all functionality through the web service graphical user interface that the embedded OC4J within JDeveloper offers.

  When you use this method, you can save and rerun XML documents.

4.7.1 Testing the Web Service

After unit testing is complete, you create a web service from the public methods in the published business service. You should verify that no problems occur when generating or invoking the web service. Testing the web service is critical because it is possible to pass all tests from a test harness and fail at creating or running a web service.

Use the JDeveloper wizard to test the web service. You access this wizard from JDeveloper New Gallery when you add an object to your project.

4.7.2 WSI Compliance Testing

After the published business service is tested as a web service, you verify that the WSDL is WSI compliant. You use JDeveloper for this task.

See Also:

- *Oracle Fusion Middleware User's Guide for Oracle JDeveloper*.

4.8 Customizing a Published Business Service

The published business services that are delivered with your JD Edwards EnterpriseOne software provide a specific, described unit of work. Although these published business services should cover the functionality that you require, you might need to run additional business logic to meet your specific business requirements. This additional business logic could require processing before, after, or during the delivered published business services unit of work. If you require additional business logic, you should create a custom published business service.

When you customize a published business service, upgrades and updates should be a primary consideration. For example, if your customizations include code changes within the published business service or business service classes that are delivered by JD Edwards EnterpriseOne, then when an upgrade or update is applied to your system, a merge of the code itself would be required. Code merging is extremely difficult to perform and is error prone, and good tooling is hard to find.

To keep updates and upgrades simple, Oracle recommends that you create a new published business service that extends the delivered published business service. You use OMW to create and manage your new, custom published business service. When you extend the delivered published business service, you can add your business logic either before or after the delivered published business service's unit of work. By extending the delivered published business service, your custom classes can access the published business service's functionality, control the transaction scope, and share its context. Extending from a published business service class instead of the internal business service class is significant. Published classes have an explicit contract. When you extend a published class, you can be sure that your customizations will continue

to work when your system is updated because the published business service signature and behavior will not change when JD Edwards EnterpriseOne is updated. Internal (business service) classes have no contract and can be changed by JD Edwards EnterpriseOne application development for an update or upgrade.

Extending from published business service classes allows for customizations before and after the delivered published business service's unit of work. If you require custom business logic that processes during the delivered published business service's unit of work, you must create a new published business service and manually copy the delivered published business service and associated business services and modify them as necessary. You use OMW to create and manage your new published business service.

4.8.1 Published Business Service Model

Two methods are required to expose a published business service class as a web service: a public method and a protected method. The sole purpose of the public method is to be called as a web service. The protected method manages and processes the call to the business service classes.

You can use this code sample as a model for your published business service class:

```
/**
* RI_AddressBookManager is the published business service class exposing
* functionality within Address Book processes.
*/
public class AddressBookManager extends PublishedBusinessService {
   /**
    * published business service Public Constructor
    */
   public AddressBookManager() {
   }
   /**
    * Published method for Adding an AddressBook Record.
    * Acts as wrapper method, passing null context and null connection,
    * will call protected addAddressBook.
    * @param vo the value object representing input data for Adding an
    * AddressBook record
    * @return confirmVO the response data from the business process for adding an
    * AddressBook record.
    * @throws BusinessServiceException
    */
   public ConfirmAddAddressBook addAddressBook(AddAddressBook vo) throws
BusinessServiceException {
       return (addAddressBook(null, null, vo));
   }
   /**
    * Protected method for RI_AddressBookManager published business
    * service.
    * addAddressBook will make calls to business service classes
    * for completing business process.
    * @param  vo the value object representing input data for adding an
    * AddressBook record.
    * @param context conditionally provides the connection for the
    * database operation and logging information
    * @param connection can either be an explicit connection or null.
    * If null, the default connection is used.
    * @return response value object is the data returned from the
    * business process for adding an AddressBook record.
    * @throws BusinessServiceException
```

```
     */
    protected ConfirmAddAddressBook addAddressBook(IContext context,
                    IConnection connection,
                    AddAddressBook vo) throws BusinessServiceException {
        //perform all work within try block, finally will clean up any
        //connections
        try {
            //Call start published method, passing context of null will
            //return context object so BSFN or DB operation can be called
            //later.
            //Context will be used to indicate default transaction
            //boundary, as well as access to formatting and logging
            //operations.
            context = startPublishedMethod(context, "addAddressBook", vo);
            //Create new published business service messages object for holding
            //errors and warnings that occur during processing.
            E1MessageList messages = new E1MessageList();
            // Create a new internal value object.
            InternalAddAddressBook internalVO =
                new InternalAddAddressBook();
            vo.mapFromPublished(context, internalVO);
            //Call business service passing context, connection and
            //internal VO
            E1MessageList bssvMessages = AddressBookProcessor.
addAddressBook(context,connection, internalVO);
            //Add messages returned from business service to message list
           //for published business service.
            messages.addMessages(bssvMessages);
            //A published business service will send either warnings in
            //the Confirm Value Object or throw a published business
            //service Exception.
            //If messages contains errors, throw the exception
            if (messages.hasErrors()) {
                //Get the string representation of all the messages.
                String error = messages.getMessagesAsString();
                //Throw new BusinessServiceException
                throw new BusinessServiceException(error, context);
            }
            //Exception was not thrown, so create the confirm VO from
            internal VO ConfirmAddAddressBook confirmVO =
                new ConfirmAddAddressBook(internalVO);
            confirmVO.setE1MessageList(messages);
            finishPublishedMethod(context, "addAddressBook");
            //return outVO, filled with return values and messages
            return confirmVO;
        } finally {
            //Call close to clean up all remaining connections and
            //resources.
            close(context, "addAddressBook");
        }
    }
```

4.8.2 Extending a Published Business Service

You can add functionality to an existing published business service. Custom processing must take place either before or after the business service call and typically, all processing is within the same transaction boundary. You extend a published business service by doing the following tasks:

1. Create a new class that extends the original published business service class.
2. Create a new public method that calls the inherited method for which you are extending functionality.
3. Create custom processing that takes place either before or after the business service call. Typically, all processing will be within the same transaction boundary.

```
  /**
  * Published method for Customized Add Address Book
  * This exposed method will call the method addAddressBook from
  * parent class.
  * @param vo the value object representing input data for adding
  * AddressBook record
  * @return confirmVO the response data from the business process for
  * adding an address book record.
  * @throws BusinessServiceException
  */
       public ConfirmAddAddressBook customAddAddressBook
(AddAddressBook vo) throws BusinessServiceException {
            //perform all work within try block, finally will clean up
            //any connections
            IContext context = null;
            IConnection connection = null;
            try {
                //Call start published method, passing context of null
                //will return context object so BSFN or DB operation can
                //be called later.
                //Context will be used to indicate default transaction
                //boundary, as well as access to formatting and logging
                //operations.
                context = startPublishedMethod(context,
"customAddAddressBook",vo);
                //Create new published business service messages object
                //for holding errors and warnings that occur during
                //processing.
                E1MessageList messages = new E1MessageList();

                //TODO:  This is where a customer customization would be
                //coded.
                //Whatever is coded here is included within the
                //transaction but occurs prior to calling the published
                //business service.

                //Call published business service method
                ConfirmAddAddressBook confirmVO = this.addAddressBook
(context, connection, vo);

                //TODO:  This is where a customer customization would be
                //coded.
                //Whatever is coded here is included within the
                //transaction but occurs after calling the published
                //business service.

                //published business service will send either warnings
                //in the Confirm Value Object or throw a published
                //business service Exception.
                //If messages contains errors, throw the exception

               if (messages.hasErrors()) {
                    //get the string representation of all the messages
```

```
                String error = messages.getMessagesAsString();
                //Throw new BusinessServiceException
                throw new BusinessServiceException(error, context);
            }

            //Call finish published method, passing the context
            //to commit default implicit transaction(in case of no
            //exceptions)
            finishPublishedMethod(context, "customAddAddressBook");
            //return confirmVO, mapped with return values and
            //messages
            return confirmVO;
        } finally {

            //Call close to clean up all remaining connections and
            //resources.
            close(context,"customAddAddressBook");
        }

    }
```

4.9 Deprecating a Published Business Service

When the signature of a published business service is modified, a new published business service is created to replace the original published business service. The JD Edwards EnterpriseOne deprecation policy for published business services is to ship and support both the original and the replacement published business service for the first release of the replacement published business service. For the second release of the replacement published business service, only the replacement published business service is shipped, but both the original and replacement published business services are supported. For the third release, only the replacement published business service is shipped and supported. The original published business service is no longer supported. For example, oracle.e1.bssv.JP010003 is shipped with 9.0. For 9.1, oracle.e1.bssv.JP010022 is created to replace JP010003. Both published business services are shipped and supported for Release 9.1. For Release 9.2, only the replacement published business service (JP010022) is shipped, but both published business services (JP010022 and JP010003) are supported. For Release 9.3, only JP010022 is shipped and supported. The original published business service (JP010003), which was shipped with 9.0 and 9.1, is no longer supported.

5

Creating a Business Service

This chapter contains the following topics:

Important: Oracle reserves the right to reorganize the business services foundation packages (jar files) for tools release upgrades. If you are planning to upgrade your system, test your custom objects and modify them as appropriate to ensure your code will continue to work as intended. You cannot upgrade custom business service objects after you install a tools release upgrade.

5.1 Understanding Business Services

Business services are JD Edwards EnterpriseOne Object Management Workbench (OMW) objects that are called by a published business service to accomplish a specific task. Business service classes are written in Java programming language and provide methods that access the business logic in JD Edwards EnterpriseOne for many supported business transactions, such as journal entries, exchange rates, accounts payable vouchers, inventory lookups, pricing, sales orders, and so on. A business service method can call a business function or a database operation. A utility business service performs a repeatable task and can be called by multiple business service classes.

This chapter focuses on business services that call a business function. Because many of the rules and best practices are the same for business services that call business functions and business services that call database operations, discussions in this chapter are applicable to both types of business services. However, some differences

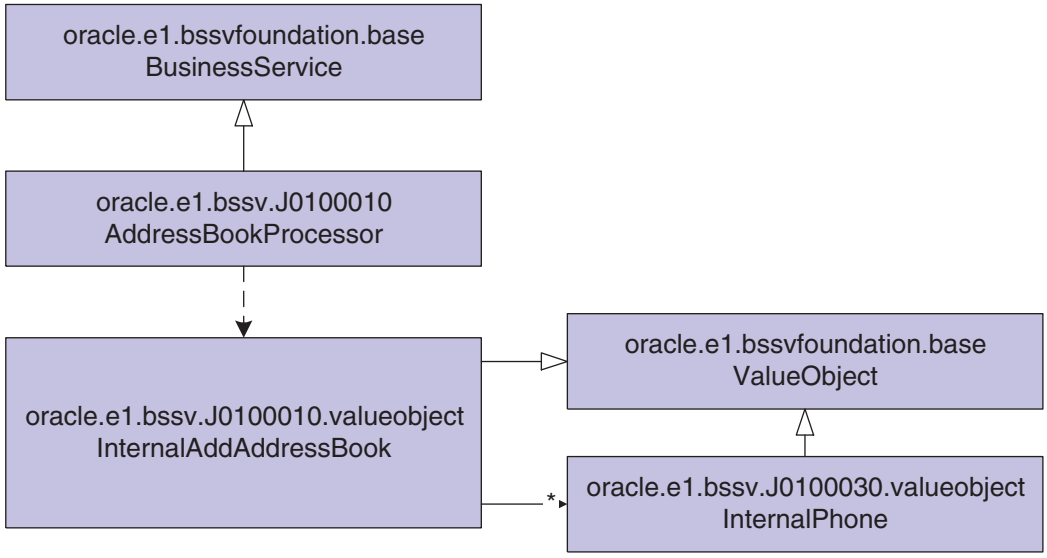
and exceptions exist, and Chapter 5, Creating a Business Service That Calls a Database Operation focuses on differences for each type of database operation.

You use wizards, which are provided by JDeveloper and the business services framework, and the Java programming language to create business service classes. If you are creating a new business service, you first create an OMW object. When you launch JDeveloper from OMW, the project should be created automatically. If the project is not created, you use the Project wizard that is provided by JDeveloper to create a project for your business service. You use the Business Service Class wizard to create a business service class that has one or more methods. A method can call a business function, a database operation, or another business service (for example, a utility business service method) to accomplish a specific task. The business services framework provides two wizards: the Create Business Function Call wizard to help you create methods that call business functions and the Create Database Call wizard to help you create methods that call database operations.

In addition to wizards, the business services framework provides a set of foundation packages that help you create a business service method. Each foundation package contains a set of interfaces and related classes. All business service classes extend from the BusinessService foundation class. The wizards that are provided by the business service framework enable you to create code that is specific for calling a business function or a database operation. Code samples, using a specific example of adding an address book record that uses AddressBook master business function, are provided throughout this chapter to demonstrate general concepts. Rules and best practices are discussed if they are applicable to the topic.

This business service class diagram shows the main business service class (AddressBookProcessor) and the internal value object class (InternalAddAddressBook) and its components:

Figure 5–1 Business service class diagram



These features are illustrated in the diagram:

- AddressBookProcessor extends BusinessService class.
- InternalAddAddressBook and its components extend ValueObject class.

5.2 Developing a Business Service

A business service represents one or more Java classes that expose public methods. A business service class can expose multiple methods, such as addAddressBook, addAddressBookWithPhones, changeAddressBook, and so on. The methods access logic in JD Edwards EnterpriseOne and support a specific step in a business process, for example, adding an address book record. When you create the business service, you should consider including methods that have similar functionality and manageability in the same business service. If multiple processes are similar and can reuse code, then these methods should exist in the same business service.

5.2.1 IContext and IConnection Objects

A business service public method must contain two objects, IContext and IConnection, as part of its signature. The IContext object provides the default connection for the business function call and holds an identifier that ties together all processing for the business service. The IConnection object enables the business service method to be run under an explicit transaction; and if the connection is null, the default transaction is used. The context and connection objects are passed to the public methods of the business service class, which in turn passes these objects to any of the methods that call a business function. To indicate the boundaries of the internal method, business service public methods must call the inherited methods, startInternalMethod(context, "methodName", valueObject) before any other logic and finishInternalMethod(context, "methodName", valueObject) when all other processing is finished.

This code sample shows how to use IContext and IConnection:

```
public static E1MessageList addAddressBook(IContext context, IConnection
connection, InternalAddAddressBook internalVO){
        //call start internal method, passing the context (which was
        //passed from published business service)
       startInternalMethod(context, "addAddressBook",internalVO);
       ...
       // calls method which then executes BSFN AddressBookMBF
       E1MessageList messages = callAddressBookMasterMBF
(context,connection, internalVO, programId);
       ...
       // call finish internal method passing context
       finishInternalMethod(context, "addAddressBook");
       //return status code from BSFN call
       ...
       return messages;
   }
```

See Also:

- Transaction Processing.

5.3 Managing Business Service Components

This section discusses naming conventions and concepts for creating business service classes, methods, internal value objects, and fields. Code samples are provided as examples for you to follow. Rules and best practices are also discussed.

5.3.1 Business Service Class Names

The naming convention for a business service class is to use the functional description with Processor added at the end of the name, for example, AddressBookProcessor and AddressBookQueryProcessor.

This code sample shows the naming convention for a business service class:

```
public abstract class AddressBookProcessor extends BusinessService {
   ....
   }
```

5.3.2 Business Service Method Names

A method is an operation that performs a business process. The naming convention for a business service public method is to name the public method the same name as the method in the published business service, for example, addAddressBook.

This code sample shows the naming convention for a public method:

```
public static E1MessageList addAddressBook(IContext context,
IConnection connection, InternalAddAddressBook internalVO){
   ...
   }
```

5.3.3 Business Service Internal Value Object Names

Internal value object classes are the input and output parameters of the business service methods. These value objects are not published interfaces. You use these internal value objects to map values to and from a business function. Internal value objects can be composed of fields, compounds, and components.

The naming convention for an internal value object class is to use the published value object name with Internal added to the beginning of the name. Some examples of names for internal value objects are InternalAddAddressBook, InternalProcessPurchaseOrder, and InternalEntity.

This code sample shows the naming convention for an internal value object class:

```
    public class InternalAddAddressBook extends ValueObject {
   ....
   }
```

Database operations use a different convention for naming internal value objects.

See Understanding Database Operations.

5.3.3.1 Field Names

The naming convention for field names in the internal value object is to use a name that matches the data structure member names of the business function that is being called, for example, mnAddressNumber and szMailingName. The number of fields exposed through the internal value object may be larger than the published value object, and you should include all of the possible business data fields and flag fields in the internal value object because this object can be used by internal applications.

5.3.3.2 Compound and Component Names for a Business Service

By design, the internal value object has a flat hierarchical structure, meaning that the structure contains few, if any, compounds and components. Compounds and

components that exist within an internal value object should be named similarly; for example, the compound name should be prefaced with the word Internal (such as, InternalPhones).

The following code sample shows an internal value object class that has one compound (internalPhones) and many field names (szAlphaName, szSearchType, and so on) at the top level that correspond to business function data structure member names.

```
public class InternalAddAddressBook extends ValueObject {
   private String szLongAddressNumber;
   private MathNumeric mnAddressBookNumber;
   private String szTaxId;
   private String szMailingName;
   private String szAddressLine1;
   private String szAddressLine2;
   private String szAddressLine3;
   private String szAddressLine4;
   private String szPostalCode;
   private String szCity;
   private String szCounty;
   private String szState;
   private String szCountry;
   private String szAlphaName;
   private String szSearchType;
   private String szVersion;
   private String szBusinessUnit;
   private Date jdDateEffective;
   private ArrayList internalPhones;
   ...
 }
```

5.3.4 Creating a Business Service Class

The business service foundation provides the Business Service Object wizard, which you use to create new business service classes. This wizard follows the methodology discussed in this document. The Business Service Object wizard prompts you for the class name, an internal input value object class, and a method name, and then it generates code for a business service class. The wizard also generates comments and TODO: statements where necessary to help you complete the generated code.

See "Creating Business Service Classes" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

5.3.4.1 Rules

When you create a business service class, follow these rules:

- Business service classes are abstract classes and must extend the foundation class BusinessService. BusinessService is the parent class that provides foundation support for transactions and logging.
- Business service classes have only static methods, so to reinforce static behavior and prevent the class from being instantiated, declare an abstract class.

This code sample illustrates extending the BusinessService class and declaring the class as abstract:

```
public abstract class AddressBookProcessor extends BusinessService {
```

```
...
}
```

You design and develop a business service as a static class that processes multiple requests simultaneously. A static class means that only one instance of the class exists in Java virtual memory (JVM), regardless of the number of simultaneous requests being processed. These requests are also called threads.

Static classes reduce object creation. If a business service was not static, one business service would exist for each request. As each request finishes, the class would be released and eventually the system reclaims the memory that the class used. Creating and releasing objects repeatedly causes performance degradation, because more memory is used and more CPU cycles are required.

To ensure that the business service provides a thread-safe environment, you cannot use instance variables in the business service class. An instance variable is a value that is useful to only one request, for example, a counter. A thread-safe environment means that the multiple requests (threads) that are being processed simultaneously do not interfere with each other. The absence of instance variables helps ensure thread safety at compile time. You can include static variables in the business service class. A static variable is a value that is useful to all requests, for example, a cached value that is used to specify a language. A static variable is shared data, independent of a request.

5.3.5 Declaring a Business Service Public Method

A public method is an operation that can be used by other classes and methods. The signature takes IContext, IConnection, and an internal value object and returns E1MessageList.

You can add additional public methods to a business service class by accessing the JDeveloper Code Templates and selecting E1SM – EnterpriseOne Business Service Method Call. This template generates code for a public method. You press Tab to move through the highlighted fields and complete the code. This template enforces methodology and gives you a head start for developing a new public method.

This code sample shows how to declare a public message:

```
public static E1MessageList  addAddressBook(IContext context,
IConnection connection, InternalAddAddressBook internalVO){
       startInternalMethod(context, "addAddressBook", internalVO);
       // call BSFN AddressBookMBF
       E1MessageList messages = callAddressBookMasterMBF(context,
connection, internalVO, programID);
       finishInternalMethod(context, "addAddressBook");
       return messages;
   }
```

5.3.5.1 Rules for Declaring a Business Service Public Method

When you declare a public method for a business service class, follow these rules:

- Business service classes must expose public static methods to a published business service class. A business service class cannot contain instance variables or nonstatic methods.
- Business service methods that are to be used by a published business service must return an E1MessageList object to that published business service. The caller of the business service determines how to handle the errors and whether to create and throw an exception. The signature of the business service method must contain

IContext and IConnection objects and a value object class that represents an internal value object that passes values to the business function calls.

5.3.5.2 Best Practices for Private and Protected Methods

When you declare methods other than the public method (for example, a utility method), consider these best practices:

- Declare nonpublic methods as protected or private; all methods must be static.
- Keep scope as private as possible.

5.3.6 Creating Internal Value Objects

Internal value object classes and their components extend the foundation ValueObject class.

The business service foundation provides value object class wizards that help you create internal value object classes that follow methodology rules. The value object wizards also assist you by pulling useful information from the JD Edwards EnterpriseOne data dictionary into the Javadoc for value objects. You must create accessor methods (getter and setter methods) because the value object wizards do not generate these methods. Also, you must provide the description name of the field for the Javadoc.

The value object wizards enable you to create value object classes from the data structures that are defined within a business function or from database tables or business views. Remember that the wizard uses the field name that comes from the data structure, table, or business view to generate member variables for the internal value object class. These generated variables look very much like JD Edwards EnterpriseOne data items.

This code sample is from a business function:

```
/**
* Address Line 1
* EnterpriseOne Alias: ADD1
* EnterpriseOne field length:  40
*/
private String szAddressLine1 = null;
```

This code sample is from a table:

```
/**
* CreditMessage
* A value in the user defined code table
* that indicates the credit status of a customer or supplier
* EnterpriseOne Alias: CM
* EnterpriseOne field length:  2
* EnterpriseOne User Defined Code: 00/CM
*/
private String F0101_CM = null;
```

See Understanding Database Operations.

See "Creating Business Function Calls" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

See "Creating Database Operation Calls" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

5.3.6.1 Rules for Internal Value Object

This list identifies the rules for internal value objects:

- The structure of an internal value object has a flatter hierarchy than the published value object, because the internal value object has few if any compounds or components.
- The collections within the internal value object can be created using either ArrayList or Array. An ArrayList is easier to work with because it can be dynamically sized. Arrays are necessary when the internal value object will be serialized. A business service that exposes JD Edwards EnterpriseOne functionality to a third party can use an ArrayList. A business service called from a business function (for example, using web service callout when JD EnterpriseOne is a web service consumer) must use an Array because the ArrayList data type cannot be serialized.

  For example, you can use the following code sample to declare the compound for phones:

  ```
  Private ArrayList internalPhones = null;
  ```

  ArrayList is populated during business service processing, and in the preceding code sample, the collection contains InternalPhone objects.

  Or you can use this code sample to declare the compound for phones:

  ```
  Private InternalPhone[] internalPhones = null;
  ```

- The data types for internal value object classes match the types used in the JD Edwards EnterpriseOne data structures, as identified in the following table:

Internal Value Object Data Type	Usage
oracle.e1.bssvfoundation.util.MathNumeric	Use for all fields that are declared as numeric in JD Edwards EnterpriseOne.
java.lang.String	Use for string and char fields.
java.util.Date	Use for all JDEDate fields in JD Edwards EnterpriseOne.
java.util.GregorianCalendar	Use for all UTIME fields in JD Edwards EnterpriseOne.

