ORACLE
JD EDWARDS ENTERPRISEONE

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Methodology Guide

September 2008

ORACLE

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Methodology Guide
SKU E1_TOOLS898TME-B 0908

Open Source Disclosure

Contents

About This Documentation Preface

JD Edwards EnterpriseOne implementation guides provide you with the information that you need to implement and use JD Edwards EnterpriseOne applications from Oracle.

This preface discusses:

- JD Edwards EnterpriseOne application prerequisites.
- Application fundamentals.
- Documentation updates and downloading documentation.
- Additional resources.
- Typographical conventions and visual cues.
- Comments and suggestions.
- Common fields in implementation guides.

Note. Implementation guides document only elements, such as fields and check boxes, that require additional explanation. If an element is not documented with the process or task in which it is used, then either it requires no additional explanation or it is documented with common fields for the section, chapter, implementation guide, or product line. Fields that are common to all JD Edwards EnterpriseOne applications are defined in this preface.

JD Edwards EnterpriseOne Application Prerequisites

To benefit fully from the information that is covered in these books, you should have a basic understanding of how to use JD Edwards EnterpriseOne applications.

You might also want to complete at least one introductory training course, if applicable.

You should be familiar with navigating the system and adding, updating, and deleting information by using JD Edwards EnterpriseOne menus, forms, or windows. You should also be comfortable using the World Wide Web and the Microsoft Windows or Windows NT graphical user interface.

These books do not review navigation and other basics. They present the information that you need to use the system and implement your JD Edwards EnterpriseOne applications most effectively.

Application Fundamentals

Each application implementation guide provides implementation and processing information for your JD Edwards EnterpriseOne applications.

For some applications, additional, essential information describing the setup and design of your system appears in a companion volume of documentation called the application fundamentals implementation guide. Most product lines have a version of the application fundamentals implementation guide. The preface of each implementation guide identifies the application fundamentals implementation guides that are associated with that implementation guide.

The application fundamentals implementation guide consists of important topics that apply to many or all JD Edwards EnterpriseOne applications. Whether you are implementing a single application, some combination of applications within the product line, or the entire product line, you should be familiar with the contents of the appropriate application fundamentals implementation guides. They provide the starting points for fundamental implementation tasks.

Documentation Updates and Downloading Documentation

This section discusses how to:

- Obtain documentation updates.
- Download documentation.

Obtaining Documentation Updates

You can find updates and additional documentation for this release, as well as previous releases, on Oracle's PeopleSoft Customer Connection website. Through the Documentation section of Oracle's PeopleSoft Customer Connection, you can download files to add to your Implementation Guides Library. You'll find a variety of useful and timely materials, including updates to the full line of JD Edwards EnterpriseOne documentation that is delivered on your implementation guides CD-ROM.

Important! Before you upgrade, you must check Oracle's PeopleSoft Customer Connection for updates to the upgrade instructions. Oracle continually posts updates as the upgrade process is refined.

See Also

Oracle's PeopleSoft Customer Connection, http://www.oracle.com/support/support_peoplesoft.html

Downloading Documentation

In addition to the complete line of documentation that is delivered on your implementation guide CD-ROM, Oracle makes JD Edwards EnterpriseOne documentation available to you via Oracle's website. You can download PDF versions of JD Edwards EnterpriseOne documentation online via the Oracle Technology Network. Oracle makes these PDF files available online for each major release shortly after the software is shipped.

See Oracle Technology Network, http://www.oracle.com/technology/documentation/psftent.html

Additional Resources

The following resources are located on Oracle's PeopleSoft Customer Connection website:

Resource	Navigation
Application maintenance information	Updates + Fixes
Business process diagrams	Support, Documentation, Business Process Maps

Resource	Navigation
Interactive Services Repository	Support, Documentation, Interactive Services Repository
Hardware and software requirements	Implement, Optimize + Upgrade; Implementation Guide; Implementation Documentation and Software; Hardware and Software Requirements
Installation guides	Implement, Optimize + Upgrade; Implementation Guide; Implementation Documentation and Software; Installation Guides and Notes
Integration information	Implement, Optimize + Upgrade; Implementation Guide; Implementation Documentation and Software; Pre-Built Integrations for PeopleSoft Enterprise and JD Edwards EnterpriseOne Applications
Minimum technical requirements (MTRs)	Implement, Optimize + Upgrade; Implementation Guide; Supported Platforms
Documentation updates	Support, Documentation, Documentation Updates
Implementation guides support policy	Support, Support Policy
Prerelease notes	Support, Documentation, Documentation Updates, Category, Release Notes
Product release roadmap	Support, Roadmaps + Schedules
Release notes	Support, Documentation, Documentation Updates, Category, Release Notes
Release value proposition	Support, Documentation, Documentation Updates, Category, Release Value Proposition
Statement of direction	Support, Documentation, Documentation Updates, Category, Statement of Direction
Troubleshooting information	Support, Troubleshooting
Upgrade documentation	Support, Documentation, Upgrade Documentation and Scripts

Typographical Conventions and Visual Cues

This section discusses:

- Typographical conventions.
- Visual cues.
- Country, region, and industry identifiers.
- Currency codes.

Typographical Conventions

This table contains the typographical conventions that are used in implementation guides:

Typographical Convention or Visual Cue	Description
Bold	Indicates PeopleCode function names, business function names, event names, system function names, method names, language constructs, and PeopleCode reserved words that must be included literally in the function call.
Italics	Indicates field values, emphasis, and JD Edwards EnterpriseOne or other book-length publication titles. In PeopleCode syntax, italic items are placeholders for arguments that your program must supply. We also use italics when we refer to words as words or letters as letters, as in the following: Enter the letter *O*.
KEY+KEY	Indicates a key combination action. For example, a plus sign (+) between keys means that you must hold down the first key while you press the second key. For ALT+W, hold down the ALT key while you press the W key.
`Monospace font`	Indicates a PeopleCode program or other code example.
“ ” (quotation marks)	Indicate chapter titles in cross-references and words that are used differently from their intended meanings.
. . . (ellipses)	Indicate that the preceding item or series can be repeated any number of times in PeopleCode syntax.
{ } (curly braces)	Indicate a choice between two options in PeopleCode syntax. Options are separated by a pipe (|).
[ ] (square brackets)	Indicate optional items in PeopleCode syntax.
& (ampersand)	When placed before a parameter in PeopleCode syntax, an ampersand indicates that the parameter is an already instantiated object. Ampersands also precede all PeopleCode variables.

Visual Cues

Implementation guides contain the following visual cues.

Notes

Notes indicate information that you should pay particular attention to as you work with the JD Edwards EnterpriseOne system.

Note. Example of a note.

If the note is preceded by *Important!,* the note is crucial and includes information that concerns what you must do for the system to function properly.

Important! Example of an important note.

Warnings

Warnings indicate crucial configuration considerations. Pay close attention to warning messages.

Warning! Example of a warning.

Cross-References

Implementation guides provide cross-references either under the heading "See Also" or on a separate line preceded by the word *See.* Cross-references lead to other documentation that is pertinent to the immediately preceding documentation.

Country, Region, and Industry Identifiers

Information that applies only to a specific country, region, or industry is preceded by a standard identifier in parentheses. This identifier typically appears at the beginning of a section heading, but it may also appear at the beginning of a note or other text.

Example of a country-specific heading: "(FRA) Hiring an Employee"

Example of a region-specific heading: "(Latin America) Setting Up Depreciation"

Country Identifiers

Countries are identified with the International Organization for Standardization (ISO) country code.

Region Identifiers

Regions are identified by the region name. The following region identifiers may appear in implementation guides:

- Asia Pacific
- Europe
- Latin America
- North America

Industry Identifiers

Industries are identified by the industry name or by an abbreviation for that industry. The following industry identifiers may appear in implementation guides:

- USF (U.S. Federal)

- E&G (Education and Government)

Currency Codes

Monetary amounts are identified by the ISO currency code.

Comments and Suggestions

Your comments are important to us. We encourage you to tell us what you like, or what you would like to see changed about implementation guides and other Oracle reference and training materials. Please send your suggestions to your product line documentation manager at Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, U.S.A. Or email us at appsdoc@us.oracle.com.

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions.

Common Fields Used in Implementation Guides

Address Book Number	Enter a unique number that identifies the master record for the entity. An address book number can be the identifier for a customer, supplier, company, employee, applicant, participant, tenant, location, and so on. Depending on the application, the field on the form might refer to the address book number as the customer number, supplier number, or company number, employee or applicant ID, participant number, and so on.
As If Currency Code	Enter the three-character code to specify the currency that you want to use to view transaction amounts. This code enables you to view the transaction amounts as if they were entered in the specified currency rather than the foreign or domestic currency that was used when the transaction was originally entered.
Batch Number	Displays a number that identifies a group of transactions to be processed by the system. On entry forms, you can assign the batch number or the system can assign it through the Next Numbers program (P0002).
Batch Date	Enter the date in which a batch is created. If you leave this field blank, the system supplies the system date as the batch date.
Batch Status	Displays a code from user-defined code (UDC) table 98/IC that indicates the posting status of a batch. Values are: *Blank:* Batch is unposted and pending approval. *A:* The batch is approved for posting, has no errors and is in balance, but has not yet been posted. *D:* The batch posted successfully. *E:* The batch is in error. You must correct the batch before it can post.

	P: The system is in the process of posting the batch. The batch is unavailable until the posting process is complete. If errors occur during the post, the batch status changes to *E*. *U:* The batch is temporarily unavailable because someone is working with it, or the batch appears to be in use because a power failure occurred while the batch was open.
Branch/Plant	Enter a code that identifies a separate entity as a warehouse location, job, project, work center, branch, or plant in which distribution and manufacturing activities occur. In some systems, this is called a business unit.
Business Unit	Enter the alphanumeric code that identifies a separate entity within a business for which you want to track costs. In some systems, this is called a branch/plant.
Category Code	Enter the code that represents a specific category code. Category codes are user-defined codes that you customize to handle the tracking and reporting requirements of your organization.
Company	Enter a code that identifies a specific organization, fund, or other reporting entity. The company code must already exist in the F0010 table and must identify a reporting entity that has a complete balance sheet.
Currency Code	Enter the three-character code that represents the currency of the transaction. JD Edwards EnterpriseOne provides currency codes that are recognized by the International Organization for Standardization (ISO). The system stores currency codes in the F0013 table.
Document Company	Enter the company number associated with the document. This number, used in conjunction with the document number, document type, and general ledger date, uniquely identifies an original document. If you assign next numbers by company and fiscal year, the system uses the document company to retrieve the correct next number for that company. If two or more original documents have the same document number and document type, you can use the document company to display the document that you want.
Document Number	Displays a number that identifies the original document, which can be a voucher, invoice, journal entry, or time sheet, and so on. On entry forms, you can assign the original document number or the system can assign it through the Next Numbers program.
Document Type	Enter the two-character UDC, from UDC table 00/DT, that identifies the origin and purpose of the transaction, such as a voucher, invoice, journal entry, or time sheet. JD Edwards EnterpriseOne reserves these prefixes for the document types indicated: *P:* Accounts payable documents. *R:* Accounts receivable documents. *T:* Time and pay documents. *I:* Inventory documents. *O:* Purchase order documents. *S:* Sales order documents.

Effective Date	Enter the date on which an address, item, transaction, or record becomes active. The meaning of this field differs, depending on the program. For example, the effective date can represent any of these dates: • The date on which a change of address becomes effective. • The date on which a lease becomes effective. • The date on which a price becomes effective. • The date on which the currency exchange rate becomes effective. • The date on which a tax rate becomes effective.
Fiscal Period and **Fiscal Year**	Enter a number that identifies the general ledger period and year. For many programs, you can leave these fields blank to use the current fiscal period and year defined in the Company Names & Number program (P0010).
G/L Date (general ledger date)	Enter the date that identifies the financial period to which a transaction will be posted. The system compares the date that you enter on the transaction to the fiscal date pattern assigned to the company to retrieve the appropriate fiscal period number and year, as well as to perform date validations.

JD Edwards EnterpriseOne Tools Business Services Development Methodology Preface

This preface discusses Business Services Development Methodology companion documentation.

Business Services Development Methodology Companion Documentation

Additional, essential information describing the setup and design of Oracle's JD Edwards EnterpriseOne Tools Business Services resides in companion documentation. You should be familiar with the information in these companion guides:

- *JD Edwards EnterpriseOne Tools Business Services Development Guide*.
- *JD Edwards EnterpriseOne Business Services Server Reference Guide*.
- *JD Edwards EnterpriseOne Interoperability Reference Implementations Guide*.
- *JD Edwards EnterpriseOne Tools Server Manager Guide*.
- *JD Edwards EnterpriseOne Tools Reference Guide*.
- *JD Edwards EnterpriseOne Package Management Guide*.
- *JD Edwards EnterpriseOne Object Management Workbench Guide*.
- *JD Edwards EnterpriseOne Security Administration Guide*.

Customers must conform to the supported platforms for the release as detailed in the JD Edwards EnterpriseOne minimum technical requirements. In addition, JD Edwards EnterpriseOne may integrate, interface, or work in conjunction with other Oracle products. Refer to the cross-reference material in the Program Documentation at http://oracle.com/contracts/index.html for Program prerequisites and version cross-reference documents to assure compatibility of various Oracle products.

See Also

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide, "Getting Started with JD Edwards EnterpriseOne Tools Business Services Development"

JD Edwards EnterpriseOne Tools 8.98 Object Management Workbench Guide, "Getting Started with JD Edwards EnterpriseOne OMW," JD Edwards EnterpriseOne OMW Overview

JD Edwards EnterpriseOne Tools 8.98 Package Management Guide, "Getting Started with JD Edwards EnterpriseOne Package Management"

JD Edwards EnterpriseOne Tools 8.98 Security Administration Guide, "Using Security Workbench," Understanding Published Business Services Security

JD Edwards EnterpriseOne Tools Release 8.98 Business Services Server Reference Guide on Oracle | PeopleSoft Customer Connection.

JD Edwards EnterpriseOne Tools 8.98 Interoperability Reference Implementations Guide on Oracle | PeopleSoft Customer Connection.

JD Edwards EnterpriseOne Tools Release 8.98 Server Manager Guide on Oracle | PeopleSoft Customer Connection.

JD Edwards EnterpriseOne Tools Release 8.98 Tools Reference Guide on Oracle | PeopleSoft Customer Connection

CHAPTER 1

Getting Started with JD Edwards EnterpriseOne Tools Business Services Development Methodology

This chapter discusses:

- Business services development methodology overview.
- Business services development methodology implementation.

JD Edwards EnterpriseOne Tools Business Services Development Methodology Overview

Oracle's *JD Edwards EnterpriseOne Tools Business Services Development Methodology 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.