5.3.6.2 Best Practices for Internal Value Object

When deciding which fields to include in the internal value object class, consider that all data fields that the application accepts and the function uses are valid fields.

If an internal business function call passes processing fields, you must determine whether these fields should be exposed in the internal value object class. An example of this type of processing field would be a field that is used to manipulate a cache. If a business service is called from another business service and a processing field is exposed and passed in from the calling business service, will the behavior be as expected? If not, that processing field should not be exposed in the internal value object class. Fields that should not be exposed in the internal value object class can be handled by creating another value object called InternalProcessing. The InternalProcessing value object can contain all unexposed processing fields as member variables. The InternalProcessing value object should not be part of the InternalValueObject class and should not be exposed from the business service method

signature. The InternalProcessing value object can be passed in the business function method calls but is not passed in or out of the business service method.

This code sample shows an InternalProcessing value object:

```
/**
* InternalProcessing contains processing fields used for
* ProcessPurchaseOrderAcknowledge
* but these will not be exposed fields.
*/
public class InternalProcessing extends ValueObject {
   /**
    * Action Flag
    * EnterpriseOne Key Field: false
    * EnterpriseOne Alias: ACFL
    * EnterpriseOne field length:  1
    * EnterpriseOne User Defined Code: 08/AC
    */
   private String cProcessHeaderDetailFlag = null;
  /**
    * Job Number
    * EnterpriseOne Key Field: false
    * EnterpriseOne Alias: JOBS
    * EnterpriseOne field length:  8
    * EnterpriseOne decimal places: 0
    * EnterpriseOne Next Number: 00/4
    */
   private MathNumeric mnF4311JobNumber = null;
   /**
    * Transaction ID
    * EnterpriseOne Key Field: false
    * EnterpriseOne Alias: TCID
    * EnterpriseOne field length:  15
    * EnterpriseOne decimal places: 0
    */
   private MathNumeric mnTransactionID = null;
   /**
    * Process ID
    * EnterpriseOne Key Field: false
    * EnterpriseOne Alias: PEID
    * EnterpriseOne field length:  15
    * EnterpriseOne decimal places: 0
    */
   private MathNumeric mnProcessID = null;
   /**
    * Job Number
    * EnterpriseOne Key Field: false
    * EnterpriseOne Alias: JOBS
    * EnterpriseOne field length:  8
    * EnterpriseOne decimal places: 0
    * EnterpriseOne Next Number: 00/4
    */
   private MathNumeric mnCacheJobNumber = null;
}
```

This code sample shows how to pass the InternalProcessing value object to business function methods:

```
     public static E1MessageList processPurchaseOrderAcknowledge
(IContext context,IConnection connection, InternalProcessPurchase
OrderAcknowledge internalVO){
```

```
       //Call start internal method, passing the context (which was
       //passed from published business service).
       startInternalMethod(context, "processPurchaseOrderAcknowledge",
 internalVO);
       //Create new message list for business service processing.
       E1MessageList messages = new E1MessageList();
       InternalProcessing internalProcessingVO = new
InternalProcessing();
       //TODO: call method (created by the wizard), which then
       //executes Business Function or Database operation.
       messages = callPurchaseOrderAcknowledgeNotify(context,
connection, internalVO,internalProcessingVO);
       //TODO:  add messages returned from E1 processing to business
       //service message list.
       //Call finish internal method passing context.
       finishInternalMethod(context, "processPurchaseOrderAcknowledge
");
       //Call finish internal method passing context.
       return messages;
   }
   /**
    * Calls the PurchaseOrderAcknowledgeNotify(B4302190) business
    * function which has the D4302190A data structure.
    * @param context conditionally provides the connection for the
    * business function call and logging information
    * @param connection can either be an explicit connection or null.
    * If null the default connection is used.
    * @param TODO document input parameters
    * @return A list of messages if there were application errors,
    * warnings,or informational messages. Returns null if there were
    * no messages.
    */
   private static E1MessageList callPurchaseOrderAcknowledgeNotify
(IContext context, IConnection connection, InternalProcessPurchase
OrderAcknowledge internal VO, InternalProcessing internalProcessingVO) {
       BSFNParameters bsfnParams = new BSFNParameters();
       // map input parameters from input value object
       bsfnParams.setValue("cProcessHeaderDetailFlag",
internalProcessingVO.
get CProcessHeaderDetailFlag());
       bsfnParams.setValue("mnF4311JobNumber", internalProcessingVO.
getMnF4311JobNumber());
       bsfnParams.setValue("mnTransactionID", internalProcessingVO.
getMnTransactionID());
       bsfnParams.setValue("mnProcessID", internalProcessingVO.get
MnProcessID());
       bsfnParams.setValue("mnCacheJobNumber", internalProcessingVO.
getMnCacheJobNumber());
       bsfnParams.setValue("cHeaderOrderStatusCode", internalVO.get
CHeaderOrderStatusCode());
       bsfnParams.setValue("mnOrderNumber", internalVO.getMnOrder
Number());
       bsfnParams.setValue("szOrderType", internalVO.
getSzOrderType());
       bsfnParams.setValue("szOrderCompany", internalVO.getSzOrder
Company());
       ....
       //get bsfnService from context
       IBSFNService bsfnService = context.getBSFNService();
       //execute business function
```

```
       bsfnService.execute(context, connection, "PurchaseOrder
AcknowledgeNotify",bsfnParams);
       //map output parameters to output value object
       internalProcessingVO.setCProcessHeaderDetailFlag(bsfnParams.
getValue("cProcessHeaderDetailFlag").toString());
       internalProcessingVO.setMnF4311JobNumber((MathNumeric)bsfn
Params.getValue("mnF4311JobNumber"));
       internalProcessingVO.setMnTransactionID((MathNumeric)bsfn
Params.getValue("mnTransactionID"));
       internalProcessingVO.setMnProcessID((MathNumeric)bsfnParams.
getValue("mnProcessID"));
       internalProcessingVO.setMnCacheJobNumber((MathNumeric)bsfn
Params.getValue("mnCacheJobNumber"));
       internalVO.setCHeaderOrderStatusCode(bsfnParams.
getValue("cHeaderOrderStatusCode").toString());
       internalVO.setMnOrderNumber((MathNumeric)bsfnParams.getValue
("mnOrderNumber"));
        ...
       //return any errors, warnings, or informational messages to the
       //caller
       return bsfnParams.getE1MessageList();
   }
```

5.3.7 Creating Internal Media Object Value Objects (Release 9.1 Update 2)

Like other internal value object classes, Media Object value objects and their components extend the ValueObject foundation class. The business service foundation provides the Media Object Value Object Class Wizard, which helps you create internal Media Object value object classes that follow methodology rules. The value object wizards also assist you by pulling useful information from the JD Edwards EnterpriseOne data dictionary into the Javadoc for value objects. You must create accessor methods (getter and setter methods) because the value object wizards do not generate these methods. Also, you must provide the description name of the field for the Javadoc. The wizard uses the field name that comes from the Media Object data structure to generate member variables for the internal Media Object value object class.

The Media Object Value Object Class Wizard generates two java classes: one is the actual internal value object, and the other is the value object that is named MOItem_Internal.java by default. The internal value object contains the properties from the Media Object data structure and the reference to the array of default value objects (MOItem_Internal.java) in order to hold multiple media objects.

The default value object, MOItem_Internal.java, contains the Media Object properties such as media object name, type, and attachment data. Do not change the name of the default value object.

Apart from the variables created from the Media Object data structure, the wizard creates the following two default methods and one member variable:

- getSzMoKey

  This method prepares the Media Object Key for the media object. If desired, you can customize the logic to override this method to meet your business need.

- getSzMoName

  This method is used to retrieve the name of the Media Object data structure.

- downloadMediaObject

This is the member variable. If set to true, then the Media Object Select operation will return a list of all media objects along with file attachments. If set to false, the Media Object Select operation will return a list of all media objects but will not download the file attachments. By default, this variable is set to true.

The following code is an example of an internal value object created through the Media Object Value Object Class Wizard for the Media Object data structure ABGT:

```
public class ABGT_Internal extends ValueObject implements Serializable {
    /**
     * Media Object Array <br>
     */
    private MOItem_Internal[] moItems = null;

    /**
     * Download Attachments <br>
     */
    private boolean downloadMediaObject = true;

    /**
     * Address Number
     * <p>
     * TODO: Description using Glossary Text from EnterpriseOne if appropriate.
     * </p>
     * EnterpriseOne Key Field: false <br>
     * EnterpriseOne Alias: AN8 <br>
     * EnterpriseOne field length:  8 <br>
     * EnterpriseOne decimal places: 0 <br>
     * EnterpriseOne Next Number: 01/1 <br>
     */
    private MathNumeric mnAddressNumber = null;

    /**
     * Builds and returns the Media Object Key with the media object attributes
     */
    public String getSzMoKey() {
        String key = String.valueOf(mnAddressNumber);
        if (key.startsWith("null|"))
        {
            key = key.substring(4, key.length());
        }
        if (key.endsWith("|null"))
        {
            key = key.substring(0, key.length() - 4);
        }
        while(key.indexOf("|null|") != -1)
        {
            key = key.replace("|null|", "||");
        }
        return key;
    }

    /**
     * Returns the Media Object name
     */
    public String getSzMoName() {
        return "ABGT";
    }
}
```

The following is an example of the default value object (MOItem_Internal) created through the Media Object Value Object Wizard:

```
public class MOItem_Internal extends ValueObject implements Serializable {
    /**
     * Media Object Attachment Type <br>
     */
    private String szMoType = null;

    /**
     * Media Object Attachment File Name <br>
     */
    private String szItemName = null;

    /**
     * Media Object Sequence Number <br>
     */
    private int moSeqNo = 0;

    /**
     * Media Object Data <br>
     */
    private DataHandler szData = null;

    /**
     * TODO: Default public constructor for instantiating: MOItem_Internal
     */
    public MOItem_Internal() {
    }

    /**
     * TODO: Default public constructor for instantiating: MOItem_Internal
     */
    public MOItem_Internal(String szMoType) {
        this.szMoType = szMoType;
    }

    /**
     * TODO: Default public constructor for instantiating: MOItem_Internal
     */
    public MOItem_Internal(int moSeqNo) {
        this.moSeqNo = moSeqNo;
    }

    /**
     * TODO: Default public constructor for instantiating: MOItem_Internal
     */
    public MOItem_Internal(String szMoType, String szItemName, DataHandler szData)
{
        this.szMoType = szMoType;
        this.szItemName = szItemName;
        this.szData = szData;
    }
}
```

5.4 Calling Business Functions

A business function is an encapsulated set of business rules and logic that can be reused by multiple applications. Business functions provide a common way to access the JD Edwards EnterpriseOne database. A business function performs a specific task.

You use the business service foundation Business Function Call Wizard to create a business function call.

See "Understanding Business Function Calls" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

This code sample is generated by the Business Function Wizard:

```
//calls method which then executes BSFN AddressBookMBF
//RI: This private function is created by the wizard, The
//business function will be executed inside this internal function
messages = callAddressBookMasterMBF(context, internalVO,programId);
```

The wizard creates a generic method. You modify the signature of the method and complete the code for the objects that will be accessed for mapping to and from the business function call. The wizard creates InputVOType as a placeholder in the signature for the internal value object class name that you provide.

This code sample shows a business function call that was created by the wizard:

```
/**
  * Calls the AddressBookMasterMBF(N0100041) business function which has
  * the D0100041 data structure.
  * @param context provides the connection for the business function call
  * and logging information
  * @param TODO document input parameters
  * @return A list of messages if there were application errors, warnings,
  * or informational messages. Returns null if there were no messages.
  */
     private static E1MessageList callAddressBookMasterMBF(IContext
context, IConnection connection, InputVOType internalVO) {
     BSFNParameters bsfnParams = new BSFNParameters();
     // map input parameters from input value object
     bsfnParams.setValue("cActionCode", internalVO.getCActionCode());
     bsfnParams.setValue("cUpdateMasterFile", internalVO.getCUpdateMaster
File());
     bsfnParams.setValue("cProcessEdits", internalVO.getCProcessEdits());
     bsfnParams.setValue("cSuppressErrorMessages", internalVO.getCSuppress
ErrorMessages());
     bsfnParams.setValue("szErrorMessageID", internalVO.getSzErrorMessage
ID());
     bsfnParams.setValue("mnSameAsExcept", internalVO.getMnSameAsExcept());
     bsfnParams.setValue("mnAddressBookNumber", internalVO.getMnAddressBook
Number());
     ...
       try {
          //get bsfnService from context
          IBSFNService bsfnService = context.getBSFNService();
          //execute business function
          bsfnService.execute(context, connection, "AddressBookMasterMBF",
bsfnParams);
      } catch (BSFNServiceInvalidArgException invalidArgEx) {
         //Create error message for Invalid Argument exception and return
         //it in ErrorList
          E1MessageList returnMessages = new E1MessageList();
          returnMessages.addMessage(new E1Message(context, "018FIS",
```

```
invalidArg
Ex.getMessage()));
          return returnMessages;
      } catch (BSFNServiceSystemException systemEx) {
          //Create error message for System exception and return it in
          //ErrorList
          E1MessageList returnMessages = new E1MessageList();
          returnMessages.addMessage(new E1Message(context, "019FIS",
systemEx.getMessage()));
          return returnMessages;
      }
     //map output parameters to output value object
     internalVO.setMnAddressBookNumber(bsfnParams.getValue("mnAddressBook
Number");
     internalVO.setSzLongAddressNumber(bsfnParams.getValue("szLongAddress
Number");
     internalVO.setSzTaxId(bsfnParams.getValue("szTaxId"));
     internalVO.setSzAlphaName(bsfnParams.getValue("szAlphaName"));
     internalVO.setSzSecondaryAlphaName(bsfnParams.getValue("szSecondary
AlphaName"));
     internalVO.setSzMailingName(bsfnParams.getValue("szMailingName"));
     internalVO.setSzSecondaryMailingName(bsfnParams.getValue("szSecondary
MailingName"));
     internalVO.setSzDescriptionCompressed(bsfnParams.getValue
("szDescriptionCompressed"));
     internalVO.setSzBusinessUnit(bsfnParams.getValue("szBusinessUnit"));
     internalVO.setSzAddressLine1(bsfnParams.getValue("szAddressLine1"));
     //return any errors, warnings, or informational messages to the caller
     return bsfnParams.getE1MessageList();
   }
```

After the wizard creates the code for the generic method, you modify the code as needed. You might need to:

- Add parameters to be passed.

  At a minimum, the internal value object includes an IContext object and an IConnection object, generated by the wizard, and an internal value object, which you define. You may need to pass an additional parameter such as an internalProcessing value object for processing fields that should not be exposed.

- Fix mappings if required.

  The generated code assumes that all fields can be mapped directly to and from the internal value object. If an additional structure exists in the internal value object or some fields should be mapped from class constant fields, you must fix the mapping statements where this assumption is not true. JDeveloper identifies incorrect statements.

- Fix the data type of the object retrieved from bsfnParams.

  The generated code adds a cast argument when mapping to internalVO by getting values from the bsfnParams object. The bsfnParams object is a collection of objects and when an object is retrieved, the type needs to be cast to the correct data type so that it can be added to the internalVO reference, as illustrated in this code sample:

  ```
  private static E1MessageList callAddressBookMasterMBF(IContext context,
                                          IConnection connection,
                                          InternalAddAddressBook internalVO,
                                          String programId) {
        // create new bsfnParams object
  ```

```
        BSFNParameters bsfnParams = new BSFNParameters();
       //set values for bsfn params based on internal vo attribute values
        bsfnParams.setValue("cActionCode", ACTION_CODE_ADD);
        bsfnParams.setValue("cUpdateMasterFile", UPDATE_MASTER_TRUE);
        bsfnParams.setValue("cProcessEdits", PROCESS_EDITS_TRUE);
        bsfnParams.setValue("cSuppressErrorMessages", SUPPRESS_ERROR_FALSE);
        bsfnParams.setValue("szVersion", internalVO.getSzVersion());
        bsfnParams.setValue("mnAddressBookNumber",
                            internalVO.getMnAddressBookNumber());
        bsfnParams.setValue("szLongAddressNumber",
                            internalVO.getSzLongAddressNumber());
        bsfnParams.setValue("szTaxId", internalVO.getSzTaxId());
        bsfnParams.setValue("szSearchType", internalVO.getSzSearchType());
        ...
        bsfnParams.setValue("szState", internalVO.getSzState());
        bsfnParams.setValue("szCountry",
                            internalVO.getSzCountry());
         //set program id to value retrieved in business service properties
         bsfnParams.setValue("szProgramId", programID );
        try {
             //get bsfnService from context
             IBSFNService bsfnService = context.getBSFNService();
             //execute business function

             bsfnService.execute(context, connection, "AddressBookMaster
MBF", bsfnParams);
         } catch (BSFNServiceInvalidArgException invalidArgEx) {
            //Create error message for Invalid Argument exception and
            //return it in ErrorList
             E1MessageList returnMessages = new E1MessageList();
             returnMessages.addMessage(new E1Message(context, "018FIS",
invalidArgEx.getMessage()));
             return returnMessages;
         } catch (BSFNServiceSystemException systemEx) {
             //Create error message for System exception and return it in
             //ErrorList
             E1MessageList returnMessages = new E1MessageList();
             returnMessages.addMessage(new E1Message(context, "019FIS",
systemEx.getMessage()));
             return returnMessages;
         }
        //set internal VO attributes based on values passed back from bsfn
        //Must cast object to appropriate data type coming from bsfnParams
collection.
        internalVO.setMnAddressBookNumber((MathNumeric)bsfnParams.
getValue("mnAddressBookNumber"));
        internalVO.setSzLongAddressNumber((String)bsfnParams.
getValue("szLongAddressNumber"));
        internalVO.setSzCountry((String)bsfnParams.getValue("szCountry"));
        internalVO.setSzBusinessUnit((String)bsfnParams.
getValue("szBusinessUnit"));
        internalVO.setJdDateEffective((Date)bsfnParams.
getValue("jdDateEffective"));
        E1MessageList messages = bsfnParams.getE1MessageList();
        //set prefix to the message list being returned to provide more
information on errors
        bsfnParams.getE1MessageList().setMessagePrefix("AB MBF N0100041");
        //return any errors, warnings, or informational messages to the
        //caller
        return bsfnParams.getE1MessageList();
```

```
    }
```

When you run a business function, two exceptions, BSFNServiceInvalidArgException and BSFNServiceSystemException, are thrown. The generated code runs the business function within a try/catch block, and in the event that an invalid argument is passed to the business function, the error will be caught and added to the message list and returned to the caller. The same behavior occurs if a database exception occurs within the business function. This code sample shows a try/catch block:

```
try {
    //get bsfnService from context
    IBSFNService bsfnService = context.getBSFNService();
    //execute business function
    bsfnService.execute(context, connection, "AddressBookMasterMBF",
bsfnParams);
} catch (BSFNServiceInvalidArgException invalidArgEx) {
   //Create error message for Invalid Argument exception and return it in
ErrorList
    E1MessageList returnMessages = new E1MessageList();
    returnMessages.addMessage(new E1Message(context, "018FIS",
invalidArgEx.getMessage()));
    return returnMessages;
} catch (BSFNServiceSystemException systemEx) {
    //Create error message for System exception and return it in ErrorList
    E1MessageList returnMessages = new E1MessageList();
    returnMessages.addMessage(new E1Message(context, "019FIS",
systemEx.getMessage()));
    return returnMessages;
}
```

5.5 Calling Database Operations

You can create business services that call database operations. You use the business service foundation Database Call wizard to create these business service methods. Database operations include query, insert, update, and delete.

This code sample shows code that is generated by the Database Call Wizard:

```
        //calls method which then executes jdbj callto the table
        //selected.
    messages = selectF0101(context, internalVO, maxRows);
```

The wizard creates a generic method. You modify the signature of the method and complete the code for the objects that will be accessed for mapping to and from the database operation call. The wizard creates InputVOType as a placeholder in the signature for the internal value object class name that you provide.

The wizard generates unique code for each type of database operation.

See Also:

- Understanding Database Operations.
- "Understanding Database Operation Calls" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

5.6 Calling Other Business Services

A method in one business service can call a method in another business service. For example, SupplierProcessor.addSupplier could call AddressBookProcessor.addAddressBook or AddressBookProcessor.addAddressBook could call PhonesProcessor.addPhones.

In this code sample, the PhonesProcessor.addPhones method takes an internalProcessPhones value object; this object is created and populated before calling the method:

```
//RI:  Business service call to business service
        //call PhonesProcessor
        //only call phones processor if phones exist.
        if (internalVO.getInternalPhones() != null) {
        //create new internalVO for phones processor
           InternalProcessPhone phones = new InternalProcessPhone();
           //map data from internalVO to phones processor internalVO
           phones.setMnAddressBookNumber(internalVO.getMnAddressBook
Number());
           phones.setPhones(internalVO.getInternalPhones());
           phones.setSzProgramId(programId);
           //call phones processor to add phones
           E1MessageList phonesMessages =
           RI_PhonesProcessor.addPhones(context, connection, phones);
           //If errors occur, change the error type to WARNING because
           //we don't want to stop processing of Address Book record due
           //to error while adding phones, interpret as warning instead.
           if (phonesMessages.hasErrors()) {
              phonesMessages.changeMessageType(E1Message.ERROR_MSG_TYPE,
                                               E1Message.WARNING_MSG_TYPE);
              //set list of phones to list w/ only added phones.
              internalVO.setInternalPhones(phones.getPhones());
           }
           //add messages returned from phones processor
            messages.addMessages(phonesMessages);
         }
```

A business service method can call a business service utility method. For example, PurchaseOrderProcessor. processPurchaseOrder can call ItemProcessor.processItem and EntityProcessor.processEntity.

This code sample shows a business service call to a business service utility:

```
//RI:  Business service call to business service
    //call business service utility
    //This business service returns a status code, this example will not
    //use the status code to drive functionality, but
    //could be evaluated to change processing.
    InternalEntityUtility utilityEntity = new InternalEntityUtility();
    utilityEntity.setMnAddressBookNumber(internalVO.getMnAddressBook
Number());
    utilityEntity.setSzLongAddressNumber(internalVO.getSzLongAddress
Number());
    utilityEntity.setSzTaxId(internalVO.getSzTaxId());

    E1MessageList entityMessages = EntityProcessor.processEntity(context,
connection, utilityEntity);
    internalVO.setMnAddressBookNumber(utilityEntity.getMnAddressBook
Number());
    internalVO.setSzLongAddressNumber(utilityEntity.getSzLongAddress
```

```
Number());
    internalVO.setSzTaxId(utilityEntity.getSzTaxId());
    //Don't stop processing in case of errors from utility, change type to
    // warning and add them to error collection.
    if(entityMessages.hasErrors())
      entityMessages.changeMessageType(E1Message.ERROR_MSG_TYPE,E1Message.
WARNING_MSG_TYPE);
    //take messages generated from EntityProcessor and add them to the
    //high level value object.
     if (retMessages == null)
     {
       retMessages = entityMessages;
     }
     else
     {
       retMessages.addMessages(entityMessages);
     }
```

5.7 Calling Media Object Operations (Release 9.1 Update 2)

You can create business services that call Media Object operations. You use the business service foundation Create Media Object Call Wizard to create these business service methods. Media Object operations include Insert, Select, List, and Delete. This code sample shows code that is generated by the Create Media Object Call Wizard:

```
messages = insertMediaObject(context, connection, internalVO);
```

The wizard creates a generic method. You modify the signature of the method and complete the code for the objects that will be accessed for mapping to and from the Media Object operation call. The wizard creates `InputVOType` as a placeholder in the signature for the internal value object class name that you provide. The wizard generates unique code for each type of Media Object operation. For more information, see Chapter 7, "Creating Business Services that Call Media Object Operations (Release 9.1 Update 2)."