JD Edwards EnterpriseOne Tools Business Services Development Methodology Implementation

This section provides an overview of the steps that are required to implement JD Edwards EnterpriseOne Tools Business Services. In the planning phase of your implementation, take advantage of all JD Edwards sources of information, including the installation guides and troubleshooting information.

Business Services Development Methodology Implementation Steps

This table lists the steps for general business services development implementation. In addition to the JD Edwards EnterpriseOne Tools guides, install any other EnterpriseOne tools, such as the Transaction Server, that are required. The JD Edwards EnterpriseOne Server Manager guide and installation reference guides, such as Business Services Server Reference Guide and Transaction Server Reference guide, are available on Oracle | PeopleSoft Customer Connection.

Step	Reference
1. Install EnterpriseOne Tools 8.98.	JD Edwards EnterpriseOne Tools Release 8.98 Server Manager Guide on Oracle | PeopleSoft Customer Connection.
2. Deploy the Business Services packages to the Business Services Server.	JD Edwards EnterpriseOne Tools 8.98 Package Management Guide.
3. Configure the Business Services Server.	JD Edwards EnterpriseOne Tools Release 8.98 Business Services Server Reference Guide on Oracle | PeopleSoft Customer Connection.
4. Create business services workspace and projects on JDeveloper using Object Management Workbench.	JD Edwards EnterpriseOne Tools 8.98 Object Management Workbench Guide.
5. Create business services and published business services using wizards and other tools and guidelines, rules, and best practices.	JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide and JD Edwards EnterpriseOne Tools 8.98 Business Services Development Methodology Guide.

CHAPTER 2

Understanding Business Services

This chapter discusses:

- JD Edwards EnterpriseOne business services.
- Development methodology.
- Value objects.
- Package naming and structure.
- Java coding standards.

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.

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.

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.

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.

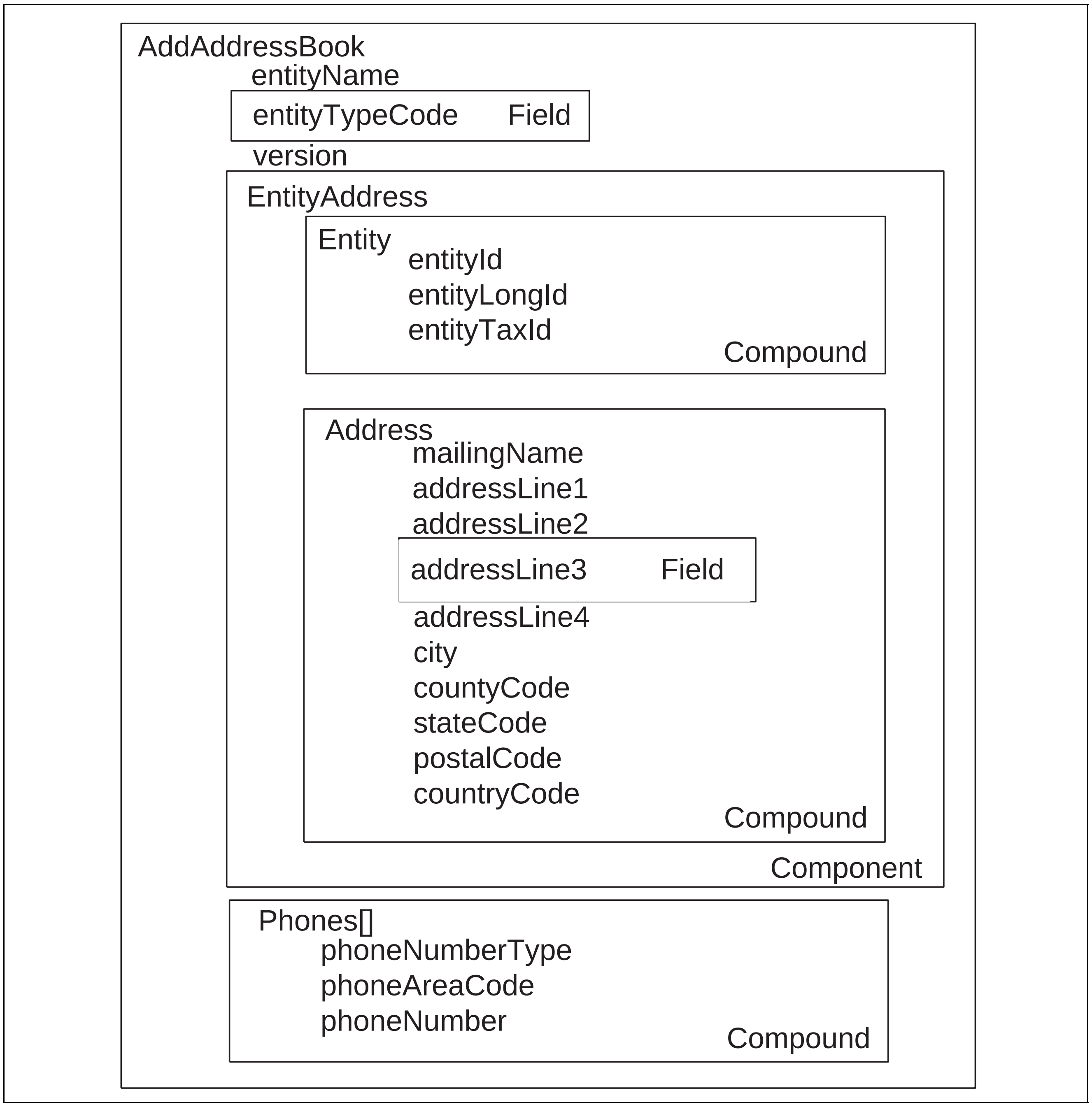
JD Edwards EnterpriseOne business service and published business service classes use value object classes. A value object is an interface to a business service or a published business service. A value object is the high-level component that contains the business data that defines a business process. 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.

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:



Value object structure

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

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.

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.

Fields

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

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:

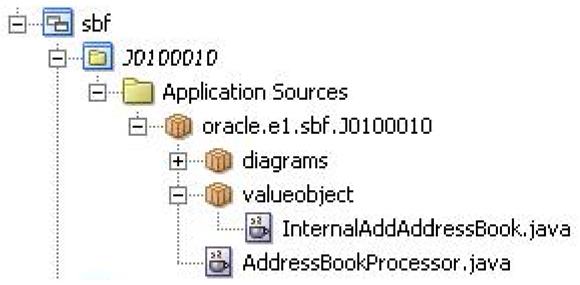
package oracle.e1.bssv.*JP010000*

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).