5.8 Managing Business Service Properties

Business service properties provide a way for you to change a value in a business service method without changing the method code. A business service property consists of a key and a value. The key is the name of the business service property and cannot be changed. You use OMW to create business service properties.

See Also:

- "Understanding Business Service Properties" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.
- "Working with Business Services Properties" in the *JD Edwards EnterpriseOne Tools Object Management Workbench Guide*.

5.8.1 Standard Naming Conventions for the Property Key

You can organize business service properties at the system level or at the business service level. Business service properties defined at the system level are used by more than one business service. Business service properties defined at the business service level are used by only one business service.

5.8.1.1 System-Level Business Service Properties

The naming convention for system-level business service properties, used by multiple business services, is to use SYS followed by a meaningful name that you provide. The naming convention looks like this:

SYS_Free_Form

where Free_Form is a name that you enter.

This is an example of a name for a system-level business service property that enables a user to define the program ID that is to be used by any of the master business functions (MBFs) for processing:

SYS_PROGRAM_ID

5.8.1.2 Business Service Level Business Service Properties

The naming convention for business service-level business service properties, used by only one business service, is to use the BusinessServiceName followed by a meaningful name that you provide. The naming convention looks like this:

BusinessServiceName_Free_Form

This table provides examples of names for business service-level business service properties:

Business Service Property Name	Usage
J0100001_AB_MBF_VERSION	This business service property allows the user to define which processing version to use when running the Address Book MBF when processing from the AddressBook business service.
J0100021_AB_MBF_VERSION	This business service property allows the user to define which processing version to use when running the Address Book MBF when processing from the Customer business service.
J0100021_CUS_MBF_VERSION	This business service property allows the user to define which processing version to use when running the Customer MBF when processing from the Customer business service.
J4200040_BYPASS_BSFN_WARNINGS	This business service property sets a Bypass Warning Flag for sales order processing. If 1, the bypass warning flag is true - treat as warnings, do not stop processing. If 0, the bypass warning flag is false - treat warnings as errors, stop processing.
J4200040_PREFIX_1	This business service property adds prefix text to an error message that is returned from a business function to give more specific context to the error message. For example, if an error is returned for a detail line, the value for the prefix message could be "Line no. sent in:". This text is then concatenated with the line number data and added as a prefix to the error message. See Handling Errors in the Business Service.

5.8.2 Business Service Property Methods

The ServicePropertyAccess class provides two utility methods for accessing property values. These methods are:

- Get property value and return null/blank if no value exists in the database, illustrated in this code sample:

```
getSvcPropertyValue(IContext context, java.lang.String key)

Example:  String processingVersion = ServicePropertyAccess.
getSvcPropertyValue(context,SVC_PROPERTY_AB_MBF_PROC_VERSION );
```

- Get service property value, but if the value is null, use the provided default value, illustrated in this code sample:

```
String getSvcPropertyValue(IContext context, java.lang.String key,
java.lang.String defaultVal)
Example:  String programID = ServicePropertyAccess.getSvcPropertyValue
                    ((Context)context, SVC_PROPERTY_PROGRAM_ID,"BSSV");
```

Both of these methods throw a ServicePropertyException message when the property key is null or does not exist in the database. A business service must call these methods in a try/catch block and catch the ServicePropertyException. You can handle business service property errors by creating a new E1Message object that collects the business service property exception message as well as other errors retrieved from business function calls. The business service returns the E1Message object to its caller, and the exception and error messages can be included in the BusinessServiceException, which is thrown by the published business service. When you create the business service, you determine whether to continue processing if an exception is caught. If you allow processing to continue, a failure (an invalid value was passed because of the ServicePropertyException) could occur in the call to the business function. Including text for the exception offers more information as to why the error occurred.

You can use the code template E1SD – EnterpriseOne Add Call to Service Property with Default Value to generate code that calls the business service property method where a default value is passed. The template generates the code and highlights the fields that you need to change.

You can use this code sample as a model for handling business service properties:

```
   public static final String SVC_PROPERTY_AB_MBF_PROC_VERSION =
"J010010_AB_MBF_PROC_VERSION";
   public static final String SVC_PROPTERY_PROGRAM_ID =
"SYS_PROGRAM_ID";
   ...
              //Call access  Business Service Property to retrieve
              //Program ID and processing Version
              //create string so it can be passed to bsfn call
              String programId = null;
              //Call to return Business Service Properties - if fails
              //to retrieve value, use default and continue.
              try {
                  programId =
                          BusinessServicePropertyAccess.
getSvcPropertyValue(context, SVC_PROPERTY_PROGRAM_ID, "BSSV");
              } catch (BusinessServicePropertyException se) {
                  context.getBSSVLogger().app(context,"@@@Attempt to
retrieve Business Service Property failed", "Verify that key exists
in database as entered.", SVC_PROPERTY_PROGRAM_ID, se);
                  //Create new E1 Message using DD item for business
                  //service property exception.
                  E1Message scMessage = new E1Message(context,
"001FIS", SVC_PROPERTY_PROGRAM_ID);
                  //Add messages to final message list to be returned.
```

```
            messages.addMessage(scMessage);
        }
```

5.9 Handling Errors in the Business Service

The business service object exposes public methods that call business functions or database operations to perform a specific business process. During business processing, the business service captures errors and warnings in an array list and returns this information to the published business service in an E1MessageList object.

5.9.1 Rules

All business services must return an E1MessageList object to the published business service. The E1MessageList object must contain all errors, warnings, and information messages that were collected throughout the business service processing.

5.9.2 Best Practices

When writing code for handling errors, remember these best practices:

- The business service foundation provides methods that you can use to add prefix messages to errors. You should add useful information such as key information or detail line information when returning error messages. If you add a prefix to an E1MessageList object that contains no errors, no prefix will be appended and no error will be thrown.

  This example shows how to add a prefix, which names the business function where the messages occurred, to the message list:

  ```
  bsfnParams.getE1MessageList().setMessagePrefix("AddressBookMasterMBF
  (N0100041): ");
  ```

  If the prefixed text can be translated to another language, use a business service property with this naming convention for the text:

  BSSVname_PREFIX_sequence

  Use this code to attach the business service property as a prefix in an error message:

  ```
  private static final String SVC_PROPERTY_PHONE_ERR_PREFIX =
  "JR010030_PREFIX_1";
  ...
  phonesMessages.setMessagePrefix(SVC_PROPERTY_PHONE_ERR_PREFIX +(i+1));
  ```

- If an error condition that is not handled by the business function call occurs, you can use a business service foundation method to create a new error and add the error to the message list. This can be used when a checked exception is thrown by business service foundation and you want to collect the exception as a message in the E1MessageList. Examples of situations requiring a new E1Message are calling the BSSVDataFormatter utility and retrieving business service properties. Because the alias for the JD Edwards EnterpriseOne error to be returned must be passed to the method, an error data dictionary item must exist in JD Edwards EnterpriseOne.

  This code shows creating a new E1Message:

  ```
  new E1Message(context, "001FIS", PROGRAM_ID);
  ```

5.9.3 Collecting Errors

When multiple business functions are called, a potential exists for several errors and warnings to be returned by the business functions. You should gather all errors and warnings in the E1MessageList object for all of the business functions that are called so that all errors and warnings are sent to the caller.

When a business service calls a business function, the business function collects all style errors, defined as error messages in the JD Edwards EnterpriseOne data dictionary, in an ArrayList. The business function always returns an E1MessageList object to its caller. The E1MessageList object contains an ArrayList of the messages returned from a business function call. If no messages are returned, the ArrayList is empty. To determine the state of the E1MessageList object or to determine whether any errors have occurred, you can use one of these methods to call the E1MessageList:

- hasErrors()
- hasWarning()
- hasInfoMessages()

The business service foundation provides several methods that let you add, remove, change, and append to the ArrayList messages.

This code sample shows how to use hasErrors() to call the E1MessageList:

```
             if (messages.hasErrors()) {
            //Get the string representation of all the messages.
            String error = messages.getMessagesAsString();
            //Throw new BusinessServiceException
            throw new BusinessServiceException(error, context);
        }
```

This code sample shows adding a prefix to an E1MessageList to show where errors occurred in a business function:

```
private static E1MessageList callAddressBookMasterMBF(IContext context,
                                    IConnection connection,
                                    InternalValueObject internalVO,
                                    String programId){

   //create new bsfnParams object
   BSFNParameters bsfnParams = new BSFNParameters();
   //set values for bsfn params based on internal vo attribute values
   bsfnParams.setValue("mnAddressBookNumber",
                       internalVO.getMnAddressBookNumber());
   bsfnParams.setValue("szLongAddressNumber",
                       internalVO.getSzLongAddressNumber());
   bsfnParams.setValue("szTaxId", internalVO.getSzTaxId());
   ...
   //execute the AddressBookMasterMBF business function
   bsfnService.execute(context,connection, "AddressBookMasterMBF",bsfnParams);
   //set internal VO attributes based on values passed back from bsfn
internalVO.setMnAddressBookNumber((MathNumeric)bsfnParams.getValue
("mnAddressBookNumber"));
internalVO.setSzLongAddressNumber((String)bsfnParams.getValue
("szLongAddressNumber"));
   internalVO.setSzTaxId((String)bsfnParams.getValue("szTaxId").
toString());
   internalVO.setSzAlphaName((String)bsfnParams.getValue
("szAlphaName"));
   ...
```

```
bsfnParams.getE1MessageList().setMessagePrefix("AddressBookMasterMBF
(N0100041): ");
    //return any errors, warnings, or informational messages to the
    //caller
   return bsfnParams.getE1MessageList();
```

This code sample shows calling the PhonesMBF within a loop and handling the errors that are being collected:

```
public static E1MessageList addPhones(IContext context, IConnection
connection,
 InternalProcessPhone internalVO){
       E1MessageList retMessages = new E1MessageList();

       E1MessageList phonesMessages;
       //Add All phones passed in
       for (int i = 0; i < internalVO.getPhones().length; i++) {
           phonesMessages = callPhonesMBFtoAdd(context, connection,
internalVO, i);
           //set message prefix to add line number
           phonesMessages.setMessagePrefix("Phone line no. sent in"+
(i+1));
           //collect messages for all phones.
           retMessages.addMessages(phonesMessages);
       }
      //send messages back to caller
      return retMessages;
```

This sample code shows returning the messages to the caller and adding them to the existing message object:

```
  public static E1MessageList addAddressBook(IContext context,
IConnection connection,
  InternalAddAddressBook internalVO){
       E1MessageList retMessages = null;
       ...
            //if no errors in address book, continue and add phones.
       if (retMessages != null && !retMessages.hasErrors()) {
           E1MessageList phonesMessages;
      //RI:  Business service call to business service
           //call PhonesProcessor
           ...
           phonesMessages = PhonesProcessor.addPhones(context,
connection,phones);
           //If errors occur, change the error type to WARNING
           if (phonesMessages != null && phonesMessages.hasErrors()){
               phonesMessages.changeMessageType(E1Message.ERROR_MSG_
TYPE,
E1Message.WARNING_MSG_TYPE);
           }
           if (retMessages == null)
           {
             retMessages = phonesMessages;
           }
           else
           {
             retMessages.addMessages(phonesMessages);
           }
       }
       ....
       return retMessages;
```

5.10 Modifying a Business Service

You can modify a business service providing that the change does not alter the signature or behavior of the published business service. You can change a business service in many ways, and how you change the business service depends on the business service design and the type of change that is required. Any change to a business service should be determined as part of the design process. You should ask yourself these questions to determine whether the modifications affect the published business service:

- Am I adding or removing required fields in the value object?
- Will these changes affect the way the existing published business service behaves?

If the answer is yes, you must create a new business service. You can copy and modify the existing business service to create a new business service.

5.11 Documenting a Business Service

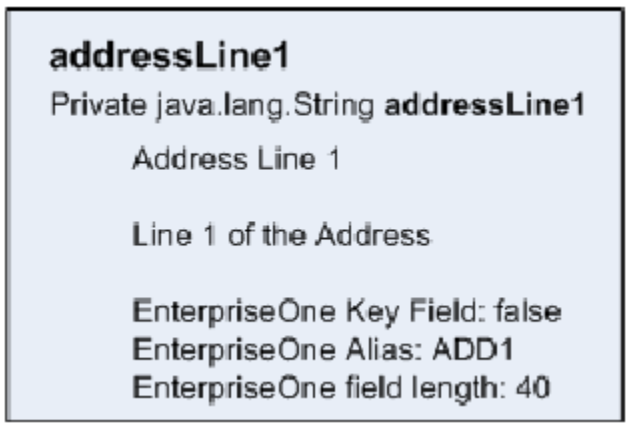
When you create code, use standard Javadoc practices to document both the business service and the published business service classes. Javadoc comments should be added for member variables for all value objects. Most of this is generated by the value object wizards. However, you are responsible for making sure that the description for exposed fields is added and is in context with the business process that is being supported.

This code is an example of Javadoc for a member variable:

```
/**
 * Address Line 1
 * Line 1 of the Address.
 * EnterpriseOne Key field: false
 * EnterpriseOne Alias: add1
 * EnterpriseOne field length:  40
 */
 private String addressLine1 = null;
```

This documentation is a result of the preceding Javadoc:

Figure 5–2 Javadoc documentation



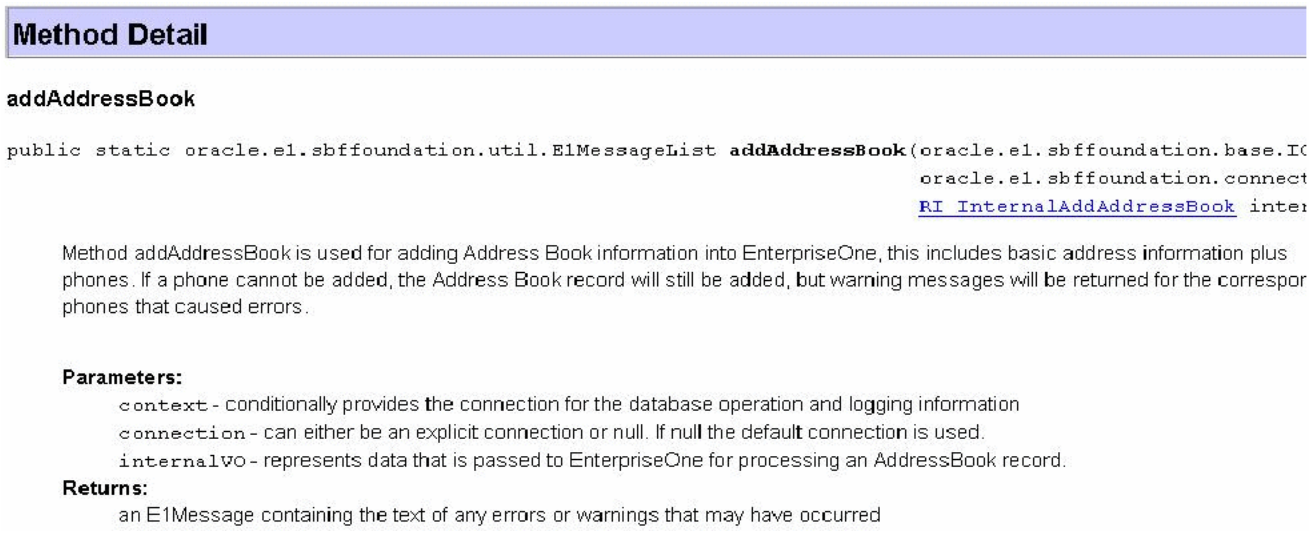
You should include Javadoc comments for all public methods. The behavior of the public methods should also be documented.

This code sample shows how to document a method using Javadoc:

```
/**
  * Method addAddressBook is used for adding Address Book information
  * into EnterpriseOne, this includes basic address information plus
  * phones.  If a phone cannot be added, the Address Book record will
  * still be added, but warning messages will be returned for the
  * corresponding phones that caused errors.
  * @param context conditionally provides the connection for the database
  * operation and logging information
  * @param connection can either be an explicit connection or null. If
  * null the default connection is used.
  * @param internalVO represents data that is passed to EnterpriseOne for
  * processing an AddressBook record.
  * @return an E1Message containing the text of any errors or warnings
  * that may have occurred
  */
 public static E1MessageList addAddressBook(IContext context,
                                     IConnection connection,
                                     InternalAddAddressBook internalVO){
```

This documentation is a result of the preceding Javadoc code:

Figure 5–3 Generated documentation resulting from Javadoc code



Method Detail

addAddressBook

```
public static oracle.e1.sbffoundation.util.E1MessageList addAddressBook(oracle.e1.sbffoundation.base.IC
                                                                        oracle.e1.sbffoundation.connect
                                                                        RI InternalAddAddressBook inter
```

Method addAddressBook is used for adding Address Book information into EnterpriseOne, this includes basic address information plus phones. If a phone cannot be added, the Address Book record will still be added, but warning messages will be returned for the correspor phones that caused errors.

Parameters:

`context` - conditionally provides the connection for the database operation and logging information
`connection` - can either be an explicit connection or null. If null the default connection is used.
`internalVO` - represents data that is passed to EnterpriseOne for processing an AddressBook record.

Returns:

an E1Message containing the text of any errors or warnings that may have occurred

6

Creating Business Services That Call Database Operations

This chapter contains the following topics:

6.1 Understanding Database Operations

Database operations include query, insert, update, and delete. Business services that publish insert, update, and delete database operations should be exposed for staging tables only. Staging tables are Z files (interface tables) that mimic JD Edwards EnterpriseOne tables. Some examples of Z files are F0101Z2 Address Book, F03012Z1 Customer Master, and F0401Z1 Supplier Master. Instead of directly updating a JD Edwards EnterpriseOne database table, data is updated to the appropriate Z file, where batch processes validate the data before updating the database. If you are not using a Z file, you should call a business function to process the data so that proper data validation can be implemented and data integrity maintained.

Many of the rules for business services that call database operations are the same as the rules for business services that call business functions, but some exceptions and differences exist. The exceptions and differences are discussed in this chapter for each of the different types of operations.

6.1.1 Data Types

The data types for the internal value objects for database operations include a long data type as well as all of the data types that are available for business function calls. You use the long data type in a database operation to show how many rows were updated, inserted, or deleted.

This table shows the data types for published value objects that expose database operations:

Published Value Object Data Type	Usage
java.lang.String	Use for string and char fields.

Published Value Object Data Type	Usage
java.util.Calendar	Use for all JDEDate and UTIME fields in JD Edwards EnterpriseOne.
java.lang.Integer	Use for MathNumeric fields defined with 0 decimals, for example, mnAddressNumber and mnShortItemNumber.
java.lang.BigDecimal	Use for MathNumeric fields defined with >0 decimals, for example, mnPurchaseUnitPrice.
java.lang.Boolean	Use for char fields specified only as true/false or 0/1 Boolean fields.
long	Use only in response value object for number of rows returned, number of rows inserted, number of rows updated, number of rows deleted, as returned from the database.

This table shows the data types for internal value objects that expose database operations:

Internal Value Object Data Type	Usage
oracle.e1.bssvfoundation.util.MathNumeric	Use for all fields declared as numeric in JD Edwards EnterpriseOne.
java.lang.Integer	Use for JD Edwards EnterpriseOne ID fields.
java.util.Date	Use for all JDEDate fields.
java.util.GregorianCalendar	Use for UTIME fields in JD Edwards EnterpriseOne.
long	Use only in response value object for number of rows inserted, number of rows updated, number of rows deleted, as returned from the database.

6.1.1.1 Database Exceptions

The code that runs the database operation is generated within a try/catch block and catches a DBServiceException. The business service creates a new E1Message that returns database errors for data dictionary error item 005FIS. When you use the business service foundation code for E1Message, you can create a new message and use the sLineSeparator constant to take advantage of text substitution within the E1Message. The following sample code shows substituting the view name for one parameter and the exception text for the other. Without text substitution, the E1 DD Error Item Description reads:

Table - &1,&2

This code sample shows using text substitution:

```
"Exception in thread "main"
oracle.e1.bssvfoundation.exception.BusinessServiceException:
Error: Table/View - F0101Z2
Error during database operation: [DUPLICATE_KEY_ERROR] Duplicate key
error obtained for table F0101Z2., at oracle.e1.bssv.JPR01002.AddressBook
StagingManager.insertAddressBookStaging
(AddressBookStagingManager.java:78)
at oracle.e1.bssv.JPR01002.AddressBookStagingManager.insertAddress
BookStaging(AddressBookStagingManager.java:39)
at oracle.e1.bssv.JTR87011.AddressBookStagingTest.testInsertAddress
```

```
BookZTable1Record(AddressBookStagingTest.java:71) at
oracle.e1.bssv.JTR87011.AddressBookStagingTest.main(AddressBook
StagingTest.java:110"
```

This sample shows the code that is generated by business service foundation:

```
       private static final String QUERY_VIEW = "V0101XPI";
       ...
       try {
           //get dbService from context
           IDBService dbService = context.getDBService();
           //execute db select operation
           resultSet = dbService.BSSVDBSelect(context, connection,
"V0101XPI", IDBService.DB_BSVW, selectDistinct,
                maxReturnedRows, selectFields, sortOrder,
whereClause);
       } catch (DBServiceException e) {
           //take some action in response to the database exception
           returnMessages.addMessage(new E1Message(context,
                                                   "005FIS",
                                                   QUERY_VIEW +
E1Message.sLineSeparator+e.getMessage()));
       }
```

6.2 Creating a Query Database Operation Business Service

The query database operation uses the Database wizard Select operation over a table or business view to retrieve records from JD Edwards EnterpriseOne.

6.2.1 Published Value Object for Query

The published interface for a select query database operation requires an input value object and an output value object.

6.2.1.1 Naming Conventions

The naming convention for an input value object is to use the verb *get* to preface the type of data to retrieve, for example, GetAddressBook. The naming convention for an output value object is to use the verb *show* to preface the type of data retrieved, for example, ShowAddressbook.

6.2.1.2 Data Types and Structure

The input value object for a query database operation represents a where clause for the query. The output value object for a query database operation returns the query results in an array.

This code sample shows the structure for the show value object:

```
public class ShowAddressBook extends MessageValueObject implements
Serializable {
   private AddressBook addressBook[];
   ...
}
```

6.2.1.3 Error Handling

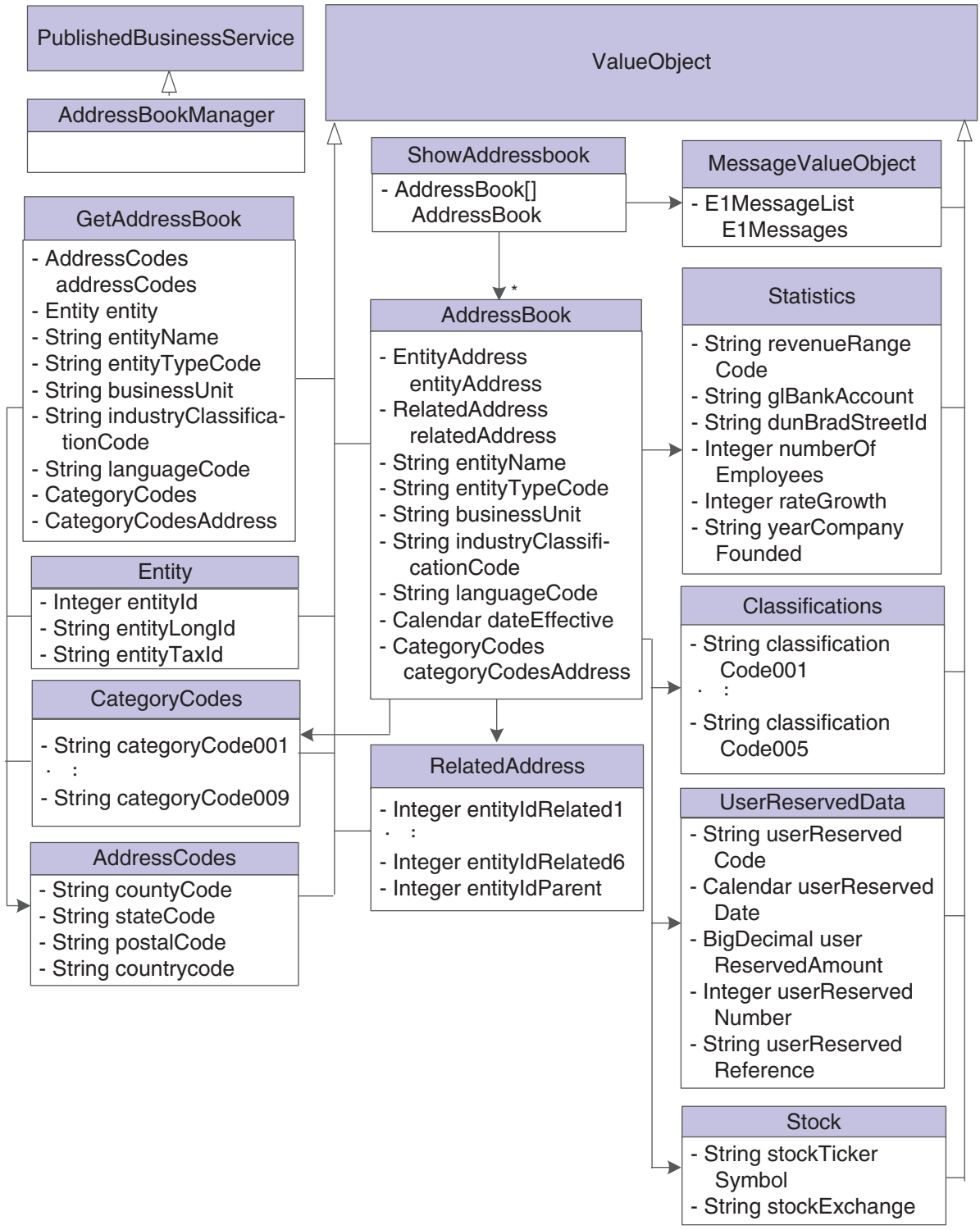
Any warnings that occurred during business service processing are included with the results in the show value object. If an error occurs during processing, the error is returned to the published business service, and the published business service throws an exception. If no results are returned, a message, without an array of records, is returned.

If an error occurs in a utility that is called during the mapping from the published to internal value object, processing should be stopped and the error returned to the published business service, which can throw an exception. For example, if the Entity Processor fails to find entity ID when tax ID is passed in, the query will not process and an error will be returned to the published business service.

6.2.1.4 Class Diagram

The following class diagram shows the published business service objects for GetAddressBook:

Figure 6–1 Published business service, GetAddressBook, class diagram.



6.2.2 Internal Value Object for Query

The internal value object for a query database operation contains two components, the where fields and the result fields.

The names that you use for variables in the internal value object are important because the generated code uses these names when calling the getter and setter methods for these objects.

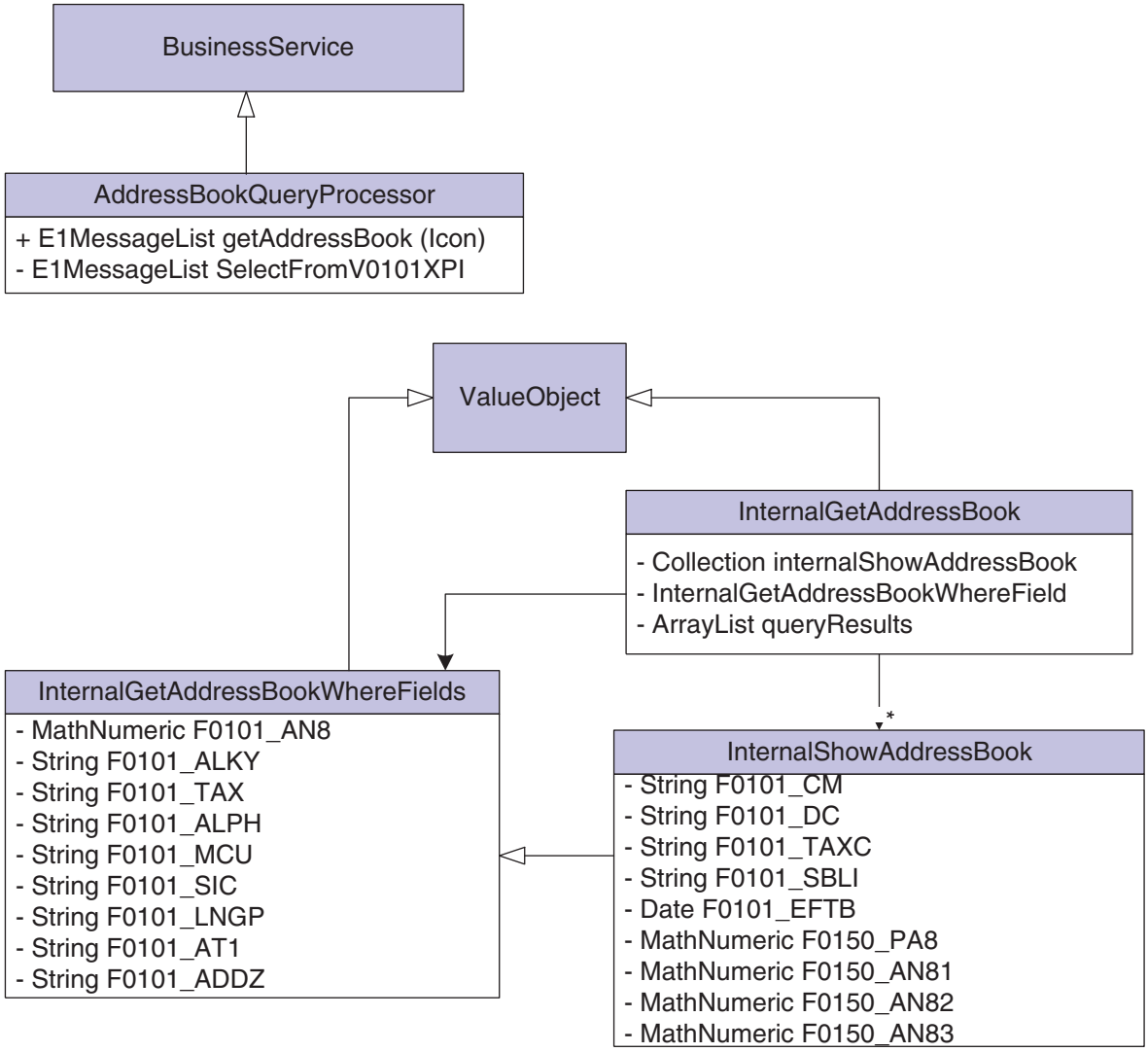
This code sample shows the structure for the internal value object:

```
public class InternalGetAddressBook extends ValueObject{
   private InternalGetAddressBookWhereFields queryWhereFields =
                                new InternalGetAddressBookWhereFields();
   private ArrayList queryResults = null;
   ...
}
```

In the preceding code sample, the variables are named queryResults and queryWhereFields. The queryResults variable represents an array list that contains InternalShowAddressBook type objects. The InternalShowAddressBook value object extends InternalGetAddressBookWhereFields. In the code sample, no additional fields are added to the InternalShowAddressBook value object. However, more fields could be returned from the query than were allowed in the where clause.

This class diagram shows the business service objects for GetAddressBook:

Figure 6–2 Business service, GetAddressbook, class diagram.



6.2.3 Empty Where Clause and Max Rows Returned

Because some tables are too large to return all records without causing significant performance degradation, the recommended practice is to write a select statement that prevents empty where clauses or one that does not select all records. Code that is generated by the wizard follows this recommendation. When you create a query database operation, you must decide whether to allow an empty where clause. If you decide that an empty where clause is appropriate for a particular query, you must modify the generated code to accommodate the empty where clause.

You must include a MaxRowsReturned business service property for all query database operations. This business service property contains the maximum number of rows to be returned to the caller from the selected resultSet variable. The business service property value is passed to the database select statement for processing. If an exception is caught while the system retrieves the business service property, the business service should stop all processing and create an E1MessageList object to pass the exception to the published business service.

Business services interpret a value of 0 (zero) in the business service property to mean return all rows. You must add code to check whether the value returned is zero, and if so, pass a CONSTANT: DBService.DB_FETCH_ALL to the database select call instead of the actual value retrieved. If zero is passed to the select call, an exception will be thrown.

This code sample shows how to check for zero:

```
//Call access property constants for Max Query Rows to be returned.
       //create long variable so it can be passed to bsfn call
       //initialize to 1 in the event, the business service property
       //call fails.
       long maxReturnedRows = 0;
       //Call to return Business Service Property - if fails to
       //retrieve value, use default and continue.
       try{
          maxReturnedRows = Long.parseLong
              (ServicePropertyAccess.getSvcPropertyValue(context,
                                        SVC_PROPERTY_QUERY_MAX_ROWS));
          //interpret property value of zero as "return all rows".
          //Need to send constant to database call.
          if (maxReturnedRows==0){
              maxReturnedRows = DBService.DB_FETCH_ALL;
          }
       }
```

The MaxRowsReturned value does not eliminate the need to check for a null where clause. On a large table, the entire table is selected for processing regardless of how many records are returned to the caller. Because the select statement processes the entire table, performance can be affected.

6.3 Creating an Insert Database Operation Business Service

The insert database operation enables you to add information to a table or business view. You use the Insert database operation in the Database wizard to create an insert business service.

6.3.1 Published Value Object for Insert

The published interface for an insert database operation uses an input value object and an output value object.

6.3.1.1 Naming Conventions

The naming convention for an input value object is to use the verb *insert* to preface the type of data to be processed; for example, *InsertAddressBookStaging*. The naming convention for an output value object is to use the verb phrase *ConfirmInsert* to preface the information that is processed, for example, *ConfirmInsertAddressBookStaging*.

6.3.1.2 Data Types and Structure

The input value object for an insert database operation represents a data set to be inserted into a table. The output value object returns messages and the number of records inserted, which is represented as a long data type. The output value object also returns any warnings that occurred during business service processing. If an error occurs during processing, an error message is sent to the published business service, and the published business service throws an exception.

6.3.1.3 Class Diagram

The following class diagram shows the published business service objects for InsertAddressBookStaging:

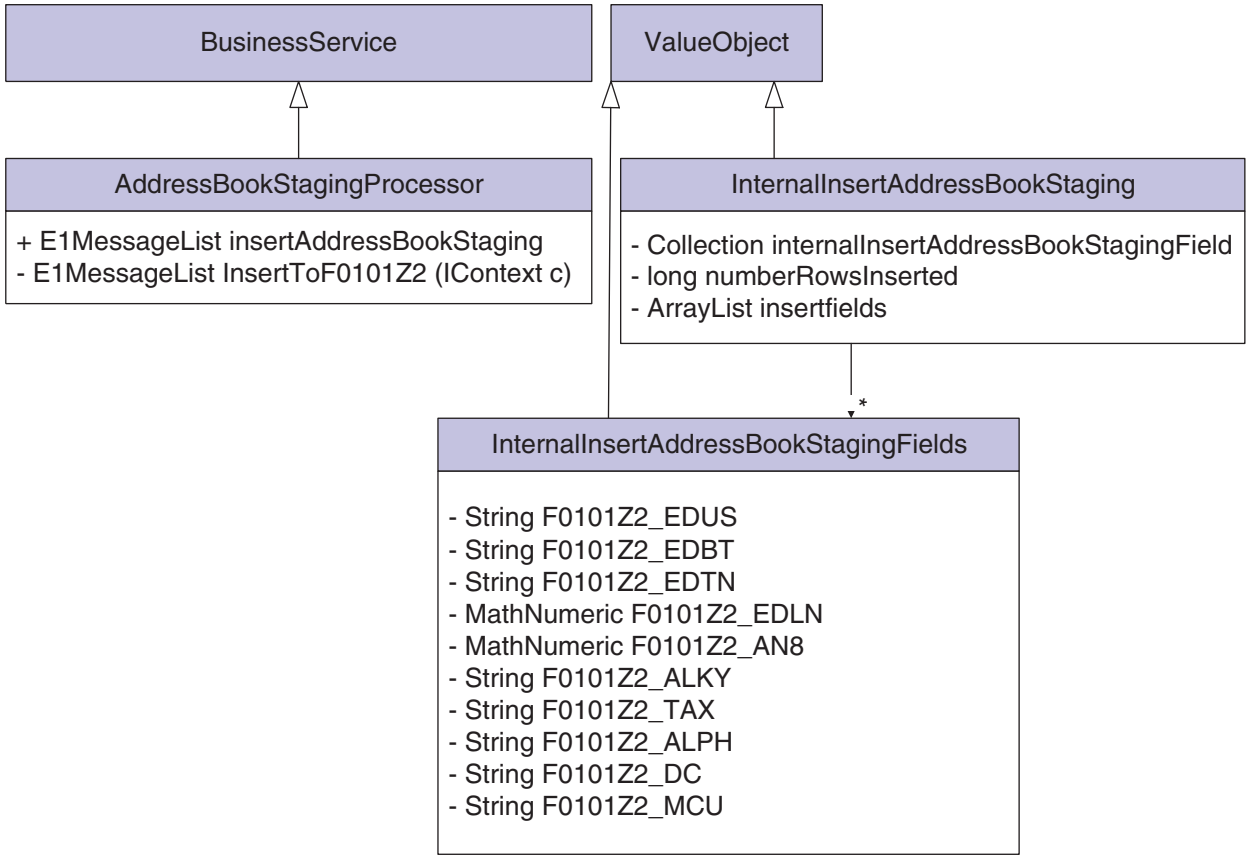
Figure 6–3 Published business service, InsertAddressBookStaging, class diagram.

6.3.2 Internal Value Object for Insert

The internal value object for an insert database operation includes an array list of records that need to be inserted. The array list contains a collection of InternalInsertAddressBook StagingFields objects.

The following class diagram shows the business service objects for InternalInsertAddressBook Staging:

Figure 6–4 Business service, InternalInsertAddressBookStaging, class diagram.



6.3.3 Inserting Multiple Records

The business service method handles multiple records for an insert database operation; however, the generated code inserts one record at a time.

This code sample shows the business service method handling multiple records:

```
public static E1MessageList insertAddressBookStaging(IContext context,
                        IConnection connection,
                        InternalInsertAddressBookStaging internalVO){
    //Call start internal method, passing the context (which was passed
    //from published business service).
       startInternalMethod(context, "insertAddressBookZTable",
internalVO);
       //Create new message list for business service processing.
       E1MessageList messages = new E1MessageList();
       long numRowsInserted = 0;
       if (internalVO.getInsertFields()!=null) {
           for (int i = 0; i < internalVO.getInsertFields().size(); i++)
```

```
{
            //call method (created by the wizard), which then
            //executes Business Function or Database operation
            E1MessageList insertMessages =
                InsertToF0101Z2(context, connection,
                                internalVO.getInsertFields(i));
            //add messages returned from E1 processing to business
            //service message list.
            messages.addMessages(insertMessages);
            //if no errors occur while inserting, add to counter.
            if (!insertMessages.hasErrors()) {
                numRowsInserted++;
            }
        }
        internalVO.setNumberRowsInserted(numRowsInserted);
    }
    //Call finish internal method passing context.
    finishInternalMethod(context, "insertAddressBookZTable");
    //Return E1MessageList containing errors and warnings that
    //occurred during processing business service.
    return messages;
```

This code sample shows the generated code for the database insert:

```
private static E1MessageList InsertToF0101Z2(IContext context,
IConnection connection, InternalInsertAddressBookStagingFields
internalVO) {
    //create return object
    E1MessageList returnMessages = new E1MessageList();
    //specify columns to insert
    BSSVDBField[] insertFields =
    {new BSSVDBField("F0101Z2.EDUS"), // String - EdiUserId
     new BSSVDBField("F0101Z2.EDBT"), // String - EdiBatchNumber
     new BSSVDBField("F0101Z2.EDTN"), // String - EdiTransactNumber
     new BSSVDBField("F0101Z2.EDLN"), // Numeric - EdiLineNumber
     new BSSVDBField("F0101Z2.AN8"),  // Numeric - AddressNumber
     new BSSVDBField("F0101Z2.ALKY"), // String - AlternateAddressKey
     new BSSVDBField("F0101Z2.TAX"),  // String - TaxId
     new BSSVDBField("F0101Z2.ALPH"), // String - NameAlpha
     new BSSVDBField("F0101Z2.DC"),   // String - DescripCompressed
     new BSSVDBField("F0101Z2.MCU")   // String - CostCenter
     };
    //specify insert values
    Object[] insertValues =
    {internalVO.getF0101Z2_EDUS(),
     internalVO.getF0101Z2_EDBT(),
     internalVO.getF0101Z2_EDTN(),
     internalVO.getF0101Z2_EDLN(),
     internalVO.getF0101Z2_AN8(),
     internalVO.getF0101Z2_ALKY(),
     internalVO.getF0101Z2_TAX(),
     internalVO.getF0101Z2_ALPH(),
     internalVO.getF0101Z2_DC(),
     internalVO.getF0101Z2_MCU()
     };
    try {
        //get dbService from context
        IDBService dbService = context.getDBService();
        //execute db insert operation
        long numRecordsInserted =
            dbService.BSSVDBInsert(context, connection, "F0101Z2",
```

```
IDBService.DB_TABLE, insertFields, insertValues);
       } catch (DBServiceException e) {
           //take some action in response to the database exception
           returnMessages.addMessage(new E1Message(context, "005FIS",
TABLE_NAME + E1Message.sLineSeparator+e.getMessage()));
       }
       return returnMessages;
   }
```

6.4 Creating an Update Database Operation Business Service

The update database operation enables you to modify existing information in a table or business view. You use the Update database operation in the Database wizard to create an update business service.

6.4.1 Published Value Object for Update

The published interface for an Update database operation uses an input value object and an output value object.

6.4.1.1 Naming Conventions

The naming convention for an update value object is to use the verb *update* to preface the type of data to be processed, for example, *UpdateAddressBookStaging*. The naming convention for an output value object is to use the verb phrase *ConfirmUpdate* to preface the information that is processed, for example, *ConfirmUpdateAddressBookStaging*.

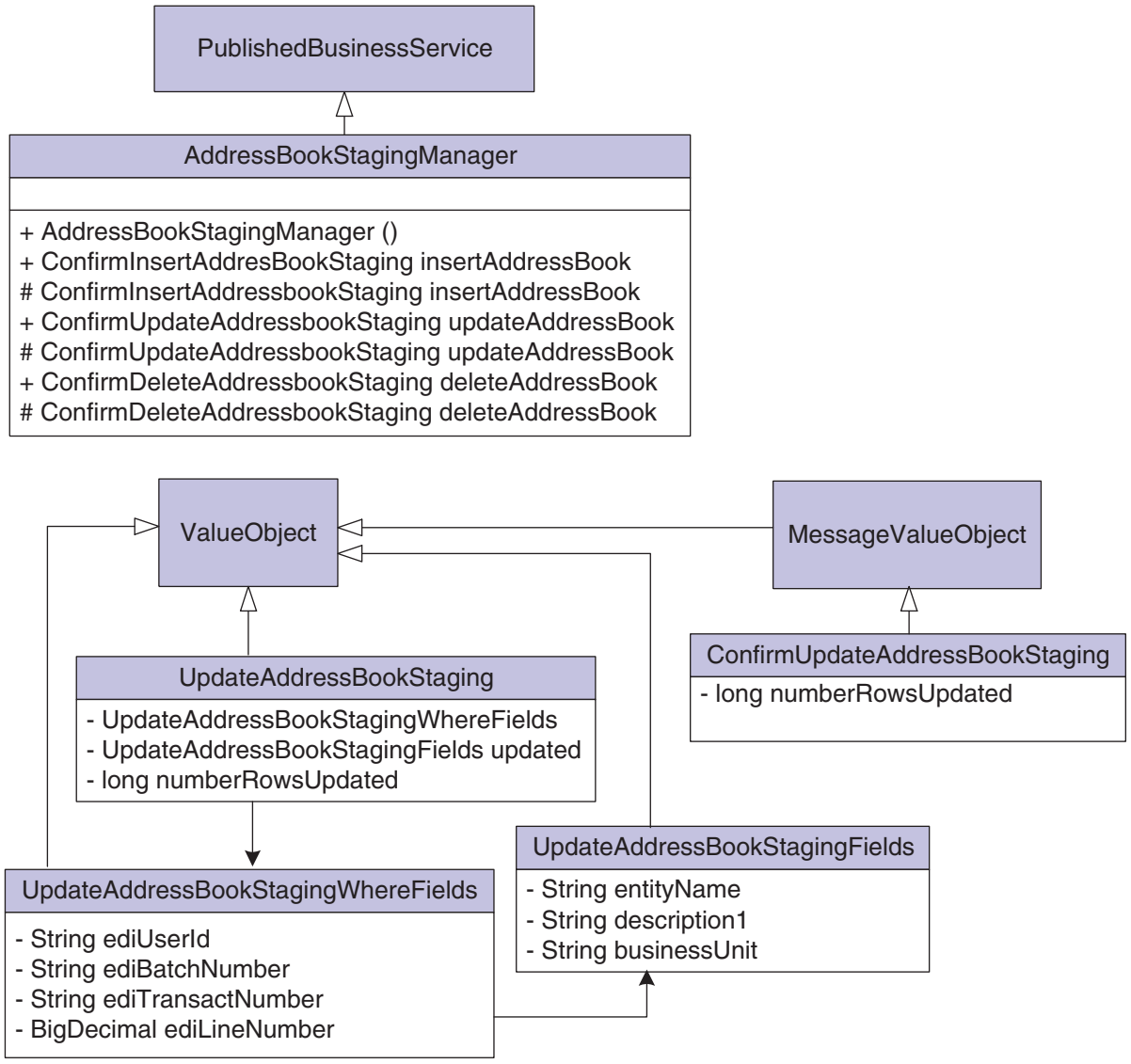
6.4.1.2 Data Types and Structure

The input value object for an update database operation represents a where clause for the records to be updated and the fields that need to be updated for those records. The records and fields are represented by two separate components under the main value object class. The output value object returns messages about the processing that occurred and the number of records updated, which is represented as a long data type. The output value object also returns any warnings that occurred during business service processing. If an error occurs during processing, an error message is sent to the published business service, and the published business service throws an exception.

6.4.1.3 Class Diagram

This class diagram shows the published business service objects for UpdateAddressBookStaging:

Figure 6–5 Published business service, UpdateAddressBookStaging, class diagram.



6.4.2 Internal Value Object for Update

The internal value object for an update database operation contains a component that represents the where clause for the records to be updated and a component that represents the fields to be updated. The variable names updateWhereFields and updateFields for these components are important because the generated code assumes that the proper naming convention is used. The generated code should require minimal changes, if any.