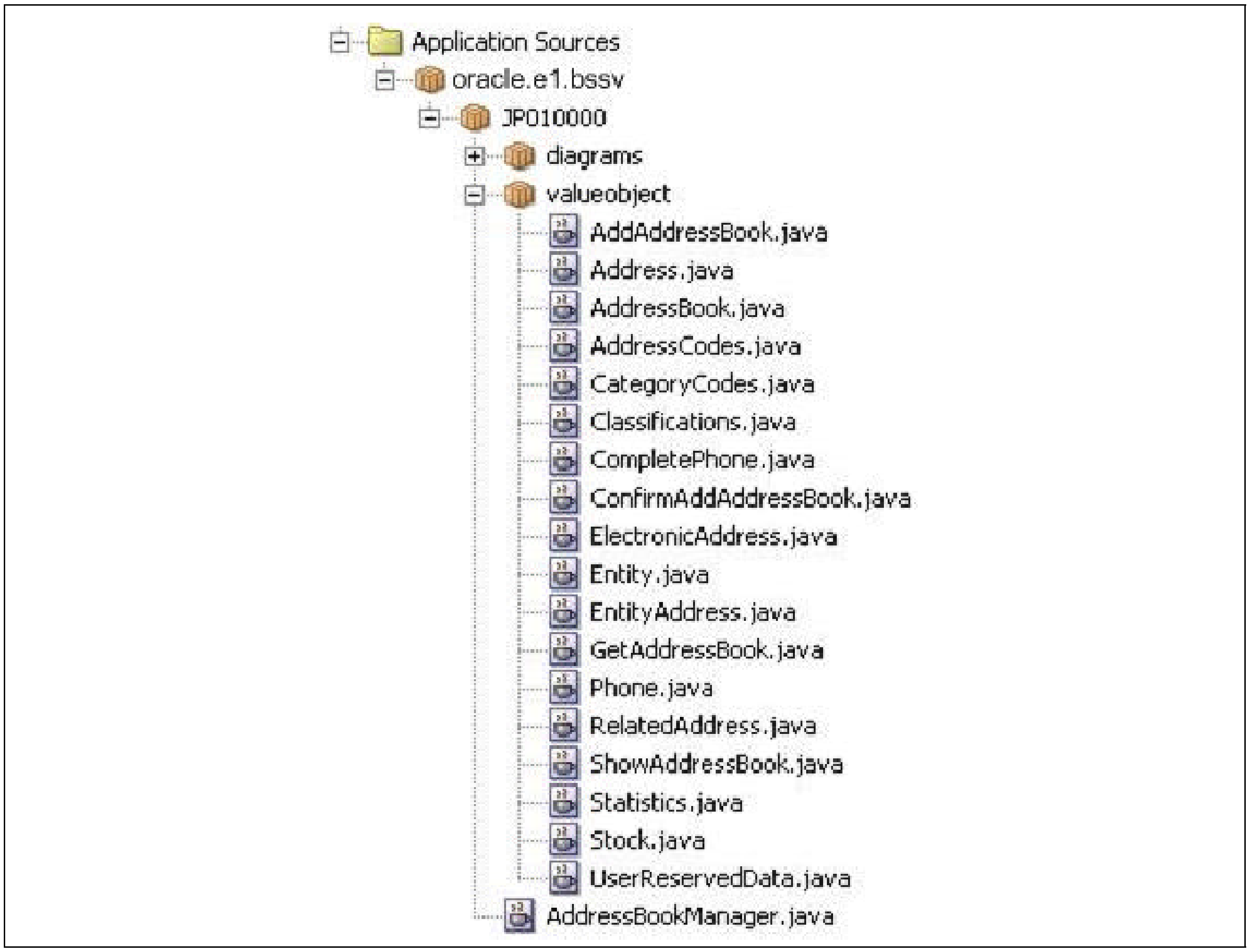
In the preceding examples, the portion of the name in italic font is the business service object name. To be compatible with OMW object names, this portion of the package name must be eight characters. The naming convention for the OMW object name is different for business service and published business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



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:



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.

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();
 }
```

CHAPTER 3

Creating a Published Business Service

This chapter provides an overview of published business services and discusses how to:

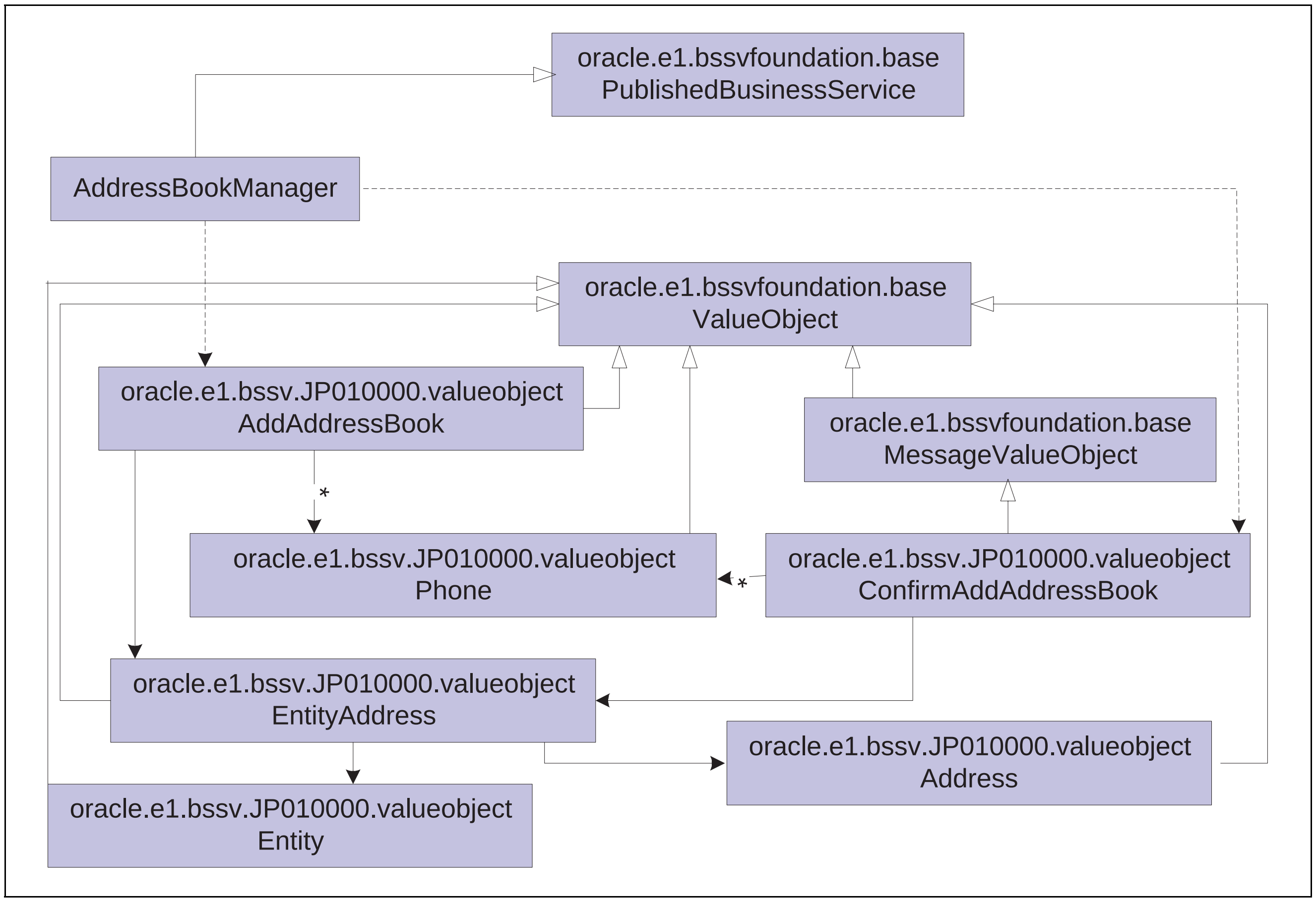
- Develop a published business service.
- Manage published business service components.
- Call a business service.
- Handle errors in the published business service.
- Test a published business service.
- Customize a published business service.
- Deprecate a published business service.

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:


oracle.e1.bssvfoundation.base
PublishedBusinessService
AddressBookManager
oracle.e1.bssvfoundation.base
ValueObject
oracle.e1.bssv.JP010000.valueobject
AddAddressBook
oracle.e1.bssvfoundation.base
MessageValueObject
*
oracle.e1.bssv.JP010000.valueobject
Phone
*
oracle.e1.bssv.JP010000.valueobject
ConfirmAddAddressBook
oracle.e1.bssv.JP010000.valueobject
EntityAddress
oracle.e1.bssv.JP010000.valueobject
Address
oracle.e1.bssv.JP010000.valueobject
Entity

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.

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.

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.

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");
       }
```

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.