This code sample shows the structure for the internal value object:

```
public class InternalUpdateAddressBookStaging extends ValueObject {
   /**
    * Internal VO representing the where clause for updating the
    * F0101Z2 table.
    */
   private InternalUpdateAddressBookStagingWhereFields
updateWhereFields = new InternalUpdateAddressBookStagingWhereFields();
```

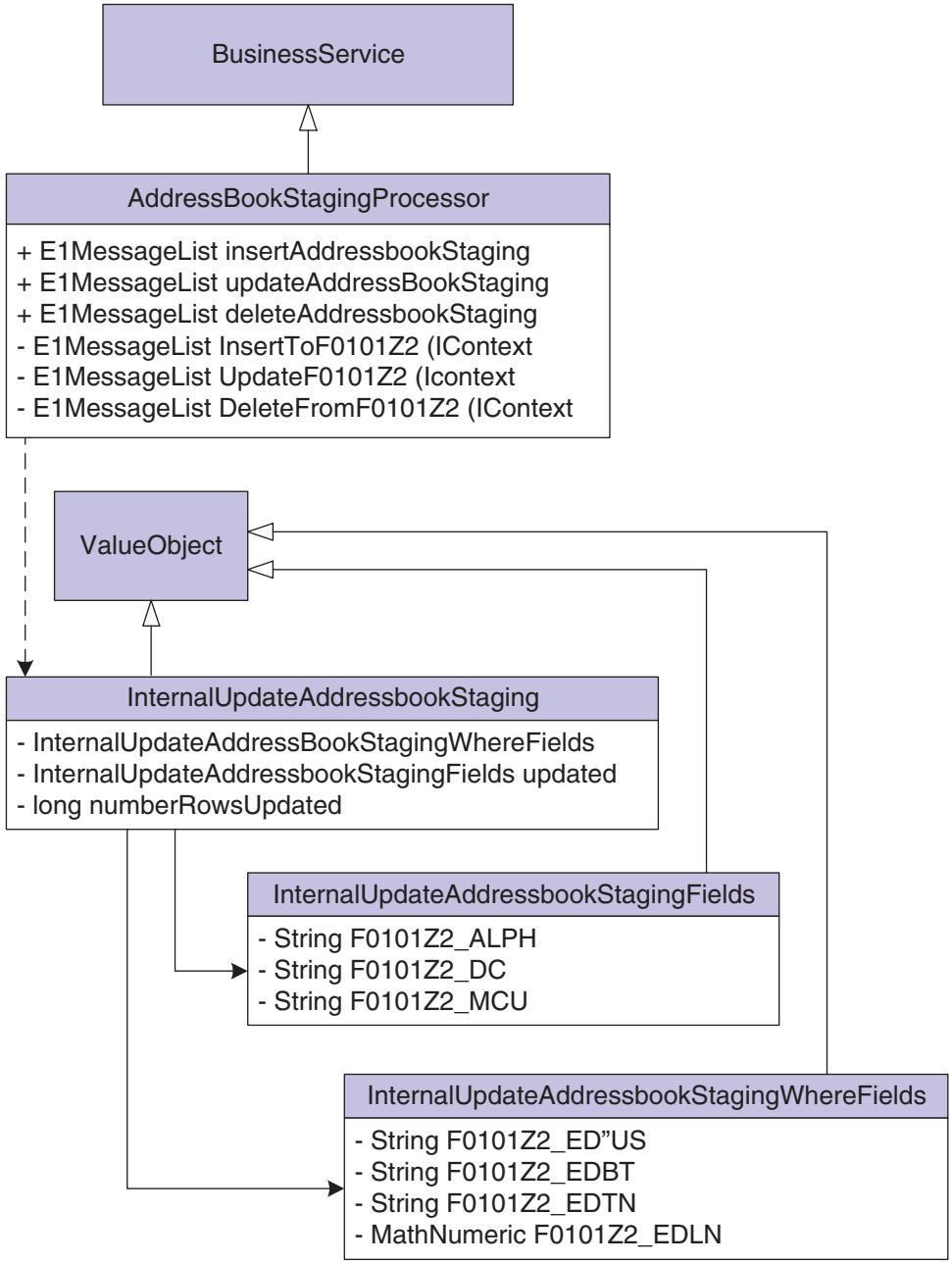
```
   /**
    * Internal VO representing the fields to be updated in the F0101Z2
    * table.
    */
   private InternalUpdateAddressBookStagingFields updateFields = new
InternalUpdateAddressBookStagingFields();
   /**
    * Number of rows updated as returned by the database call.
    */
   private long numberRowsUpdated = 0;
```

This code sample shows the generated code for the update database operation, with the updates that you are required to make in bold type:

```
    private static E1MessageList UpdateF0101Z2(IContext context,
IConnection connection, InternalUpdateAddressBookStaging internalVO) {
        //create return object
       E1MessageList returnMessages = new E1MessageList();
       //specify columns to update
       BSSVDBField[] updateFields =
       {new BSSVDBField("F0101Z2.ALPH"), // String - NameAlpha
        new BSSVDBField("F0101Z2.DC"), // String - DescripCompressed
        new BSSVDBField("F0101Z2.MCU") // String - CostCenter
        };
       //specify update values
       Object[] updateValues =
       {internalVO.getUpdateFields().getF0101Z2_ALPH(),
        internalVO.getUpdateFields().getF0101Z2_DC(),
        internalVO.getUpdateFields().getF0101Z2_MCU()
        };
       //specify condition records must meet to be updated
       BSDBWhereField[] whereFields =
       {new BSDBWhereField(null, new BSSVDBField("F0101Z2.EDUS"),
IDBService.EQUALS, internalVO.getUpdateWhereFields().getF0101Z2_EDUS()),
        new BSDBWhereField(IDBService.AND, new BSSVDBField("F0101Z2.
EDBT"),
IDBService.EQUALS, internalVO.getUpdateWhereFields().getF0101Z2_EDBT()),
        new BSDBWhereField(IDBService.AND, new BSSVDBField("F0101Z2.
EDTN"),
IDBService.EQUALS, internalVO.getUpdateWhereFields().getF0101Z2_EDTN()),
        new BSDBWhereField(IDBService.AND, new BSSVDBField("F0101Z2.
EDLN"),
IDBService.EQUALS, internalVO.getUpdateWhereFields().
getF0101Z2_EDLN())};
       BSSVDBWhereClauseBuilder whereClause =
           new BSSVDBWhereClauseBuilder(context, whereFields);
       try {
           //get dbService from context
           IDBService dbService = context.getDBService();
           //execute db update operation
           long numRecordsUpdated =
               dbService.BSSVDBUpdate(context, connection, "F0101Z2",
IDBService.DB_TABLE, updateFields, updateValues, whereClause);
           internalVO.setNumberRowsUpdated(numRecordsUpdated);
       } catch (DBServiceException e) {
           // take some action in response to the database exception
           returnMessages.addMessage(new E1Message(context, "005FIS",
TABLE_NAME + E1Message.sLineSeparator+e.getMessage())); }
       return returnMessages;
   }
```

This class diagram shows the business service objects for UpdateAddressBookStaging:

Figure 6–6 Business service, UpdateAddressBookStaging, class diagram.



6.5 Creating a Delete Database Operation Business Service

The delete database operation enables you to remove information in a table or business view. You use the Delete database operation in the Database wizard to create a delete business service.

6.5.1 Published Value Object for Delete

The published interface for a delete database operation uses an input value object and an output value object.

6.5.1.1 Naming Conventions

The naming convention for a delete input value object is to use the verb *delete* to preface the type of data to be processed, for example, *DeleteAddressBookStaging*. The naming convention for the delete output value object is to use the verb phrase *ConfirmDelete* to preface the type of data processed, for example, *ConfirmDeleteAddressBookStaging*.

6.5.1.2 Data Types and Structure

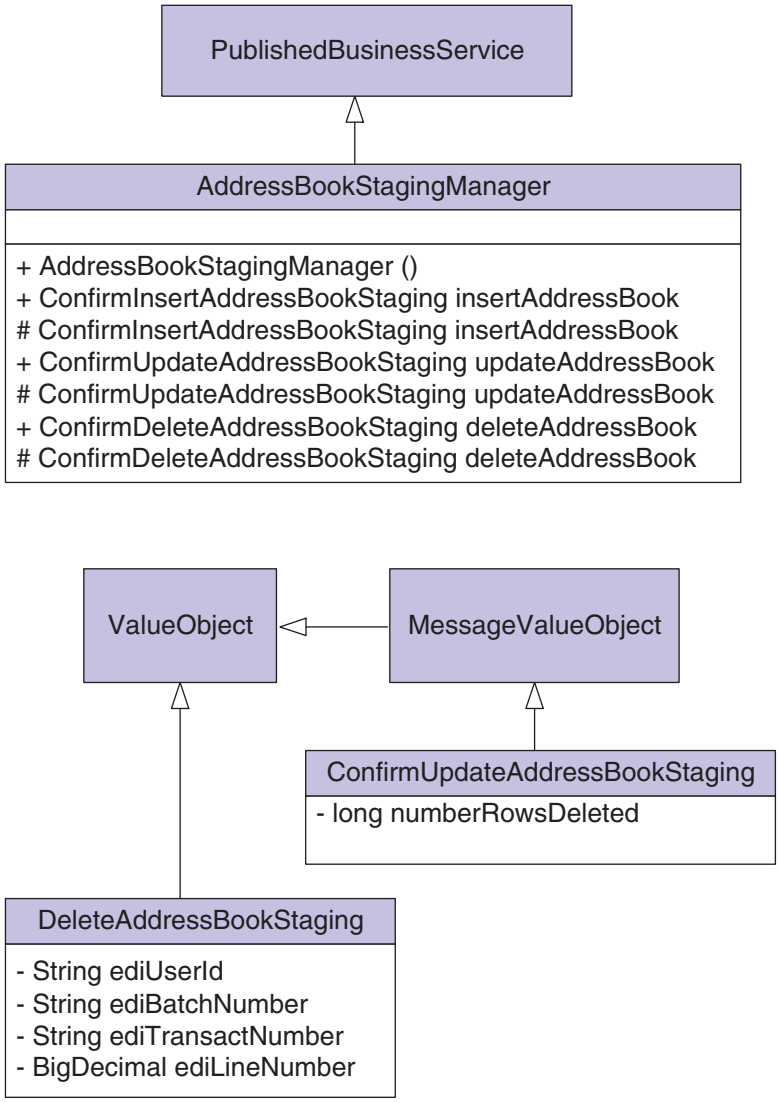
The input value object for a delete database operation represents a where clause for the records to be deleted. The input value object contains key fields to the table or business view. A value must be passed for each key field so that only one record at a time is selected for deletion. The where clause is not conditionally created based on whether a value is sent for a field. The delete operation should not be used for deleting all records at once; therefore, do not use a null where clause in the code.

The output value object for a delete database operation returns messages and the number of records deleted, which is represented as a long data type. The output value object also returns any warnings that occurred during business service processing. If an error occurs during processing, an error message is sent to the published business service, and the published business service throws an exception.

6.5.1.3 Class Diagram

This class diagram shows the published business service objects for DeleteAddressBookStaging:

Figure 6–7 Published business service, DeleteAddressBookStaging, class diagram.

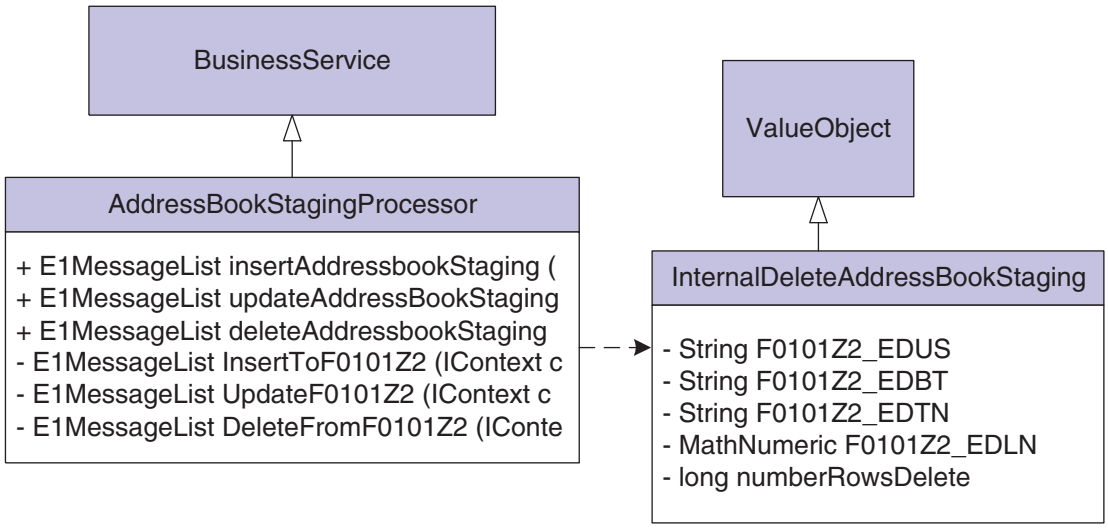


6.5.2 Internal Value Object for Delete

The internal value object for a delete database operation includes the key fields that are required for selecting a record to be deleted and the numberRowsDeleted field.

The following class diagram shows the business service objects for DeleteAddressBookStaging:

Figure 6–8 Business service, DeleteAddressBookStaging, class diagram.



7

Creating Business Services that Call Media Object Operations (Release 9.1 Update 2)

This chapter contains the following topics:

- Section 7.1, "Understanding Media Object Operations"
- Section 7.2, "Creating a Media Object Business Service"

7.1 Understanding Media Object Operations

Many of the rules for business services that call Media Object operations are the same as the rules for business services that call business functions and call database operations, but some exceptions and differences exist. The exceptions and differences are discussed in this chapter for each of the different types of operations.

You can create business services that call Media Object operations. You use the business service foundation Create Media Object Call Wizard to create these business service methods. Media Object operations include Insert, Select, and Delete. This code sample shows code that is generated by the Create Media Object Call Wizard:

```
//calls method which then executes jdbj call to the table selected.
messages = insertMediaObject(context, connection, internalVO);
```

The wizard creates a generic method. You modify the signature of the method and complete the code for the objects that will be accessed for mapping to and from the Media Object operation call. The wizard creates InputVOType as a placeholder in the signature for the internal value object class name that you provide. The wizard generates unique code for each type of Media Object operation.

7.1.1 Data Types

The data types for the internal/published value objects for Media Object operations include data types that are available for business function calls or database operation calls. You use the long data type in a database operation to show how many Media objects were deleted. This table shows the new data type used for published or internal value objects that expose Media Object operations:

Published or Internal Data Type	Usage
javax.activation.DataHandler	Used for Media object attachments transmitted as binary data over a network.
oracle.e1.bssvfoundation.base.MOInfo	Represents the metadata of the media object stored in EnterpriseOne.

7.2 Creating a Media Object Business Service

A Media Object business service contains the business logic to call a Media Object operation such as select, insert, or delete. The java code to call the Media Object operations is automatically generated by the Create Media Object Call Wizard. The code generated for each operation is unique. In the Create Media Object Call Wizard, you select the operation that you plan to implement. In addition, you must create a Media Object value object to call a Media Object operation. You use the Media Object Value Object Class Wizard to create the Media Object value object.

See the following sections in the *JD Edwards EnterpriseOne Business Services Development Guide* for more information:

- "Running the Media Object Value Object Class Wizard"
- "Running the Create Media Object Call Wizard"

7.2.1 Internal Value Object

The internal value object for calling a Media Object operation is created by selecting the internal option in the Media Object Value Object Class Wizard. The same internal value object can be used for calling any of the Media Object operations such as Select, Insert, or Delete. This internal value object will contain an array list of Media Object records that need to be inserted, selected, or deleted.

The following class diagram shows the business service objects for the internal business service RI_AddressBookMediaObjectProcessor:

Figure 7–1 Class diagram of the business service objects for RI_ AddressBookMediaObjectProcessor.

7.2.2 Published Value Object

The published interface for any Media Object operation requires an input value object and an output value object. The published value object for calling any Media Object operation can be created by selecting the "publish" option in the Media Object Value Object Class Wizard. This published Media Object value object is referenced within the published input value object and the output value object classes.

7.2.2.1 Naming Conventions

For the input value object naming convention, use the verb "Insert" to preface the type of data to be processed; for example, InsertAddressBookMO. For the output value object name, use the verb phrase "ConfirmInsert" to preface the information that is processed; for example, ConfirmInsertAddressBookMO or ShowAddressBookMO.

7.2.2.2 Data Types and Structure

The input value object for a Media Object operation represents a data set that is required to perform a particular Media Object operation. The data set includes the Media Object value object. The output value object returns messages and the output data set, such as media object records for a select operation, for a particular operation. The output value object also returns any warnings that occur during business service

processing. If an error occurs during processing, the system sends an error message to the published business service, and the published business service throws an exception.

This code sample shows the structure for the output value object:

```
public class RI_ShowAddressBook extends oracle.e1.bssv.JPR01MO1.MessageValueObject
implements Serializable {
    private ABGT_Publish mediaObject = null;
…
}
```

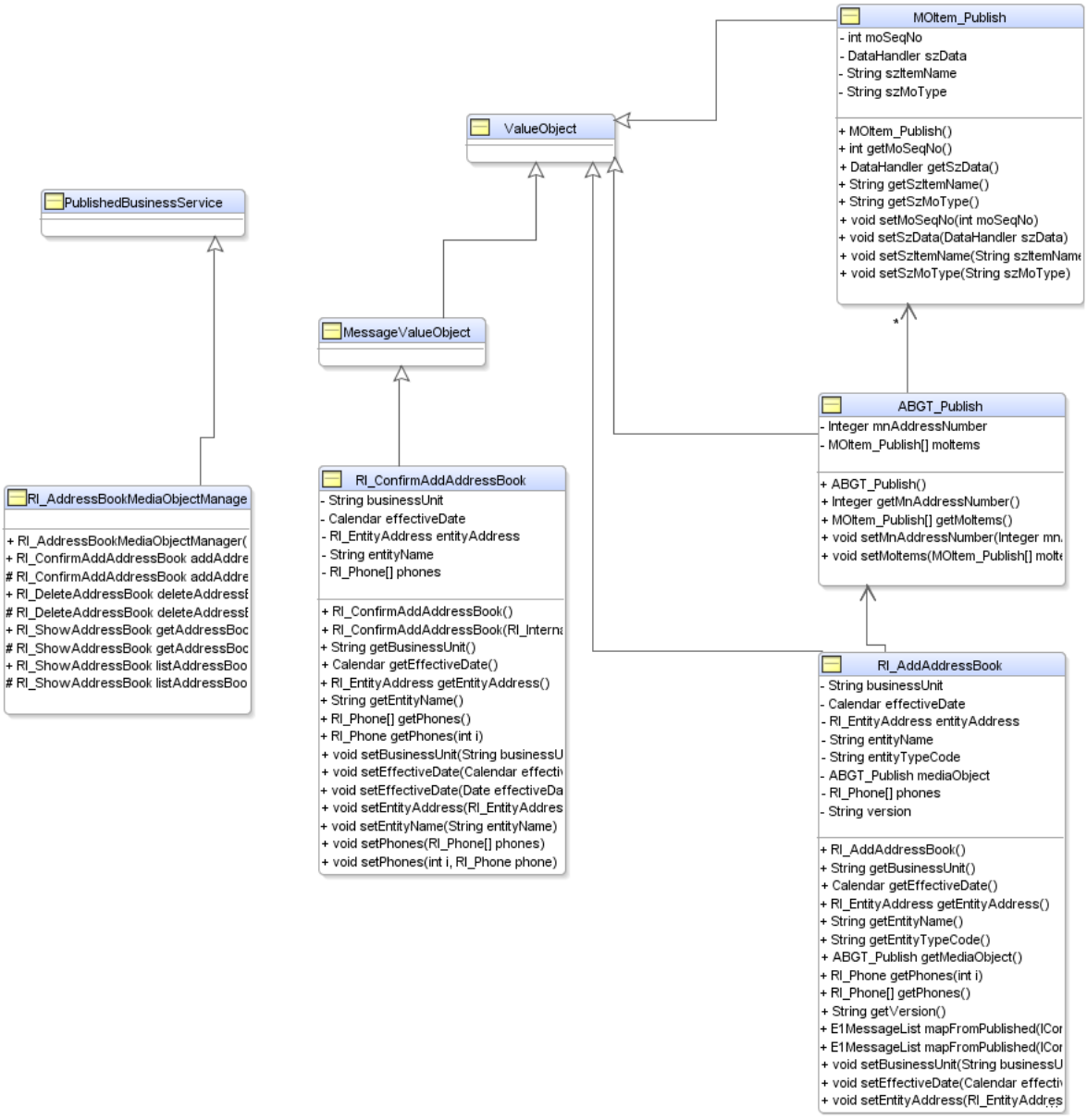
This code sample shows the structure for the input value object:

```
public class RI_AddAddressBook extends oracle.e1.bssv.JPR01MO1.ValueObject
implements Serializable {
    /**    Add the ABGT_Publish class reference to include the Mediaobject fields
in the WSDL*/
        private ABGT_Publish mediaObject = null;
…
}
```

7.2.2.3 Class Diagram

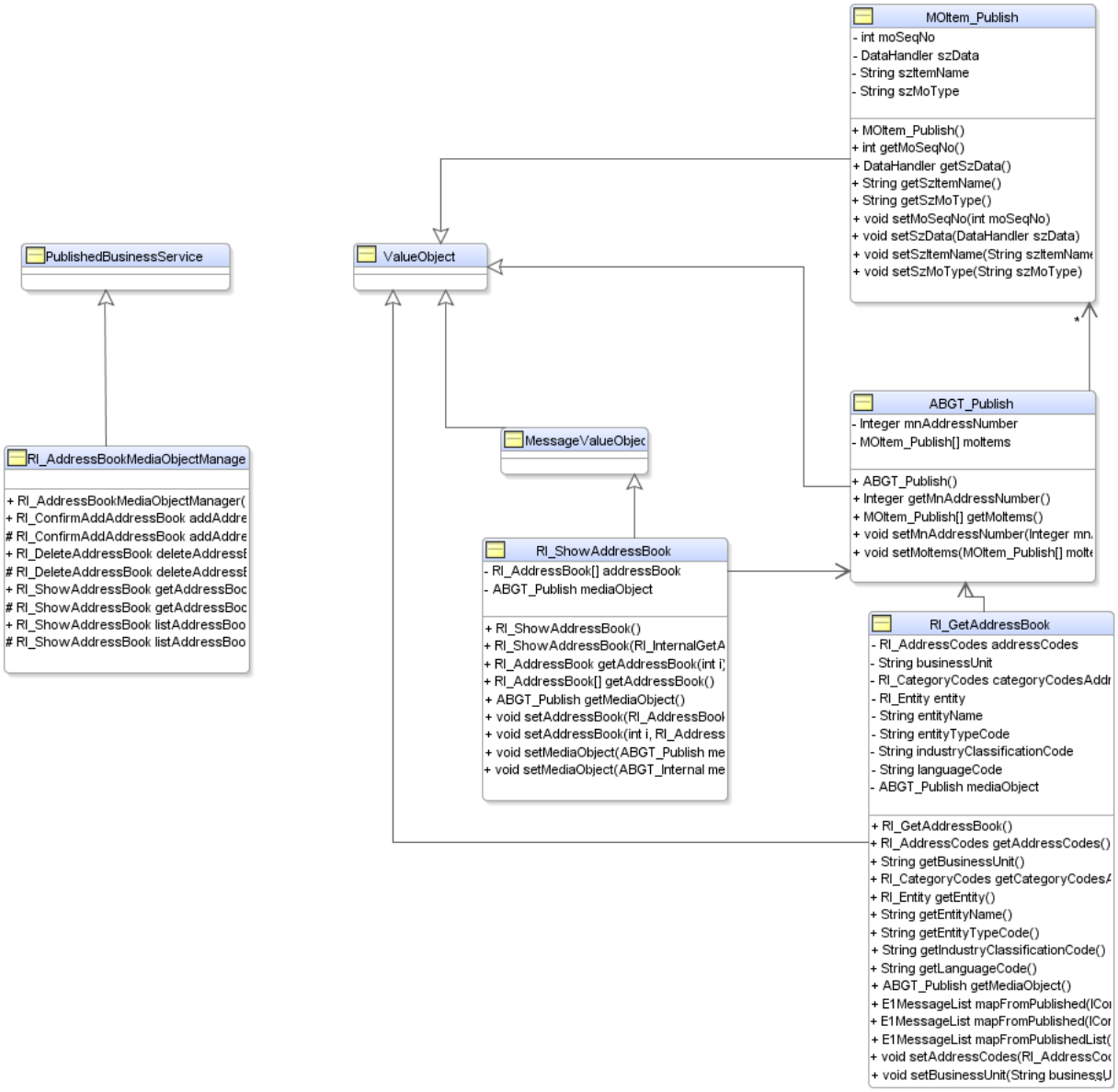
The following class diagram shows the published business service objects for RI_AddressBookMediaObjectManager Insert operation:

Figure 7–2 Class diagram of published business service objects for RI_AddressBookMediaObjectManager Insert operation.



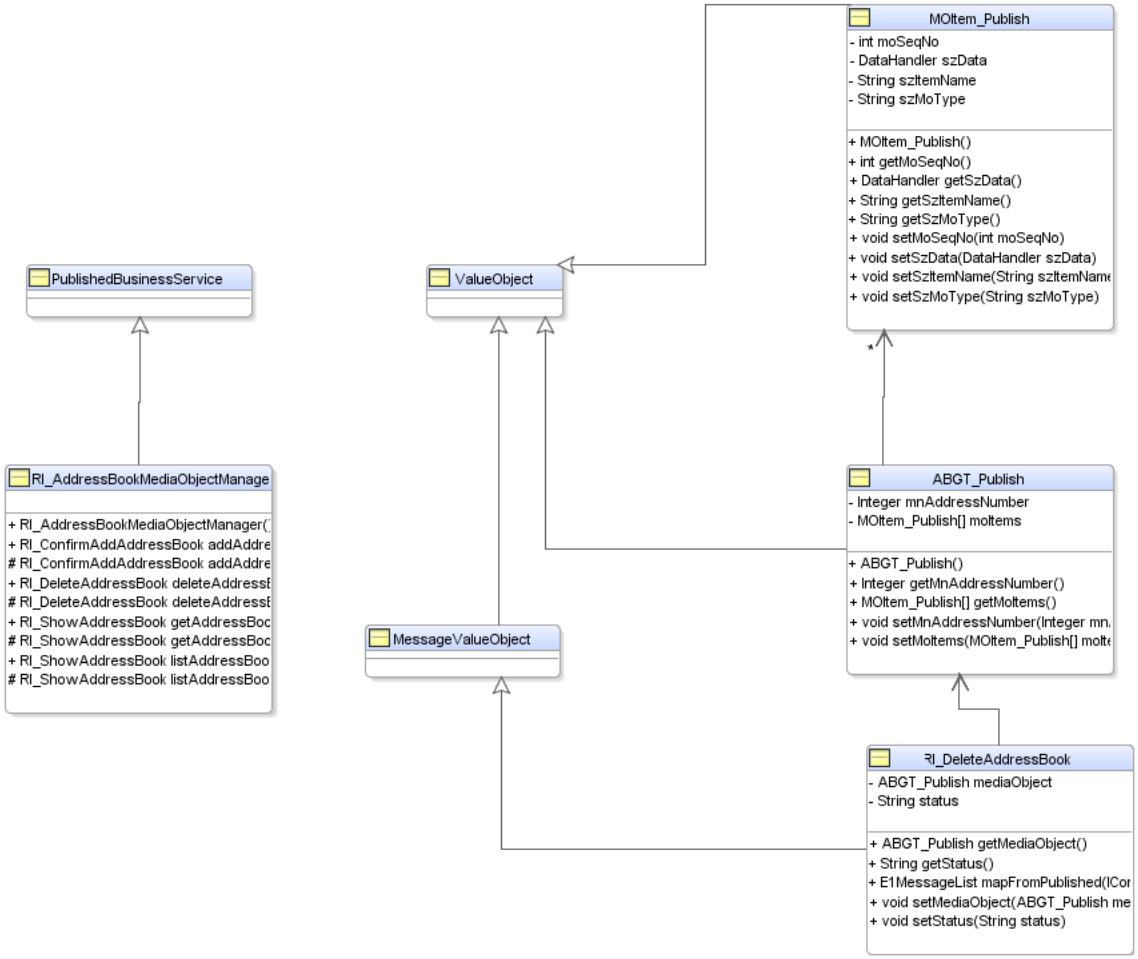
The following class diagram shows the published business service objects for the RI_AddressBookMediaObjectManager Select operation:

Figure 7–3 Class diagram of published business service objects for RI_AddressBookMediaObjectManager Select operation.



The following class diagram shows the published business service objects for the RI_ AddressBookMediaObjectManager Delete operation:

Figure 7–4 Class diagram of published business service objects for RI_AddressBookMediaObjectManager Delete operation.



8

Versioning JD Edwards EnterpriseOne Web Services

This chapter contains the following topics:

Important: This chapter is intended primarily for JD Edwards EnterpriseOne software engineers who design and develop business services and published business services. If you create your own web services, you can use this chapter as a guide for creating versioned web services.

8.1 Overview

When a web service exposes an interface in the form of a WSDL, that interface is assumed to be static from that point on. The published interface for a web service is considered as a service contract, and the methods and inputs to those methods are intended to remain unchanged for the life of that web service. However, over time, it may become necessary to change the behavior or interface of an existing JD Edwards EnterpriseOne web service to provide enhanced processing or to add fields. When you enhance an existing web service, it is very important that you do not change the original methods and interfaces. This chapter provides concepts and procedures for enhancing business services by creating a version of the original business service.

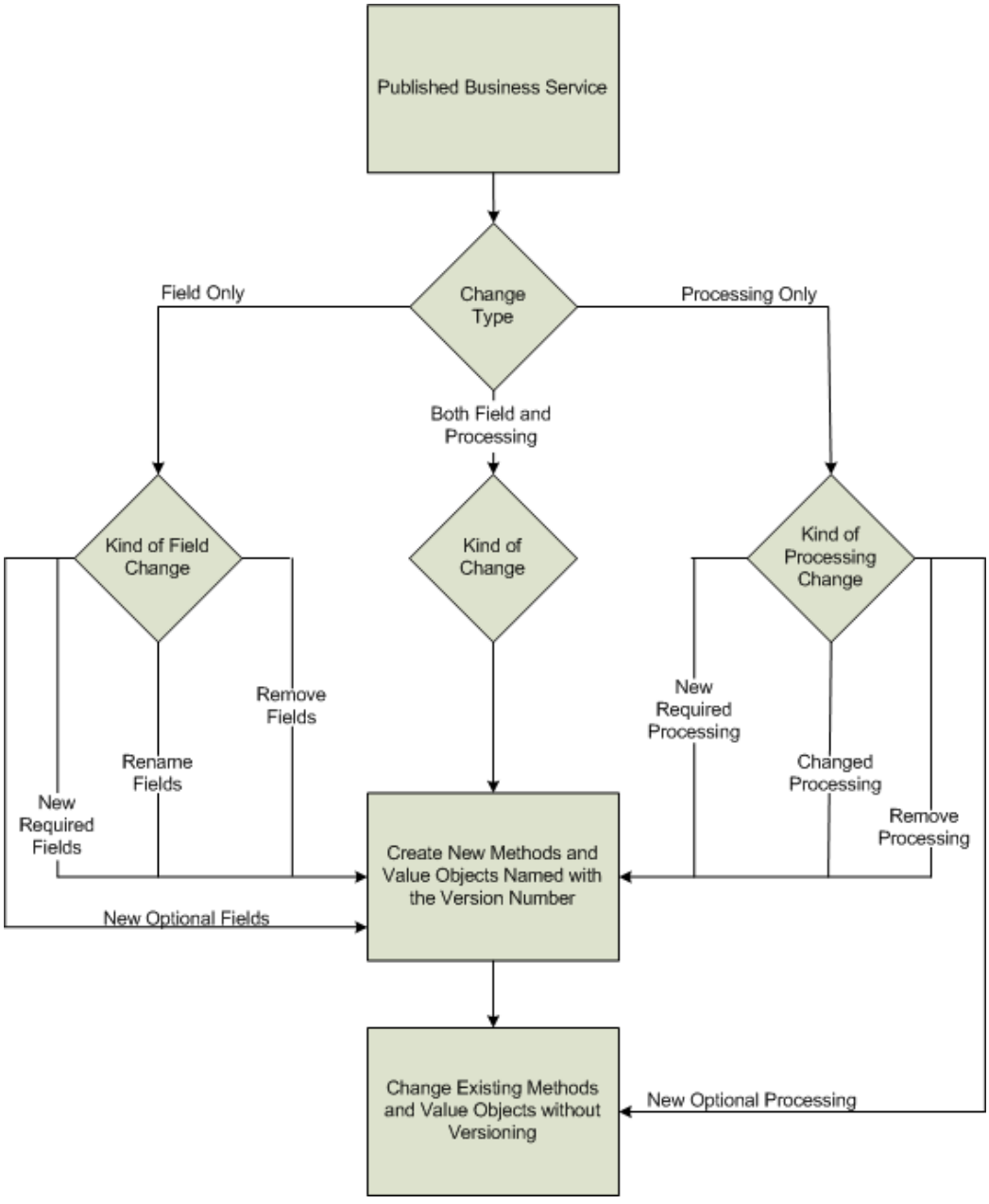
8.2 Published Business Services

JD Edwards EnterpriseOne provides web services, called published business services, for public consumption. The methods and interfaces are exposed in the final web service WSDL. You cannot change the original method names, the original names of the published value object classes, and the original web service behavior without affecting the consumer of the business service. Only new optional processing can be introduced without versioning. Conceptually, optional processing is a kind of invisible change where there is some way (for example, a service constant) to get existing functionality and new functionality from an existing method without changing the interface, as well as maintaining the availability of the original processing.

8.2.1 Determining if Versioning Is Required

Basically, any change to the published interface requires that you create a version of the published business service. The following diagram is provided to help decide whether you should create a version of the original method name or published value object class name or whether the processing is changed:

Figure 8–1 Determine whether to version a published business service



8.2.2 Naming Conventions for Versions

If you determine that you must change the behavior or interface of an existing published business service, you can create a version of the original published business service. When you create a version of a published business service, the name of the versioned published business service must clearly indicate that it is a version of an

original published business service. This enables users of the web service to choose the version with the desired behavior and interface.

Changes requiring versioning require new methods and value objects with a version appended to their name; for example, myMethodV2 and ValueObjectV3. For field changes, you may need to version multiple value objects, depending on the depth of placement of the new fields.

For example, a published business service exposes the method *processAddressBook*. An enhancement request requires that 10 new address book fields be exposed and processed by the method. The new method name will be *processAddressBookV2*. The original value object that will contain the new fields is called *AddressBook*. You copy the original value object, *AddressBook* to a new value object called *AddressBookV2*. Then you create a new top-level value object named *processAddressBookV2* that contains the new version of the value object (*AddressBookV2*) and maps to the new fields.

8.2.3 Creating a Published Business Service Version

The following high-level steps are provided to help you create versions to a published business service:

1. Determine where new and changed fields exist in the published value object.

   Version the containing class and all classes above it in the published value object hierarchy.

2. Version the methods that use the top-level value object in the published *Manager* class.

3. Add and change fields within the internal value object (not a hierarchy.)

4. Add and change internal functionality (business function or input/output calls) in the internal business service.

5. In the new published value object version, change and add mappings in the *mapFromPublished* method.

6. Test both the original and the new version of the business service.

8.2.4 Example: Correct Field Names and Format of Interface

This section provides an example of the process for creating a version of a JD Edwards EnterpriseOne published business service.

This example change involves modifications made in the published business service only. The business service is JP010000. A field is incorrectly named *isEntityTypeNettingIndicator* but it should be *isARAPNettingUsed*. Use these steps to create a version of the published business service.

1. Create a new version of the value object where the field resides.

   a. Create a copy of the *AddressBookResult* value object and name it *AddressBookResultV2*.

   b. Inside *AddressBookResultV2*, change the field *isEntityTypeNettingIndicator* to *isARAPNettingUsed*.

   c. Create a copy of *ShowAddressBook* and name it *ShowAddressBookV2*.

   d. Inside*ShowAddressBookV2*, change all references in *mapFromPublishedMethod* to *ShowAddressBookV2*, including the name change for the *isARAPNettingUsed* field.

2. Create a new published method.
   a. Copy the existing *getAddressBook* method and paste it at the end of the class.
   b. Change the name of the copy to *getAddressBookV2*.
   c. Within *getAddressBookV2*, change all references to the value object *ShowAddressBook* to the new value object *ShowAddressBookV2*.

8.3 Business Services

Business services, commonly called internal business services, perform a specific task. Internal business services do not have a public interface; methods and interfaces are called by published business services. You can change methods and value objects as long as the change to the internal business service does not affect the behavior of the published business service that calls it.

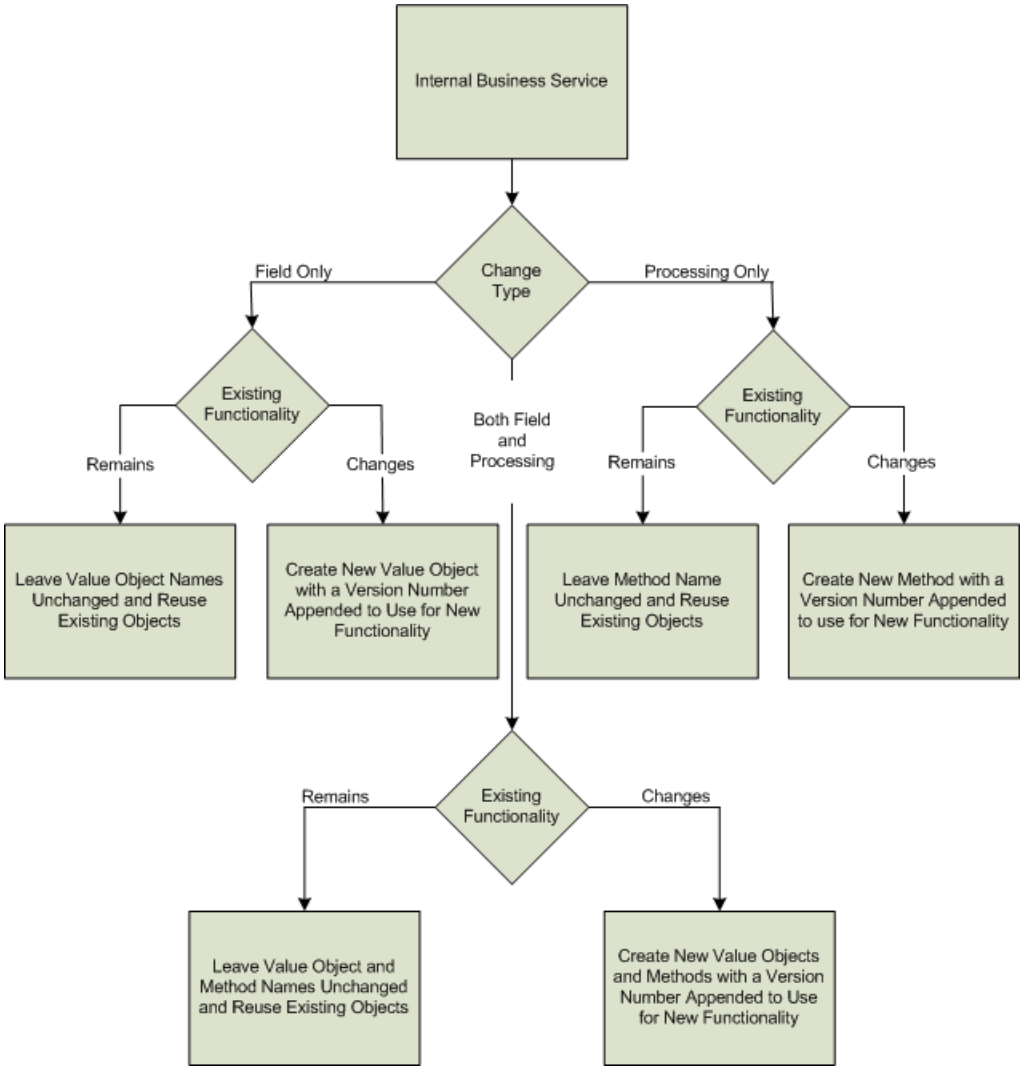
Because the methods and interfaces of internal business services are not public, it is practical that these will be reused, and may be called by both the new version and the existing version of the published business service. The internal business service can provide existing behavior for the existing method while still providing new behavior for the new method; the internal business service does not require renaming or version numbers. However, if the behavior is different, you create a new method or you could copy the original method and append a version number to the method that you copied.

For the internal value object, new non-required fields can be added without affecting the published business service. Typically the internal value object contains all of the fields that could potentially be passed into a business function or input/output call. So it is likely that the field is already included in the internal value object. Fields may be moved from one internal value object to another. You can make these changes to the internal business service without affecting the public interface.

8.3.1 Determining if Versioning is Required

Use the following diagram to help decide whether you should create a version of the original method name and value object class name:

Figure 8–2 Determine whether to version an internal business service



When determining whether to version an internal business service method or internal value object, you should focus on the behavior of the internal business service. The goal is to maintain the existing behavior for the existing published methods while still providing a new behavior.

You may want to employ a technique where a version parameter is passed to the internal function. When called from the original published business service, a value of V1 is passed in the parameter and when called from a new version published business service, a value of V2 is passed. Within the internal business service logic, only new logic is performed if the parameter is V2. This keeps original logic intact while allowing additional functionality for V2.

8.3.2 Example: Enhancement that Includes New Fields and Associated Processing

This section provides an example of the process for creating a version of a JD Edwards EnterpriseOne internal business service and then creating a version of the published business service that calls the internal business service version.

This example change involves modifications made at all levels of the business service Java code. This example is approached from the published interface through the

internal business service to the JD Edwards EnterpriseOne business function calls. The published business service is JP010020 and the internal business service is J0100021.

Use these steps to create a version of the internal business service and the published business service that calls it:

1. Determine where the new fields belong in the value object.

   In this example, the top-level published value object is called *ProcessCustomer*. The fields are related to invoicing information, so the new fields will be updated to the *Invoice* object.

2. In JDeveloper, do the following in the value object folder of the business service:

   a. Create a copy of *ProcessCustomer* and name it *ProcessCustomerV2*.

   b. Create a copy of *Invoice* and name it *InvoiceV2*.

   c. Inside *ProcessCustomerV2*, change the current member reference from *Invoice* to *InvoiceV2*.

   d. Inside *ProcessCustomerV2*, change all references in *mapFromPublishedMethod* to *InvoiceV2*.

3. Create a new published method.

   a. Copy the existing *processCustomer* method and paste it at the end of the class.

   b. Change the name of the copy to *processCustomerV2*.

   c. Within *processCustomerV2*, change all references to the value object *ProcessCustomer* to the new value object *Process CustomerV2*.

4. Evaluate and change the internal business service.

   a. The new fields must be added to the internal business service, too. You can add the new fields to the internal value object, *InternalProcessCustomer*, by just adding them as additional members in the class.

   b. Modify *CustomerProcessor* to pass the new value object fields to *CustomerMBF*, which is already called. Because these are new non-required fields, it does not matter if they are blank, as they would be called from the existing business service. The processing functions as it always has when fields are blank, and when these new fields are passed in, they will be processed as expected.

5. Return to the value object *ProcessCustomerV2* and add new code to *mapFromPublishedMethod* that maps the new published value object fields to the new internal value object fields.

6. Test both the new *processCustomerV2* and the original *processCustomer* methods.

8.4 JD Edwards EnterpriseOne as a Web Service Consumer

JD Edwards EnterpriseOne can consume web services from third-party systems. Although these web services do not expose a contract in the form of published methods and interfaces, they may need to be changed to take advantage of third-party enhancements or new services. Because JD Edwards EnterpriseOne business functions call the methods of the third-party web services, any new version or method must be added as new code that is called by the business function. You must determine how to control which version of the business service the business function calls. You might consider using a processing option or a service constant to control the behavior.

The only reason for changing a business service that consumes a third-party web service is that the third-party web service has changed. The following scenarios

illustrate how to control the behavior of the business services using a processing option or service constant.

Scenario 1: A third party web service has changed--use a processing option

For this scenario, you should version method or value objects by appending a version number to the name. Most likely, the third-party service that changed is also versioned. The new version of the business service method is called directly from the JD Edwards EnterpriseOne business function, which may or may not pass new data to the changed third-party web service. You can create a new JD Edwards EnterpriseOne processing option to control the version of the business service method that is called and the data that is passed to it.

Scenario 2: A third-party web service has changed--use a service constant

An alternative to Scenario 1, is that the existing method could call the new method based on a service constant that controls what version is being called. In this scenario, all of the data passed from the business function must be the same for both versions. This scenario minimizes the impact to existing business function calls while allowing you to control what version of the third-party service is called.

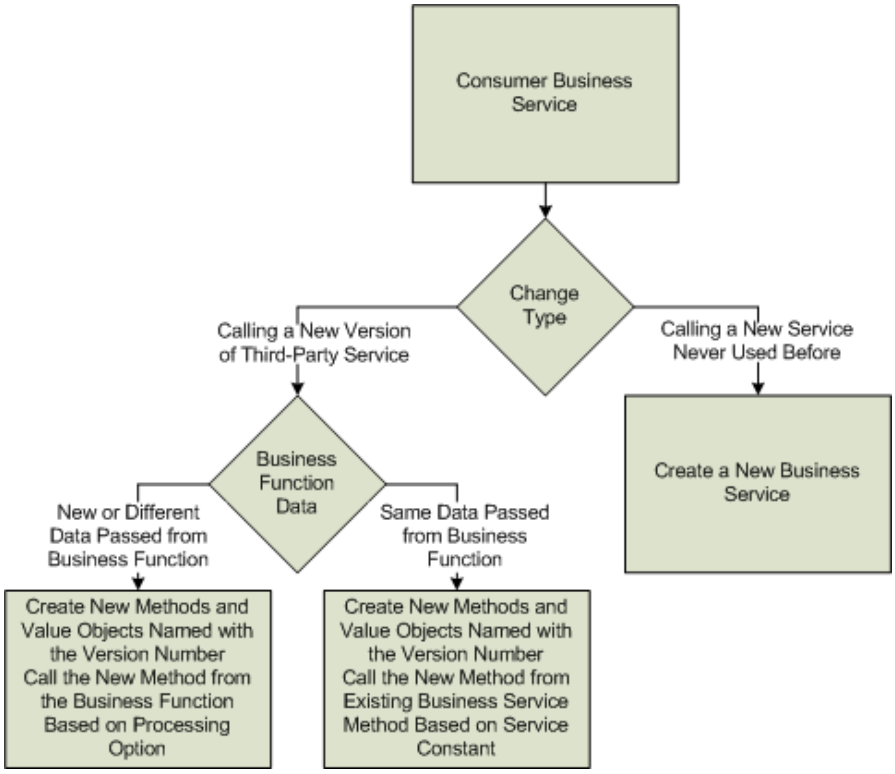
Scenario 3: The consumer business service is calling a free web service that has been updated

You have decided to upgrade the consumer business service to use the new version of a free web service. There will be no impact to users of the consumer business service if the business service starts calling the new version of the free web service without giving the user the option to use the previous version. There is no need to version the consumer business service. You can enhance the JD Edwards EnterpriseOne web service to use the new version of the free web service providing no backward compatibility is required.