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 {
 ....
 }
```

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{
 ...
 }
```

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 {
   ...
   }
```

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;
....
}
```

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 *JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide*, "Creating Business Services," Understanding Business Services.

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() {
      }
  ....
  }
```

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.

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 get and set 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 get and set 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;
    }
```

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.

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`

Rules

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

- Implement the serializable 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.

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;
   ....
   }
```

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;
   ....
   }
```

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 E1MessagleList:

```
 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);
          }
       }
    }
```

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

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());
}
```

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.
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);
```

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()));
          }
        }
```

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.

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");
       }
   }
```

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;
```

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.

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.

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 JDeveloper Guide, http://www.oracle.com/technology/index.html.

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.

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");
       }
   }
```

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");
           }

       }
```

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.

CHAPTER 4

Creating a Business Service

This chapter provides an overview of business services and discusses how to:

- Develop a business service.
- Manage business service components.
- Call business functions.
- Call database operations.
- Call other business services.
- Manage business service properties.
- Handle errors in the business service.
- Modify a business service.
- Document a business service.

Understanding Business Services

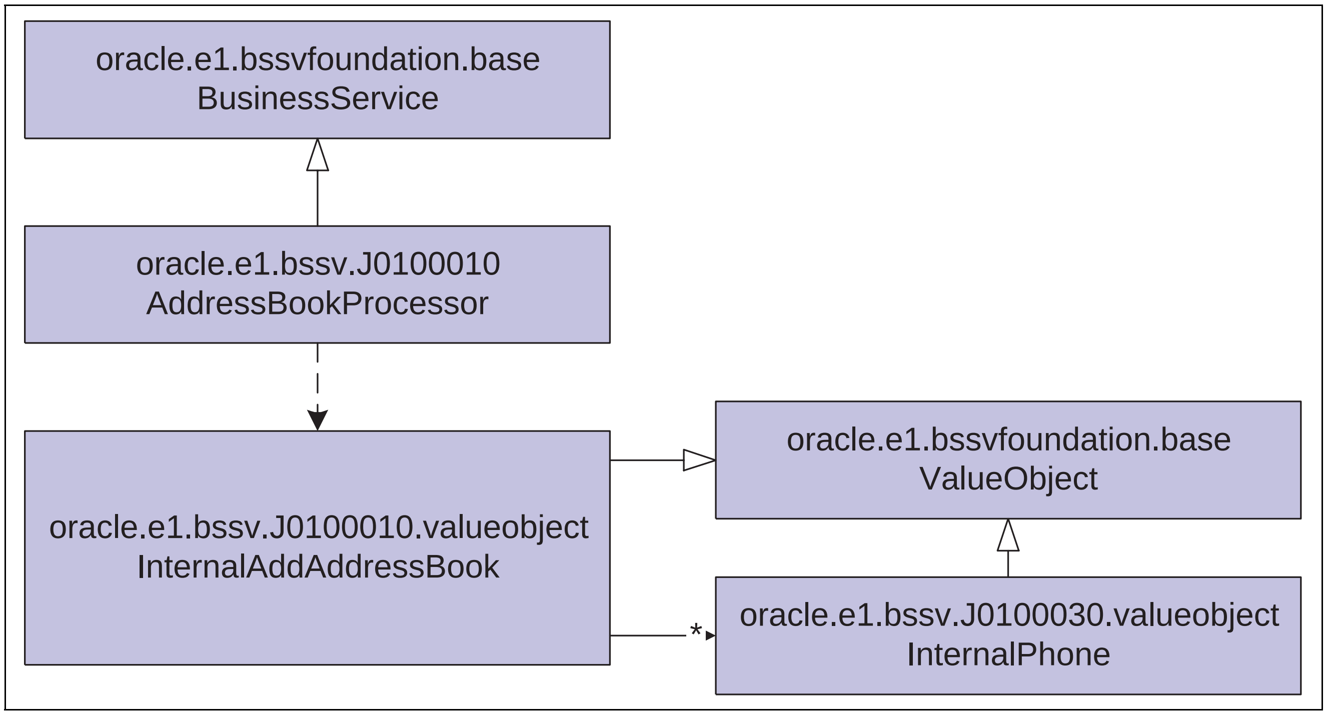
Business services are JD Edwards EnterpriseOne Object Management Workbench (OMW) objects that are called by a published business service to accomplish a specific task. Business service classes are written in Java programming language and provide methods that access the business logic in JD Edwards EnterpriseOne 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:


oracle.e1.bssvfoundation.base
BusinessService
oracle.e1.bssv.J0100010
AddressBookProcessor
oracle.e1.bssv.J0100010.valueobject
InternalAddAddressBook
oracle.e1.bssvfoundation.base
ValueObject
oracle.e1.bssv.J0100030.valueobject
InternalPhone
*

Business service class diagram

These features are illustrated in the diagram:

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

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.

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

Chapter 6, "Understanding Transaction Processing," Transaction Processing, page 81

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.

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 {
   ....
```

```
      }
```

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){
   ...
   }
```

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 Chapter 5, "Creating Business Services That Call Database Operations," Understanding Database Operations, page 61.

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.

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;
   ...
 }
```

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 *JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide*, "Creating Business Services," Creating Business Service Classes.

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.

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;
   }
```

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.

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.

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 (get and set 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 Chapter 5, "Creating Business Services That Call Database Operations," Understanding Database Operations, page 61.

See *JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide*, "Creating Business Services," Creating Business Function Calls.

See *JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide*, "Creating Business Services," Creating Database Operation Calls.

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.

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("mn⇒ProcessID"));
       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();
   }
```

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 *JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide*, "Creating Business Services," Understanding Business Function Calls.

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;
}
```

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

Chapter 5, "Creating Business Services That Call Database Operations," Understanding Database Operations, page 61

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide, "Creating Business Services," Understanding Database Operation Calls

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);
     }
```

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

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide, "Working with Business Service Properties," Understanding Business Service Properties

JD Edwards EnterpriseOne Tools 8.98 Object Management Workbench Guide, "Working with Business Services," Working with Business Services Properties

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.

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

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.