8.4.1 Determining if Versioning is Required

Use the following diagram to help decide whether you should create a version of the original method name and value object class name:

Figure 8–3 Determine whether to version a consumer business service



8.4.2 Creating a Version to a Consumer Business Service

The following high-level steps are provided to help you create versions to business services that consume third-party web services:

1. Determine where new and changed fields exist in the value object.

   Version the containing class and all classes above it in the value object hierarchy.

2. Version the method that uses the top-level value object in the *Processor* class.

3. Call the new method from a JD Edwards EnterpriseOne business service and do one of the following:

   - Create a processing option to call the new or old method.
   - Create a service constant to call the new method from the existing method.

8.4.3 Example: Enhancement to Call Latest Version of a Third-Party Service

This section provides an example of the process for creating a version of a JD Edwards EnterpriseOne business service that consumes a third-party web service.

In this example, a JD Edwards EnterpriseOne consumer business service calls a third-party service for weather forecast. Originally, the web service used only a zip code as input, but now it accepts city and state, too. The consumer business service is J8500001.

Use these steps to create a version of the internal business service and the published business service that calls it:

1. Create a new version of the value object to include the new fields; for example:

   Create a copy of *GetWeatherInput* and name it *GetWeatherInputV2*.

2. Within the *WeatherProcessor* class, create a copy of the method *getWeather* and name it *getWeatherV2*.
3. Use the endpoint of the new version of the weather forecast service to create a new proxy for the service, and name the new proxy *ProxyV2*.
4. In the new *getWeatherV2* method, change the code to call the method from the new proxy, *ProxyV2*.
5. To support either version of the service, create processing options or system settings that indicate that city and state can be used.
6. Enhance the JD Edwards EnterpriseOne applications and business function to follow the settings and allow users to enter city and state, as well as to include the values in the XML generated by the business function that calls the business service.
7. Modify *getWeatherV2* to map the new city and state fields from the value object to the input of the web service call.

9

Understanding Transaction Processing

This chapter contains the following topics:

9.1 Transaction Processing

You update the JD Edwards EnterpriseOne database by processing a transaction. A transaction is a logical unit of work performed on the database to complete a common task and maintain data consistency. A transaction consists of transaction statements that are closely related and perform interdependent actions. Each transaction statement performs part of the task, but all of the statements are required for completing the task. Transaction processing ensures that related data is added to or deleted from the database simultaneously, thus preserving data integrity. In transaction processing, data is not written to the database until a commit command is issued. When this happens, data is permanently written to the database. You can use one of these ways to commit transactions:

- Auto commit
- Manual commit

9.1.1 Auto Commit

An auto commit transaction encompasses individual table updates within a business function call or direct database call from the business service. Each individual update is committed or rolled back immediately. The commitment or rollback does not depend on success or failure or any other call. Transaction processing that uses auto commit does not require an explicit call to commit or roll back data. When you use auto commit, you cannot include another business function or database call as part of the transaction for rollback. You cannot include multiple table updates called from within the business function as part of a transaction for rollback.

9.1.2 Manual Commit

When you use manual commit, the record is held until commit or rollback is explicitly called. Business function and database calls can be strung together and committed or rolled back based on success or failure of any one of the calls. Although business function and database operations can be called within the same published business service or business service transaction boundary, two separate connections are created in the background. When you code for these two types of operations, consider that one

should not depend on the other's data. For example, if you are calling insert for a business unit and then you try to add an address book record that contains that business unit, the transaction will fail because the database call hasn't been committed yet.

9.2 Default Transaction Processing Behavior

The business service framework provides two types of default transactions: manual commit connection and auto commit connection.

For a single manual commit transaction, the default behavior is to scope all processing within the published business service method. If any operation within this scope fails, all operations are rolled back, and the published business service method throws an exception. This behavior is recommended when multiple records are committed to multiple tables.

For a single auto commit transaction, the default behavior is that each operation commits or rolls back immediately, which means that each table update within each business function call is either committed or rolled back immediately. This behavior is recommended for queries for which no transaction is needed or when you are committing a single record to a single table.

When you are deciding which type of connection to use, you should always consider the business function behavior.

9.2.1 Published Business Service Boundary for Manual Commit

The startPublishedMethod, finishPublishedMethod, and close methods within the published business service indicate the boundary of the transaction. All activities that occur within the startPublishedMethod and finishPublishedMethod calls will be committed when finishPublishedMethod is called. You must include the close method to clean up all connections.

9.2.2 Published Business Service Boundary for Auto Commit

The startPublishedMethod, finishPublishedMethod, and close methods within the published business service are used to create the auto commit connection and to clean up the connections. All activities that occur within the startPublishedMethod and finishPublishedMethod calls are committed or rolled back immediately because no transaction boundary exists that encompasses more than one operation, including the table updates within the business function. For an auto commit connection, the purpose of finishPublishedMethod is different than for a manual commit because no need exists to commit the transaction. The finishPublishedMethod plays a roll in monitoring and tying the entire business process together. You call the close method to clean up all connections.

For both manual commit and auto commit, you should use a try block to enclose startPublishedMethod and finishPublishedMethod. You call the close method from a *finally* block to ensure that all transactions are finished and no connections linger.

This code sample shows the structure for defining the transaction processing boundary for the published business service:

```
public ConfirmAddAddressBook addAddressBook(AddAddressBook vo) throws
BusinessServiceException {
       return (addAddressBook(null, null, vo));
   }
    protected ConfirmAddAddressBook addAddressBook(IContext context,
```

```
                    IConnection connection,
                    AddAddressBook vo) throws BusinessServiceException{
       //perform all work within try block, finally will clean up any
       //connections
       try {
           // call start published method, passing null,
           //will return context object so BSFN can be called later
           //used to indicate transaction boundary as well as used for
           //logging
           //Start Implicit Transaction
           context = startPublishedMethod(context, "addAddressBook");
           // create a new internal VO.
           InternalAddAddressBook internalVO= new InternalAddAddress
Book();
           messages.addMessages(vo.mapFromPublsihed(context, internal
VO));//
           // start business service addAddressBook passing context and
           // internal VO published business service Calling business
           // service
           E1MessageList messages = AddressBookProcessor.addAddressBook
(context, connection, internalVO);
           // published business service will send either warnings in
           // the Confirm Value Object or throw a published business
           // serviceException.
           if (messages.hasErrors()) {
               // get the string representation of all the messages
                //RI: Error Handling
               String error = messages.getMessagesAsString());
               // Throw BusinessServiceException.(
               throw new BusinessServiceException(error,context);
           }
           // exception was not thrown, so create the confirm VO from
           // internal VO
         ConfirmAddAddressBook confirmVO = new ConfirmAddAddressBook
(internalVO);
           confirmVO.setE1MessageList(messages);
           //Call to commit default transaction.
           finishPublishedMethod(context, "addAddressBook");
           // return confirm VO, filled with return values and messages
           return confirmVO;
       }
       finally {
           //Call close to clean up all remaining connections and
           //resources.
           close(context,"addAddressBook");
       }
  }
```

9.3 Explicit Transaction Processing Behavior

Oracle recommends that you use default transaction behavior whenever possible. However, you can define your published business service or business service to explicitly manage transactions. To handle the transaction correctly in the business service, you must understand the detailed transaction behavior of the business function being called.

The published business service protected method and all business service methods are required to have both IContext and IConnection as part of their signature, as are any

calls to business functions or database operations. If you are using default transaction processing, the connection can be null. If you use explicit transaction processing, you must provide an explicit connection, either auto or manual commit. When you use an explicit connection, you decide whether having multiple transactions is appropriate and whether they are auto commit or manual commit connections. If you create an explicit transaction from your business service, you are not required to check for null on the connection before using it, because the foundation classes ensure that the connection is never null. If the token is dropped, a runtime connection is thrown, which is consistent with the default transaction processing.

9.3.1 Creating a New Connection

You can create a new transaction in either the published business service or business service, depending on where control begins. Typically, the business service controls the transaction. The context object has exposure to all connections; so to create a new connection, you call a method from the context object. You create either a manual commit or an auto commit method. Both methods are illustrated in this code sample:

```
IConnection soConnection = context.getNewConnection(IConnection.MANUAL)
;);
//manual commit
IConnection soConnection = context.getNewConnection(IConnection.AUTO);
//auto commit
```

A manual commit method holds the record until commit or rollback is explicitly called. You create a manual commit method by using IConnection.MANUAL (false) as the parameter in the context object. An auto commit method commits the record immediately without an explicit call to commit the record. With auto commit, the record is committed when the Close method is called. You create an auto commit method by using IConnection.AUTO (true) as the parameter in the context object.

The default connection is available even when an explicit connection is created.

9.3.2 Using an Explicit Transaction

The following scenarios illustrate two ways to use an explicit transaction and achieve the same result. In these scenarios, a business service processes a sales order. Inventory records are updated when each record is processed instead of waiting until the end of the sales order processing to update the inventory records. In each scenario, if an error occurs before the sales order process is completed and committed, an exception message is sent to the caller, and updates that were made to the inventory records are rolled back.

9.3.2.1 Scenario 1

This scenario uses an explicit auto commit transaction that updates the Inventory table and commits and releases the table immediately before continuing with the remainder of the sales order processing. Because inventory records are committed before the sales order is committed, an error could occur during the continued processing of the sales order. If an error occurs, another business function (referred to as a compensating business function) must be called to undo the inventory updates.

You use another explicit transaction to call the compensating business function. You can either reuse the original auto commit connection or create a new connection. The best option is to reuse the original auto commit connection, because this limits the number of objects that are created. You cannot use the default transaction because you want to send an exception message to the caller indicating that the sales order processing failed, and you want to roll back any updates that were made to the

inventory records. You use an explicit connection so that you can control the compensating business function to ensure that updates are rolled back, even if an exception is thrown.

This code sample illustrates this scenario:

```
public E1MessageList processSalesOrder(IContext context, IConnection
connection, InternalProcessSalesOrder internalVO){
 ...
     //Create new explicit auto commit connection to add inventory
     //records
     IConnection invConnection = context.getNewConnection
(IConnection.AUTO);


        //call method (created by the wizard), which then executes
Business Function or Database operation
           E1MessageList invMessages =  callInventoryMBF(context,
                                                invConnection,
                                                internalVO,
                                                programId);
           //add messages returned from E1 processing to business
           //service message list.
           messages.addMessages(invMessages);
           if (!invMessages.hasErrors()) {
              //No errors continue processing SO
           IConnection soConnection = context.getNewConnection
(IConnection.MANUAL);
           try {
                     //Call  SO
                     E1MessageList soMessages = callSOMBF(context,
                                                   soConnection,
                                                   internalVO);
                     //Check for errors, collect in messages.
                     if (!soMessages.hasErrors()) {
                        soConnection.commit();
                     }else{
                        soConnection.rollback();
              //Errors in SO processing, call MBF to compensate for
              //added inventory
                   E1MessageList compMessages = callInventory
CompensateMBF(context,invConnection,internalVO);
                        if(compMessages.hasErrors()){
                           compMessages.setMessagePrefix("Unable to
Compensate for Added Inventory");
                        }
                        messages.addMessages(compMessages);
                     }
                  }
                  catch (BSSVConnectionException e) {
                     //Create new error and return E1MessageList
                     E1Message txMessage =  new E1Message
(context, "006FIS",  e.getMessage());
                     messages.addMessage(txMessage);
                  }
                  soConnection.close();

             }

           invConnection.close();
```

```
    finishInternalMethod(context, "addAddressBook");
    return messages;
  }
```

9.3.2.2 Scenario 2

This scenario uses an auto commit connection to create a default transaction by calling startPublishedMethod and passing an additional parameter that specifies the auto commit connection—*startPublishedMethod(context,"processSalesOrder",IConnection.AUTO).* Because inventory records are committed before the sales order is committed, an error could occur during the continued processing of the sales order. If an error occurs, another business function (referred to as a compensating business function) must be called to undo the inventory updates.

To control the transaction and handle a sales order failure, you use a manual commit connection to call the Sales Order Commit business function. Everything within the business function call will roll back. You can call a compensating business function to roll back the inventory records that were automatically committed. You want the default auto commit transaction to call the compensating business function.

This code sample illustrates this scenario:

```
public E1MessageList processSalesOrder(IContext context, IConnection
connection, InternalProcessSalesOrder internalVO){
 ...

        //call method (created by the wizard), which then executes
Business Function or Database operation
           E1MessageList invMessages =  callInventoryMBF(context,
                                                         connection,
                                                         internalVO,
                                                         programId);
           //add messages returned from E1 processing to business
           //service message list.
           messages.addMessages(invMessages);
           if (!invMessages.hasErrors()) {
              //No errors continue processing SO using manual commit
              //connection
           IConnection soConnection = context.getNewConnection
(IConnection.MANUAL);
           try {
                     //Call  SO
                     E1MessageList soMessages = callSOMBF(context,
                                                soConnection,
                                                internalVO);
                     //Check for errors, collect in messages.
                     if (!soMessages.hasErrors()) {
                        soConnection.commit();
                     }else{
                        soConnection.rollback();
              //Errors in SO processing, call MBF to compensate for
              //added inventory
              E1MessageList compMessages = callInventoryCompensateMBF
(context,connection,internalVO);
                        if(compMessages.hasErrors()){
                           compMessages.setMessagePrefix
("Unable to Compensate for Added Inventory");
                        }
                        messages.addMessages(compMessages);
                     }
```

```
                    }
                    catch (BSSVConnectionException e) {
                       //Create new error and return E1MessageList
                       E1Message txMessage =  new E1Message
(context, "006FIS",  e.getMessage());
                       messages.addMessage(txMessage);
                    }
                    soConnection.close();

          }
      finishInternalMethod(context, "addAddressBook");
      return messages;
  }
```

10

Understanding Logging

This chapter contains the following topic:

- Section 10.1, "Logging"

10.1 Logging

You use log files to troubleshoot system behavior. The location of the business service and published business service log files is defined in the jdelog.properties file under <pathcode>/ini/bssv. The default location of these log files is <pathcode>/bssv/log, which you can change.

10.1.1 Default Logging

The business service foundation provides default logging behavior. When startInternalMethod(IContext context, String methodName, ValueObject internalVO) is called, the following information is automatically written in the log file:

```
22 Aug 2006 22:25:24,125 [Line ?] [DEBUG ]  - [BSSVFRAMEWORK]
[Context ID: 141.144.96.127:1907:1156307000656]    startInternalMethod()
executed for addAddressBook
22 Aug 2006 22:25:24,140 [Line ?] [DEBUG ]  - [BSSVFRAMEWORK]
[Context ID: 141.144.96.127:1907:1156307000656]    ValueObject for
addAddressBook:
============================================
ValueObject oracle.e1.bssv.J0100010.valueobject.InternalAddAddressBook:
InternalPhones[0]:
============================================
ValueObject oracle.e1.bssv.J0100030.valueobject.InternalPhone:
   SzPhoneNumber: 444-5555
   SzPhoneAreaCode: 303
   SzPhoneNumberType: HOM
============================================
InternalPhones[1]:
============================================
ValueObject oracle.e1.bssv.J0100030.valueobject.InternalPhone:
   SzPhoneNumber: 444-1555
   SzPhoneAreaCode: 303
   SzPhoneNumberType: 02
============================================
InternalPhones[2]:
============================================
ValueObject oracle.e1.bssv.J0100030.valueobject.InternalPhone:
   SzPhoneNumber: 444-1655
   SzPhoneAreaCode: 303
   SzPhoneNumberType: HOM
```

```
==========================================
   SzTaxId: 11655018
   SzCountry: US
   SzState: CO
   SzCounty: Arapahoe
   SzCity: Denver
   SzPostalCode: 80807
   SzAddressLine4: Line 4
   SzAddressLine3: Line 3
   SzAddressLine2: Line 2
   SzAddressLine1: 223 W. Teller Ave
   SzMailingName: Green Tracy18
   MnAddressBookNumber: 0
   SzLongAddressNumber: 165346418
   JdDateEffective: Mon Sep 04 22:23:20 MDT 2006
   SzBusinessUnit:           30
   SzVersion: XJDE0001
   SzSearchType: E
   SzAlphaName: Tracy, Green18
```

10.1.2 Explicit Logging

You can use this code to provide explicit logging in the business service:

```
      //RI: call to logger - log the beginning of the business service method
processing using app ID
       context.getBSSVLogger().app(context,
                  "@@@@@@Begin call for BSSV AddressBookProcessor.
addAddressBook",
                  "\n",
                  internalVO.toString(),
                  null)
       ...
   }
```

Many logging methods exist for signifying APP, DEBUG, WARN, or SEVERE conditions. Plain text as well as exceptions can be passed as parameters to these methods for inclusion in the logs.

This information is logged into the jas.log file as a result of the preceding sample code:

```
17 Jul 2006 16:53:51,125 [Line ?] [APP   ]  - [oracle.e1.foundation.util.
IBSSVLogger]
       [Context ID: 10.139.87.66:2751:1153176823468]
       @@@@@@Begin call for BSSV AddressBookProcessor.addAddressBook
```

11

Understanding JD Edwards EnterpriseOne as a Web Service Consumer

This chapter contains the following topics:

- Section 11.1, "JD Edwards EnterpriseOne as a Web Service Consumer"
- Section 11.2, "C Business Function Calling a Business Service"
- Section 11.3, "Creating a Business Service for JD Edwards EnterpriseOne as a Web Service Consumer"
- Section 11.4, "Using Softcoding"
- Section 11.5, "Testing the Business Service for JD Edwards EnterpriseOne as a Web Service Consumer"

> **Important:** Oracle reserves the right to reorganize the business services foundation packages (jar files) for tools release upgrades. If you are planning to upgrade your system, test your custom objects and modify them as appropriate to ensure your code will continue to work as intended. You cannot upgrade custom business service objects after you install a tools release upgrade.

11.1 JD Edwards EnterpriseOne as a Web Service Consumer

JD Edwards EnterpriseOne can call and process external web services. Being a native consumer of web service enables JD Edwards EnterpriseOne integration with other Oracle products and third-party systems. To enable JD Edwards EnterpriseOne integration with other systems, you create a business function that calls a business service. The business service calls an external web service. You also create a web service proxy that identifies where the web service can be found. The web service proxy contains any security information that must be passed in the web service call. Some web services do not require security. The results of the call are returned to the business service. The business service passes the results to the business function. This diagram illustrates JD Edwards EnterpriseOne as a web service consumer.

Figure 11–1 Process flow for JD Edwards EnterpriseOne as a web service consumer.

11.2 C Business Function Calling a Business Service

The C business function builds an XML document that contains required input and output parameters, and passes the XML document to an API that calls the business service. The XML document is based on the business service value object. Similarly, the return from the API includes an XML document with the results of the call.

11.2.1 Best Practices for Business Functions Calling Business Services

When a need for calling a web service from within JD Edwards EnterpriseOne occurs, a business function is required to make that call. To preserve changes that you have made to the JD Edwards EnterpriseOne business function when you upgrade or update your system, Oracle recommends that you create a new business function specifically for this task. This web service consumer business function can be called by a JD Edwards EnterpriseOne application or business function. Processing in this web service business function would include:

- Initialize XML.
- Build XML.
- Call the API that calls the business service.
- Map the response.
- Handle errors.
- Return to the calling business function.

11.3 Creating a Business Service for JD Edwards EnterpriseOne as a Web Service Consumer

To use JD Edwards EnterpriseOne as a web service consumer, you create a business service and its value object using methodology and tools discussed in preceding chapters of this guide and in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

You can use the XML Template utility to create an empty XML document that is based on a business service value object. The XML Template utility is provided by JDeveloper and supports these data types:

- java.lang.Integer
- java.math.BigDecimal

- oracle.e1.bssvfoudnation
- util.MathNumeric
- java.util.GregorianCalendar
- java.util.Date
- java.lang.Short
- java.lang.Boolean
- java.lang.String

11.3.1 Naming Convention for Consumer Business Services

For a business service that consumes third-party web services, the OMW object name is JC, system code, and zeros, where the zeros are a number that you assign; for example JC850001.

Note: JCXXXXXX is used to distinguish between JD Edwards EnterpriseOne published business services (which are JPXXXXXX), internal business services (J0100003), and consumer business services. Some early consumer business services are named as J, system code, and XXXXX (for example, J8500001). These existing consumer business services will not be changed, only new consumer business services will include the JC preface.

11.3.2 Rules for Value Object for JD Edwards EnterpriseOne as a Web Service Consumer

A business service that is called from a business function must represent collections as arrays. You cannot use the ArrayList data type because it cannot be serialized. This code sample shows using an array for declaring the compound for phones:

```
private InternalPhone[] internalPhones = null;
```

11.4 Using Softcoding

Softcoding is a way to dynamically provide the where and who information to the web service proxy. The web service proxy needs to know exactly which machine to call for the service (the where), and it needs to know the credentials to pass for the call (the who). Also, values you use to test your business server in the development environment probably will be different from the actual values that are used in the production environment. Softcoding allows the where and who values to be plugged in at runtime instead of hard-coding these values into the business service.

A web service proxy has at least one softcoding template and one softcoding record; but a web service proxy can have many templates and many records. You can use softcoding templates to create softcoding records. Using a softcoding template is productive because softcoding records have similar values. Using a template also helps to minimize typing errors when you are entering record information.

11.4.1 Softcoding Template Naming Conventions

JD Edwards EnterpriseOne softcoding templates are named like this:

- E1_J34A0010

- E1_J34A0010A

E1 indicates that the template was created by JD Edwards EnterpriseOne developers at Oracle. J34A000, which is the key, is the business service name. The A indicates that a second template exists for the same business service.

To keep updates and upgrades simple, Oracle recommends that you not modify a JD Edwards EnterpriseOne softcoding template. Instead, you should copy the JD Edwards EnterpriseOne template, provide a new name, and make the appropriate modifications. For example, if you need to add security information to a template that has the correct right endpoint information, you can copy the existing template, rename it, and add the security information. You might name the new template similar to the JD Edwards EnterpriseOne template, for example:

CUST_J34A000

See Also:

- "Understanding Softcoding" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

11.5 Testing the Business Service for JD Edwards EnterpriseOne as a Web Service Consumer

You test the business service in the development environment. You can test a business service that calls an external web service using one of these methods:

- Create a test business service.
- Use the development business services server.

Guidelines for using these methods are provided in Appendix B of the *JD Edwards EnterpriseOne Tools Business Services Development Guide.*

See Also:

- "Creating a Test Business Service" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

12

Using Business Services with HTTP Request/Reply

This chapter contains the following topics:

- Section 12.1, "Understanding Business Services and HTTP POST"
- Section 12.2, "Using Business Services with HTTP Request/Reply"
- Section 12.3, "Testing the Servlet"

12.1 Understanding Business Services and HTTP POST

JD Edwards EnterpriseOne enables you to use a business service to communicate with a third-party system using HTTP POST. In this scenario, a business function is invoked by a request from a JD Edwards EnterpriseOne HTML web client, which in turn calls a business service to make an HTTP POST request. You provide callback information in the posted data for the third-party system to send an asynchronous reply to the request. When the callback reply is received from the third-party system, the published business service that was included in the callback information is invoked. The response is returned to the JD Edwards EnterpriseOne HTML web client.

The business services server uses a servlet listener to receive incoming messages from third-party systems. Received messages contain callback information, which is used to associate the message with the correct request.

See Also:

- "Understanding Business Services and HTTP POST" in the *JD Edwards EnterpriseOne Tools Business Services Development Guide*.

12.2 Using Business Services with HTTP Request/Reply

When you use business services to do an HTTP request/reply, follow these rules:

- The listener servlet checks for authorization before calling the published business service. Therefore, you must have authorization to invoke the specified method on the published business service.
- The value object class of the method to be called must have only one string field and the accessor (getter/setter) method for the string field. The received XML payload will be passed to the method in this string field.
- The method to be called must have three parameters. This code sample shows the signature for this method:

```
public responseVO methodToBeCalled((IContext context, IConnection
```

```
connection,requestVO vo)
```

Note: This method must have a public modifier. The wizard that you use to create the structure for a published business service generates a method with a protected modifier. You must change the method from protected to public so that the published business service can be called from the listener service.

- The listener servlet does not wait for a response from the business service call. Any response is ignored.
- This kind of published business service must be used as the bridge between getting a response from external sites and calling the processor business service that does the business logic.

12.3 Testing the Servlet

You should test the servlet to ensure that it receives the return messages. You can do this by creating an XML document that has the HTTP URL in it and ensuring that the message is delivered to the URL.

A

Utility Business Services

This appendix contains the following topics:

A.1 Understanding Utility Business Services

Utility business services are generic, reusable services that perform standard operations. Utility business services are called by other business services to process information that is associated with the calling service. Utility business services eliminate the need to write the same code in a number of business services, and they ensure that a specific process is performed in a uniform manner.

The utility business services follow the rules, best practices, and guidelines discussed in this methodology guide. Utility business service processing should be transparent to consumers of the published business service that calls them. General information about each utility business service is provided in this appendix. If you create custom published business services, you can use these predefined utilities, or you can copy a predefined utility business service into a new business service object, modify it, and call it from your new business service.

A.1.1 Implementing Utility Business Services

General information for creating utility business services is provided in this guide. Here are some key items about utility business services:

- Utility business services are called from more than one business service or published business service.
- All data mappings are made inside of the utility, not by the service calling the utility.
- Any errors that are encountered by the utility during processing are returned to the calling service to handle.

A.2 Entity Processor Business Service

This section discusses the Entity Processor business service.

A.2.1 Understanding the Entity Processor Business Service

The Entity Processor business service (J0100010) provides a published interface that exposes three ways to provide address book key information for an entity.

The Entity Processor business service retrieves entity ID, entity long ID, and entity tax ID based on input that is supplied by the published business service that calls the utility. This utility business service processes data in these ways:

- Retrieves Entity ID and Entity Long ID when Entity Tax ID is supplied as input.
- Retrieves Entity ID and Entity Tax ID when Entity Long ID is supplied as input.
- Retrieves Entity Tax ID and Entity Long ID when Entity ID is supplied as input.

A.2.2 Implementation Detail

This topic identifies the methods, signature, and value object (VO) classes for the Entity Processor business service.

A.2.2.1 Methods

Methods for the business service are:

- processEntity(Entity)
- processEntity(InternalEntityUtility)

A.2.2.2 Signature

The signature for the business service is:

```
Public static E1MessageList processEntity(IContext context, IConnection,
connection, ValueObject inputObject, ValueObject currentObject)
```

A.2.2.3 Value Object Classes

Value object classes for the business service are:

- Entity
- InternalEntityUtility

The Entity value object class is a published value object that is owned and managed by the Entity Processor business service. Any published business service that wants to use the Entity value object class within its interface must import the class.

A.2.2.4 Functional Processing

A published business service calls processEntity and passes an input value to the method. The processEntity method sets processing parameters based on the input value. The method compares the input with null or an empty string to determine which values are not included in the input. The order of null comparison is:

1. Address Number
2. Long Address Number
3. Tax ID

If the comparison is successful, the processEntity method calls the internal method, InternalEntityUtility. The InternalEntityUtility method calls the ScrubAddressNumber business function (B0100016) passing in the desired action code. The business function retrieves the appropriate data from the Address Book Master table (F0101).

> **Note:** This method retrieves records from F0101 and ensures that the ScrubAddressNumber business function selects the appropriate data. Using the business function instead of using direct table input/output has no significant performance impact.

A.2.3 Value Object Classes

The tables in this section provide data information for value object classes.

A.2.3.1 Business Service Value Object

InternalEntityUtility	N/A	N/A
Business Service VO Field Name	**Data Type**	**Input/Output**
mnAddressNumber	MathNumeric	I/O
szLongAddressNumber	String	I/O
szTaxId	String	I/O

A.2.3.2 Published Reusable Value Object

Entity	N/A	N/A	N/A	N/A
Published Business Service VO Field Name	**Data Type**	**Input**	**Key**	**Javadoc**
entityId	Integer	Yes	Yes	Address book number
entityLongId	String	Yes	No	NA
entityTaxId	String	Yes	No	NA

A.2.3.3 Output from Business Service to Published Value Object

InternalEntityUtility	Entity	N/A	N/A	N/A
Business Service VO Field Name	**Data Type**	**Published Business Service VO Field Name**	**Data Type**	**Transformer**
mnAddressNumber	MathNumeric	entityId	Integer	MathNumeric to Integer
szLongAddressNumber	String	entityLongId	String	Map
szTaxId	String	entityTaxId	String	Map

A.3 GL Account Processor Business Service

This section discusses the GL Account Processor business service.

A.3.1 Understanding the GL Account Processor Business Service

The GL Account Processor business service (J0900010) provides a published interface that exposes four ways to provide general ledger account information.

The GL Account Processor business service retrieves account information based on input that is supplied by the published business service that calls the utility. This utility business service processes data in these ways:

- Retrieves GL Account Long ID, GL Account Alternate data, and account information from objectAccount, businessUnit, and subsidiary fields when GL Account ID is supplied as the input field.
- Retrieves GL Account ID, GL Account Alternate data, and account information from objectAccount, businessUnit, and subsidiary fields when Account Long ID is supplied as the input field.
- Retrieves GL Account ID, GL Account Long ID data, and account information from objectAccount, businessUnit, and subsidiary fields when Account Alternate is supplied as the input field.
- Retrieves GL Account ID, GL Account Long ID, and GL Account Alternate data when account information fields (objectAccount, businessUnit and subsidiary) are supplied as the input field.

A.3.2 Implementation Detail

This topic identifies the methods, signature, and value object classes for the GL Account Processor business service.

A.3.2.1 Methods

Methods for the business service are:

- processGLAccount(InternalGLAccountUtility)
- processGLAccount(ProcessGLAccount)

A.3.2.2 Signature

The signature for the business service is:

```
Public static E1MessageList processGLAccount(IContext context, IConnection
connection, ValueObject inputObject, ValueObject currentObject)
```

A.3.2.3 Value Object Class

Value object classes for the business service are:

- InternalGLAccountUtility
- ProcessGLAccount
  - GLAccount
  - GLAccountKey

The GLAccount and GLAccountKey classes are published value objects that are owned and managed by the GL Account Processor business service. Any published

business service that wants to use the GLAccount or GLAccountKey classes within its interface must import these classes.

A.3.2.4 Functional Processing

A published business service calls processGLAccountUtility and passes an input value to the method. The processGLAccountUtility method sets processing parameters based on the input value. The method compares the input with null to determine which values are not included in the input. The order of null comparison is:

1. Account ID
2. Account Long ID
3. Account Alternate

If all the values are null for these account fields, then the method evaluates these fields:

- objectAccount
- businessUnit
- subsidiary

If the comparison is successful, the processGLAccountUtility method calls the internal method, InternalGLAccountUtility. The InternalGLAccountUtility method calls the ValidateAccountNumber business function (XX0901), passing in the desired action code. The business function retrieves the appropriate data from the Account Master table (F0901).

> **Note:** This method retrieves records from F0901. The ValidateAccountNumber business function selects 19 columns from the table. Using the business function does not have a significant performance impact.

A.3.3 Value Object Classes

The tables in this section provide data information for value object classes.

A.3.3.1 Business Service Input and Output Interface

InternalGLAccountUtility	**N/A**	**N/A**
Business Service VO Field Name	**Data Type**	**Input/Output**
szAccountNumber	String	Input/Output
szAccountId	String	Input/Output
szUnstructuredAccount	String	Input/Output
szDatabaseBusinessUnit	String	Input/Output
szDatabaseObject	String	Input/Output
szDatabaseSubsidiary	String	Input/Output

> **Note:** For the account to be located, business unit, object, and subsidiary must be passed.

A.3.3.2 Published Reusable Value Object

ProcessGLAccount	N/A	N/A	N/A
Published Business Service VO Field Name	**Data Type**	**Input**	**Key**
GLAccount	N/A	N/A	N/A
objectAccount	String	Yes	No
businessUnit	String	Yes	No
subsidiary	String	Yes	No
GLAccountKey	N/A	N/A	N/A
accountId	String	Yes	Yes
accountLongId	String	Yes	No
accountAlternate	String	Yes	No

A.3.3.3 Published to Business Service Value Object

ProcessGLAccount	InternalGL AccountUtility	N/A	N/A	N/A
Published VO Field Name	**Data Type**	**Business Service VO Field Name**	**Data Type**	**Transformer/ Formatter**
GL Account	N/A	N/A	N/A	N/A
objectAccount	String	szDatabaseObject	String	Map
businessUnit	String	szDatabaseBusinessUnit	String	Map
subsidiary	String	szDatabaseSubsidiary	String	Map
GLAccountKey	N/A	N/A	N/A	N/A
accountId	String	szAccountId	String	Map
accountLongId	String	szAccountNumber	String	Map
accountAlternate	String	szUnstructuredAccount	String	Map

A.4 Inventory Item ID Processor Business Service

This section discusses the Inventory Item ID Processor business service.

A.4.1 Understanding the Inventory Item ID Processor Business Service

The Inventory Item ID Processor business service (J4100010) provides a published interface that exposes five ways to provide item identification information.

The Inventory Item ID Processor business service retrieves all potential identifiers for an inventory item based on input that is supplied by the published business service that calls the utility. This utility business service processes data in these ways:

- Retrieves itemProduct and itemCatalog when itemId is supplied as the input field.
- Retrieves itemId and itemCatalog when itemProduct is supplied as the input field.
- Retrieves itemId and itemProduct when itemCatalog is supplied as the input field.

- Retrieves itemId, itemProduct, and itemCatalog when itemCustomer or itemSupplier and entity ID and cross-reference type code are supplied as input fields.
- Retrieves itemId, itemProduct, and itemCatalog when itemFreeForm, branch plant, cross-reference type code, and entityId are supplied as input fields.

A.4.2 Implementation Detail

This topic identifies the methods, signature, and value object classes for the Inventory Item ID Processor business service.

A.4.2.1 Methods

Methods for the business service are:

- processInventoryItemId (InternalInventoryItemId)
- processInventoryItemId (ProcessItemCustomer)
- processInventoryItemId (ProcessItemSupplier)

A.4.2.2 Signature

The signature for the business service is:

```
Public static E1MessageList processInventoryItemID(IContext context,
IConnection connection, ValueObject inputObject, ValueObject currentObject)
```

A.4.2.3 Value Object Classes

Value object classes for this business service are:

- InternalInventoryItemId
- ProcessItemCustomer

  – ItemGroupCustomer
- ProcessItemSupplier

  – ItemGroupSupplier

The ItemGroupCustomer and ItemGroupSupplier classes are published value objects that are owned and managed by the Inventory Item ID Processor business service. Any other business service that wants to use the ItemGroupCustomer and ItemGroupSupplier classes as part of its interface must import these classes.

A.4.2.4 Functional Processing

The Inventory Item ID Processor determines processing based on whether a supplier item or a customer item class was passed by the published business service. The utility retrieves related cross-reference data for the supplier or customer item, if required. The ProcessItemCustomer or ProcessItemSupplier method compares the input value with null or an empty string to determine processing. The first match that the utility finds determines how the utility retrieves the data. The order of null comparison is:

1. ItemCrossReference
2. FreeForm
3. ItemId
4. ItemProduct

5. ItemCatalog

Depending on which field, if any, is selected during the comparison process, the ProcessItemCustomer or ProcessItemSupplier method calls the internal method, InternalInventoryItemID, and makes a call to the appropriate business function, passing the expected parameters. Finally, all retrieved item numbers (mnShortItemNumber, sz2ndItemNumber, sz3rdItemNumber) are populated at the end of the process.

These business functions are used with this utility business service:

- Validate and Retrieve Item Master (X4101)
- Get Item Master Description UOM (B4001040)
- Verify and Get Item Xref (B4100600)
- Verify and Get Branch Plant Constants (B4101390)

Depending on the business function that is used, data is retrieved from these tables:

- F4101 (Item Master)
- F4104 (Item Cross Reference)
- F41001 (Inventory Constants)

> **Note:** Database I/O operations are performed through business functions in the JD Edwards EnterpriseOne Validate and Retrieve Item Master module (X4101). This module performs efficient fetches from F4101, retrieving only the columns needed for each type of fetch.
>
> To prevent recalling the VerifyandGetBranchPlantConstants function, any cross-reference code that is fetched will be passed back so that users can pass it in instead of having the utility pass the cross-reference code.

A.4.3 Value Object Classes

The tables in this section provide data information for value object classes.

A.4.3.1 Business Service Value Object

InternalInventoryItemId	**N/A**	**N/A**
Business Service VO Field Name	**Data Type**	**Input/Output**
mnShortItemNumber	MathNumeric	Input and Output
sz2ndItemNumber	String	Input and Output
sz3rdItemNumber	String	Input and Output
szFreeFormItemNumber	String	Input
szBranchPlant	String	Input
szCrossRefItemNumber	String	Input
mnAddressNumber	MathNumeric	Input
szCrossRefTypeCode	String	Input

A.4.3.2 Published Reusable Value Object

ProcessItem	**N/A**	**N/A**	**N/A**
Published Business Service VO Field Name	**Data Type**	**Input**	**Key**
crossReferenceType	String	Yes	No
entityId	Integer	Yes	No
branchPlant	String	Yes	No
ItemGroupCustomer	N/A	N/A	N/A
itemId	Integer	Yes	No
itemProduct	String	Yes	No
itemCatalog	String	Yes	No
itemFreeForm	String	Yes	No
itemCustomer	String	Yes	No
ItemGroupSupplier	N/A	N/A	N/A
itemId	Integer	Yes	No
itemProduct	String	Yes	No
itemCatalog	String	Yes	No
itemFreeForm	String	Yes	No
itemSupplier	String	Yes	No

A.4.3.3 Input Business Service Processing

From VO/BSFN/Business Service Property/Other	**To BSFN/Other**	**N/A**	**N/A**	**N/A**
Field Name	**Data Type**	**Field Name**	**Data Type**	**Transformer**
Based on Input VO	N/A	**VerifyAndGetBranchPlant Constants (B41001390 / D41001390A)**	N/A	N/A
szBranchPlant	String	szBranchPlant	String	Map
ItemIdsByXref-MODE	N/A	N/A	N/A	N/A
N/A	N/A	**VerifyAndGetItemXref (B4100600 / D4100600)**	N/A	N/A
Business Service Property NUMBER_OF_KEYS = 3	String	szKeys	String	Map
Business Service Property INDEX_ID = 4	String	szIndex	String	Map
szCrossRefItemNumber	String	szCustomerItemNumber	String	Map
mnAddressNumber	MathNumeric	mnAddressNumber	MathNumeric	Map
szCrossRefTypeCode	String	szCrossRefTypeCode	String	Map
N/A	N/A	**getItemIdsByItemId (internal function)**	N/A	N/A

From VO/BSFN/Business Service Property/Other	To BSFN/Other	N/A	N/A	N/A
mnShortItemNumber	MathNumeric	mnShortItemNumber	MathNumeric	Map
ItemIdsByItemFreeform – MODE	N/A	N/A	N/A	N/A
N/A	N/A	**GetItemMasterDescription UOM (B4001040 / D4001040)**	N/A	N/A
szFreeFormItemNumber	String	szPrimaryItemNumber	String	Map
szBranchPlant	String	szBranchPlant	String	Map
ItemIdsByItemId – MODE	N/A	N/A	N/A	N/A
N/A	N/A	**GetItemMasterByShortItem (X4101 / DSDX4101B)**	N/A	N/A
mnShortItemNumber	MathNumeric	mnShortItemNumber	MathNumeric	Map
ItemIdsbyItemFreeForm – MODE	N/A	N/A	N/A	N/A
N/A	N/A	**GetItemMasterBy2ndItem (X4101 / DSDX4101C)**	N/A	N/A
sz2ndItemNumber	String	sz2ndItemNumber	String	Map
ItemIdsByItemCatalog – Mode	N/A	N/A	N/A	N/A
N/A	N/A	**GetItemMasterBy3rdItem (X4101 / DSDX4101D)**	N/A	N/A
sz3rdItemNumber	String	sz3rdItemNumber	String	Map

A.5 Net Change Processor Business Service

This section discusses the Net Change Processor business service.

A.5.1 Understanding the Net Change Processor Business Service

The Net Change Processor business service (J0000020) handles net change processing for both fields and value objects. The utility processes changes depending on which method is called:

- Net Change by Field

  The Net Change Processor utility determines the value of a field to use to update an entity. If you do not specify a new value for a field, the utility preserves the current value.

- Net Change by Value Object

  The Net Change Processor utility determines the value of all of the fields within a value object to use to update an entity. If you do not specify a new value for a field within a value object, the utility preserves the current value of the field within the value object.

Blank and **zero** are valid values for fields in the input object, and the utility preserves these values. If a field in the input object has a null value, the utility replaces the null value with the current database value.

A.5.2 Implementation Detail

This topic identifies the methods, signature, and value object classes for the Net Change Processor business service. Each method is discussed separately.

A.5.2.1 Method

The method that handles net change processing for value objects of this business service is performNetChange.

A.5.2.2 Signature

The signature for the business service is:

```
public static E1MessageList performNetChange(IContext context,
IConnection connection, ValueObject inputObject, ValueObject currentObject)
```

A.5.2.3 Value Objects

Value object classes for the business service are:

- ValueObject inputObject

  This value object holds the values received from a business service or published business service.

- ValueObject currentObject

  This value object holds the values of an entity as they exist in the database.

A.5.2.4 Functional Processing

When you update an entity in the database, the performNetChange method determines the value of all fields within the value object that you are using. If no new value for a field within a value object is specified, the method preserves the current database value of the field. This method allows processing of value objects of different types. The performNetChange method assumes that the value object is flat. Field values of **blank** and **zero** are valid values in the input object, and the method preserves them. Only fields with a value of null in the input object are replaced with the current database value.

A.5.2.5 Method

The method that handles net change processing for fields of this business service is performNetChangeOnFields.

A.5.2.6 Signature

The signature for the business service is:

```
public static Object performNetChangeOnFields(IContext context,
IConnection connection, Object inputFieldValue, Object currentFieldValue)
```

A.5.2.7 Value Objects

This utility business service has no specific value objects.

A.5.2.8 Functional Processing

The performNetChangeOnFields method determines the value of a field to use when you are updating an entity. If no new value for a field is specified in the input field, the

method returns the current value of the field. Field values of blank and zero are valid values in the input object, and the method preserves them. Only a value of null in the input object is replaced with the current database value.

Note: The value object net change methods operate on a flat value object class only. Processing over compound value objects is complex and negatively affects performance.

The net change processor exposes the performNetChangeOnFields method to expose a less complex implementation of net change processing for use in those instances in which processing the full value object is undesirable.

A.5.3 Value Object Classes

This utility handles all objects that extend the value object super class. Because the utility is written to handle generic objects, the utility does not have any specific value object mappings.

A.6 Processing Version Processor Business Service

This section discusses the Processing Version Processor business service.

A.6.1 Understanding the Processing Version Processor Business Service

The Processing Version Processor business service (J0000010) determines the processing option version that a business service uses when it calls a business function. The consumer of a published business service is responsible for providing the service constant key to the Processing Version Processor utility. If no version is specified in the published business service, the Processing Version Processor utility retrieves a processing option version from service constants.

A.6.2 Implementation Detail

This topic identifies the methods, signature, and value object classes for the Net Change Processor business service. Each method is discussed separately.

A.6.2.1 Method

The method for the business service is getProcessingVersion.

A.6.2.2 Signature

The signature for the business service is:

```
public static E1MessageList getProcessingVersion(IContext context,
IConnection connection, InternalProcessingVersion processingVersionVO))
```

A.6.2.3 Value Object

The value object for the business service is InternalProcessingVersion.

A.6.2.4 Functional Processing

A business service calls the getProcessingVersion method. This method verifies that the required input parameters are specified. If all required parameters are passed, the

method checks the processingOptionVersionValue parameter to determine whether it contains a value. If no value exists, the method looks up the default value in service constants using a consumer-provided key. The default value must be set up in service constants. The utility does not validate the value that it retrieves from the service constants systems. If no errors are encountered, the correct processing option version value is returned to the published business service consumer.

Note: The Processing Version Processor utility retrieves a value from the service constants system only when a processing option version value is not provided by the consumer of the published business service.

A.6.3 Value Object Classes

The tables in this section provide data information for value object classes.

A.6.3.1 Business Service Value Object

InternalProcessAddressBook	**N/A**	**N/A**	**N/A**
Business Service VO Field Name	**Data Type**	**Input / Output**	**Comments**
processingOptionVersionValue	String	Input/Output	On input, this field contains the processing option version value provided by the consumer.
defaultValueServiceConstantKey	String	Input	This field contains the service constant key for the default processing option version to use if no processing option version is provided by the consumer.

Glossary

Accessor Methods/Assessors

Java methods to "get" and "set" the elements of a value object or other source file.

business service

EnterpriseOne business logic written in Java. A business service is a collection of one or more artifacts. Unless specified otherwise, a business service implies both a published business service and business service.

business service framework

Parts of the business service foundation that are specifically for supporting business service development.

business service property

Key value data pairs used to control the behavior or functionality of business services.

Business Service Property Admin Tool

An EnterpriseOne application for developers and administrators to manage business service property records.

business service property business service group

A classification for business service property at the business service level. This is generally a business service name. A business service level contains one or more business service property groups. Each business service property group may contain zero or more business service property records.

business service property key

A unique name that identifies the business service property globally in the system.

business service property utilities

A utility API used in business service development to access EnterpriseOne business service property data.

business service property value

A value for a business service property.

business services server

The physical machine where the business services are located. Business services are run on an application server instance.

business services source file or business service class

One type of business service artifact. A text file with the .java file type written to be compiled by a Java compiler.

business service value object template

The structural representation of a business service value object used in a C-business function.

Business Service Value Object Template Utility

A utility used to create a business service value object template from a business service value object.

business services server artifact

The object to be deployed to the business services server.

exposed method or value object

Published business service source files or parts of published business service source files that are part of the published interface. These are part of the contract with the customer.

internal method or value object

Business service source files or parts of business service source files that are not part of the published interface. These could be private or protected methods. These could be value objects not used in published methods.

JDeveloper Project

An artifact that JDeveloper uses to categorize and compile source files.

JDeveloper Workspace

An artifact that JDeveloper uses to organize project files. It contains one or more project files.

published business service

EnterpriseOne service level logic and interface. A classification of a published business service indicating the intention to be exposed to external (non-EnterpriseOne) systems.

softcoding

A coding technique that enables an administrator to manipulate site-specific variables that affect the execution of a given process.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W

X