Business Service Property Name	Usage
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

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);
               }
```

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.

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.

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);
  ```

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;
```

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.

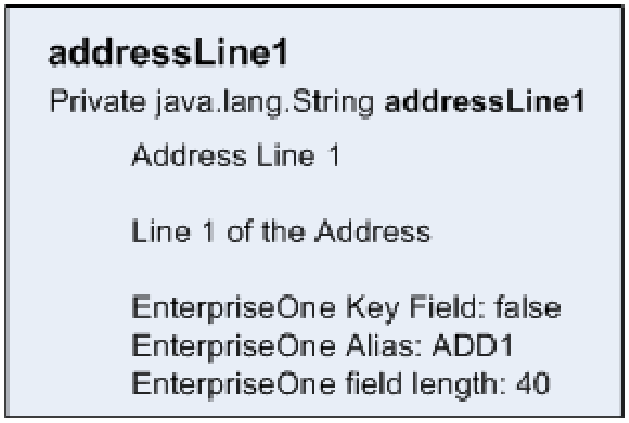
Documenting a Business Service

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
 * tENERPRISEoNE aLIAS: add1
 * EnterpriseOne field length:  40
 */
 private String addressLine1 = null;
```

This documentation is a result of the preceding Javadoc:



addressLine1

Private java.lang.String **addressLine1**

Address Line 1

Line 1 of the Address

EnterpriseOne Key Field: false
EnterpriseOne Alias: ADD1
EnterpriseOne field length: 40

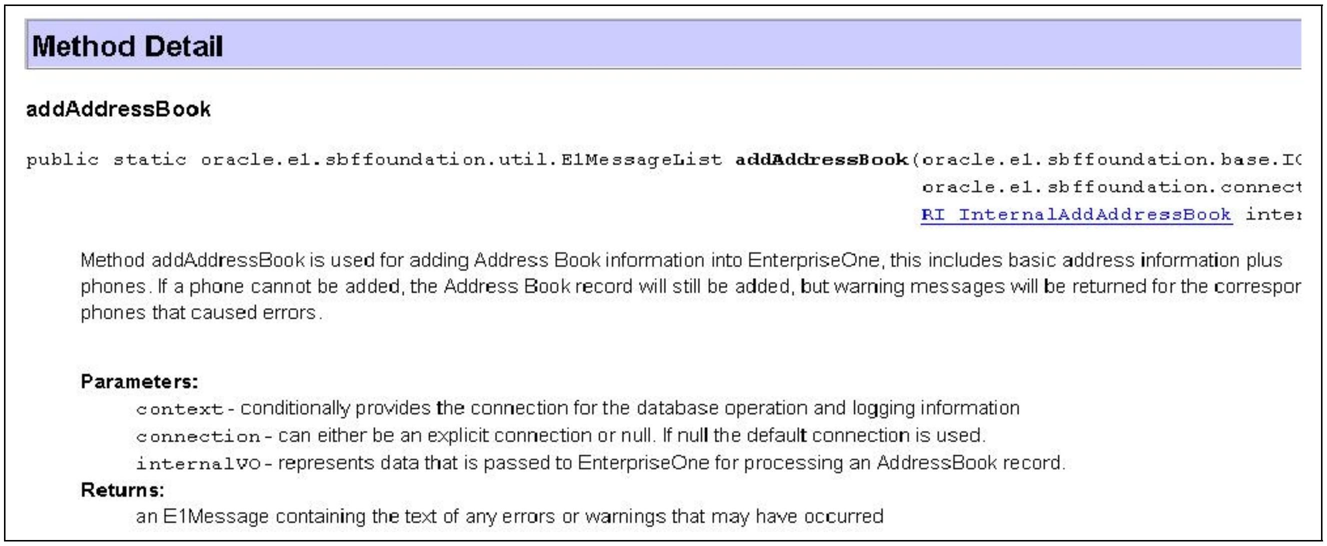
Javadoc documentation

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:


Method Detail

addAddressBook

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

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

Generated documentation resulting from Javadoc code

CHAPTER 5

Creating Business Services That Call Database Operations

This chapter provides an overview of database operations and discusses how to:

- Create a query database operation business service.
- Create an insert database operation business service.
- Create an update database operation business service.
- Create a delete database operation business service.

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.

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.
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.

Published Value Object Data Type	Usage
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.

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()));
       }
```

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.

Published Value Object for Query

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

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.

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[];
```

```
        . . .
    }
```

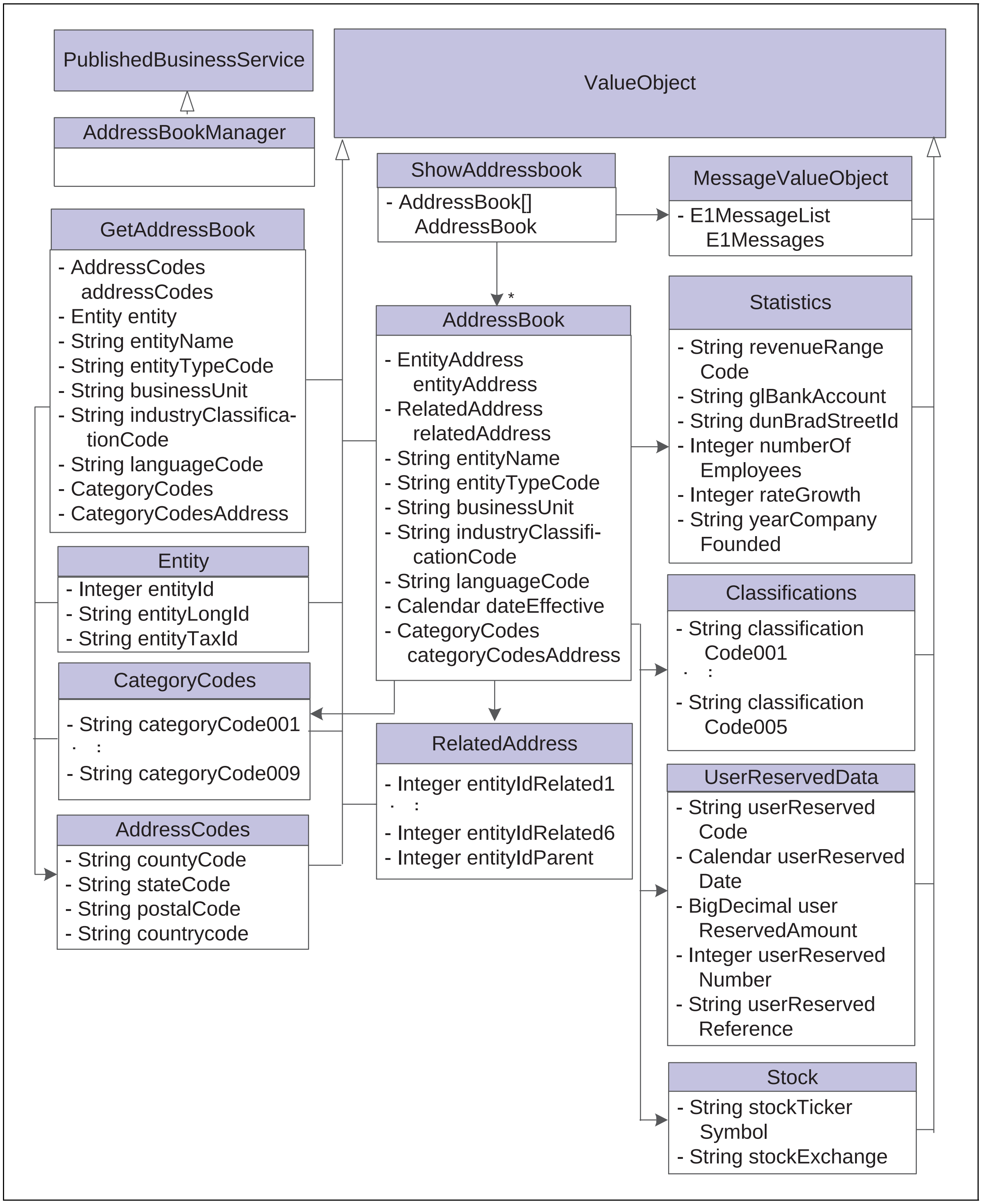
Error Handling

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.

Class Diagram

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



Published business service, GetAddressBook, class diagram.

Internal Value Object for Query

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

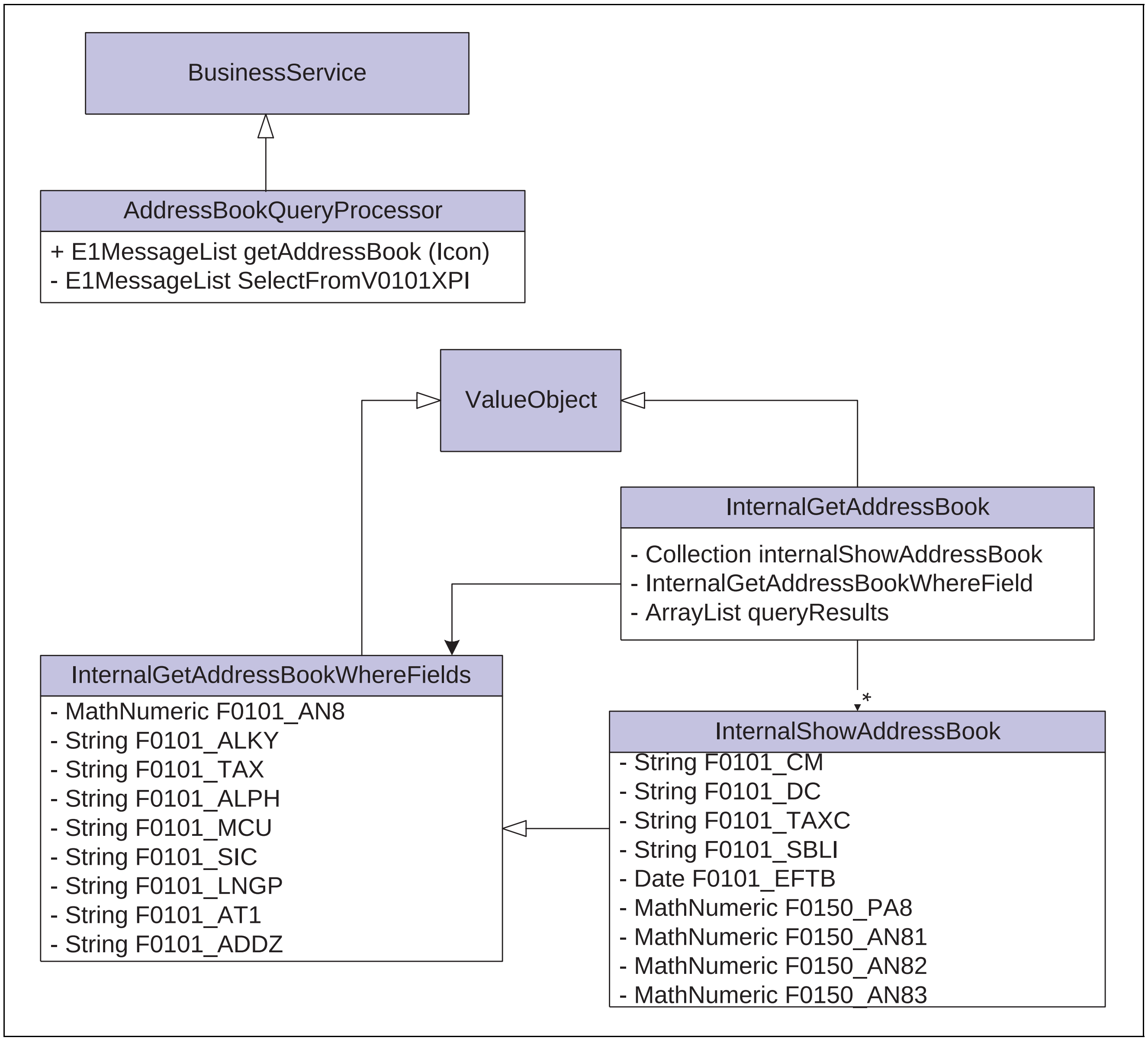
The names that you use for variables in the internal value object are important because the generated code uses these names when calling the get and set methods for these objects.

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:



Business service, GetAddressbook, class diagram.

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.

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.

Published Value Object for Insert

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

Naming Conventions

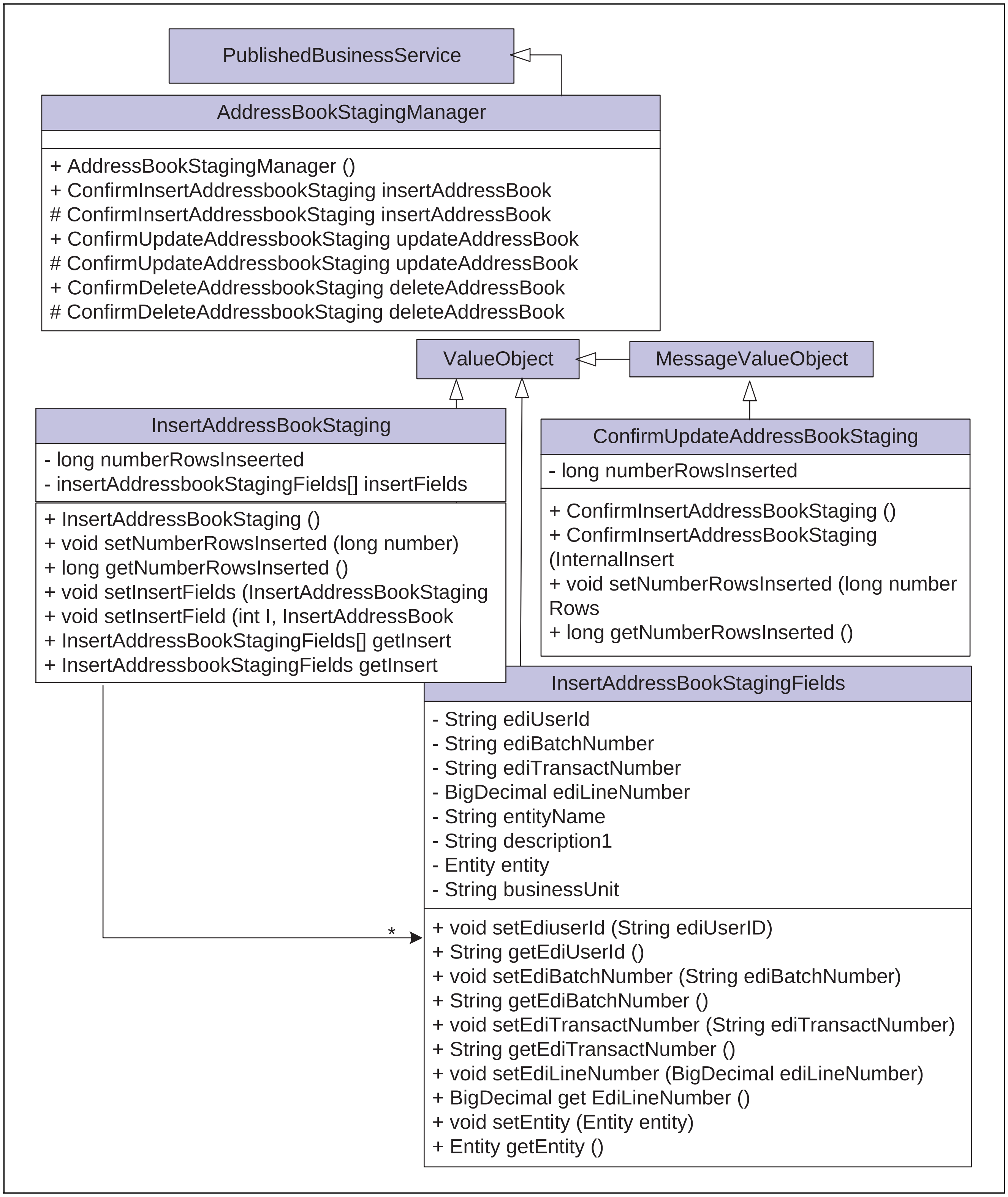
The naming convention for an input value object is to use the verb *insert* to preface the type of data to be processed; for example, *InsertAddressBookStaging*. The naming convention for an output value object is to use the verb phrase *ConfirmInsert* to preface the information that is processed, for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

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.

Class Diagram

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

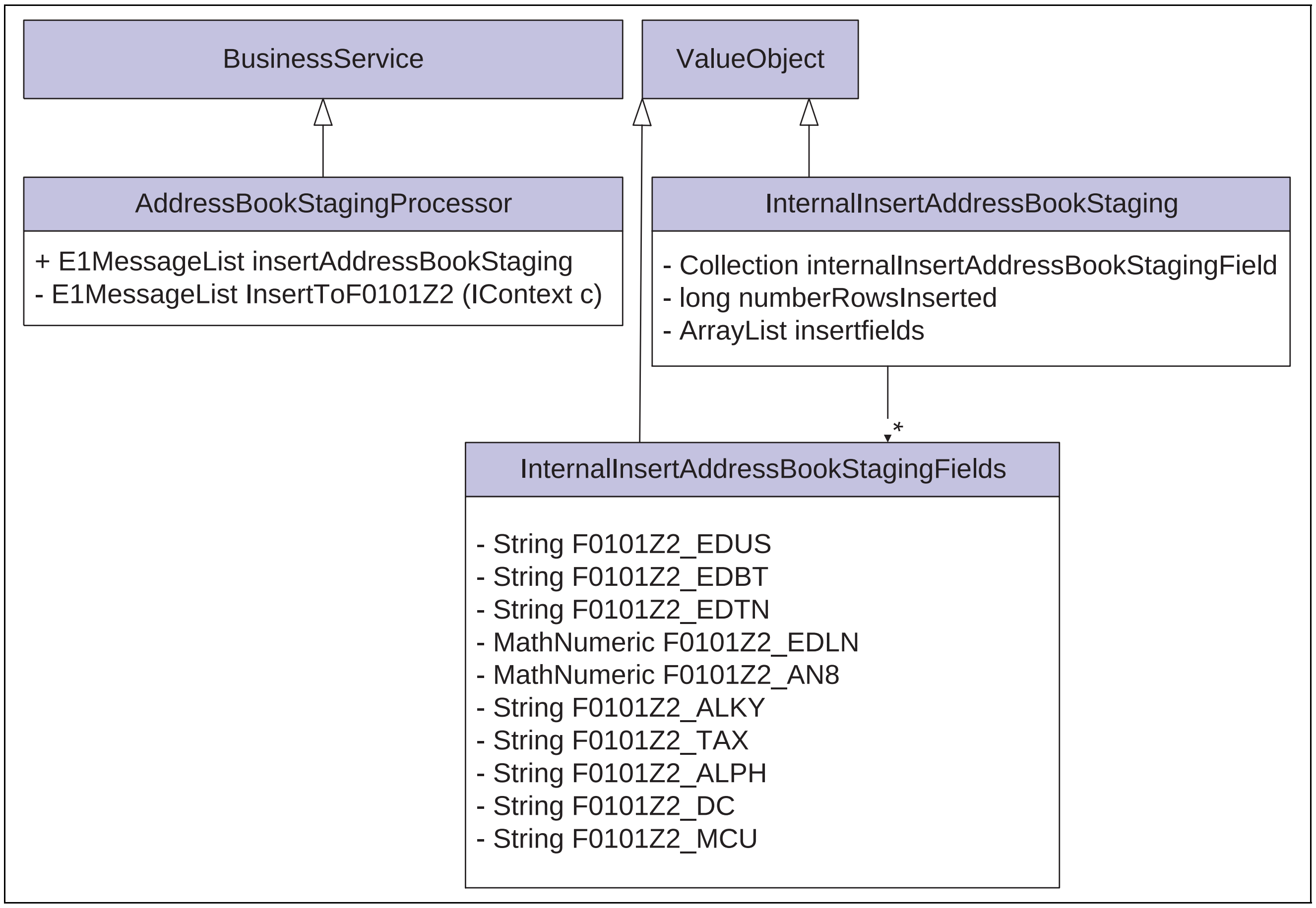

PublishedBusinessService
AddressBookStagingManager
+ AddressBookStagingManager ()
+ ConfirmInsertAddressbookStaging insertAddressBook
# ConfirmInsertAddressbookStaging insertAddressBook
+ ConfirmUpdateAddressbookStaging updateAddressBook
# ConfirmUpdateAddressbookStaging updateAddressBook
+ ConfirmDeleteAddressbookStaging deleteAddressBook
# ConfirmDeleteAddressbookStaging deleteAddressBook
ValueObject
MessageValueObject
InsertAddressBookStaging
- long numberRowsInseerted
- insertAddressbookStagingFields[] insertFields
+ InsertAddressBookStaging ()
+ void setNumberRowsInserted (long number)
+ long getNumberRowsInserted ()
+ void setInsertFields (InsertAddressBookStaging
+ void setInsertField (int I, InsertAddressBook
+ InsertAddressBookStagingFields[] getInsert
+ InsertAddressbookStagingFields getInsert
ConfirmUpdateAddressBookStaging
- long numberRowsInserted
+ ConfirmInsertAddressBookStaging ()
+ ConfirmInsertAddressBookStaging (InternalInsert
+ void setNumberRowsInserted (long number Rows
+ long getNumberRowsInserted ()
InsertAddressBookStagingFields
- String ediUserId
- String ediBatchNumber
- String ediTransactNumber
- BigDecimal ediLineNumber
- String entityName
- String description1
- Entity entity
- String businessUnit
+ void setEdiuserId (String ediUserID)
+ String getEdiUserId ()
+ void setEdiBatchNumber (String ediBatchNumber)
+ String getEdiBatchNumber ()
+ void setEdiTransactNumber (String ediTransactNumber)
+ String getEdiTransactNumber ()
+ void setEdiLineNumber (BigDecimal ediLineNumber)
+ BigDecimal get EdiLineNumber ()
+ void setEntity (Entity entity)
+ Entity getEntity ()
*

Published business service, InsertAddressBookStaging, class diagram.

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:


BusinessService
ValueObject
AddressBookStagingProcessor
+ E1MessageList insertAddressBookStaging
- E1MessageList InsertToF0101Z2 (IContext c)
InternalInsertAddressBookStaging
- Collection internalInsertAddressBookStagingField
- long numberRowsInserted
- ArrayList insertfields
*
InternalInsertAddressBookStagingFields
- String F0101Z2_EDUS
- String F0101Z2_EDBT
- String F0101Z2_EDTN
- MathNumeric F0101Z2_EDLN
- MathNumeric F0101Z2_AN8
- String F0101Z2_ALKY
- String F0101Z2_TAX
- String F0101Z2_ALPH
- String F0101Z2_DC
- String F0101Z2_MCU
Business service, InternalInsertAddressBookStaging, class diagram.

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;
    }
```

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.

Published Value Object for Update

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

Naming Conventions

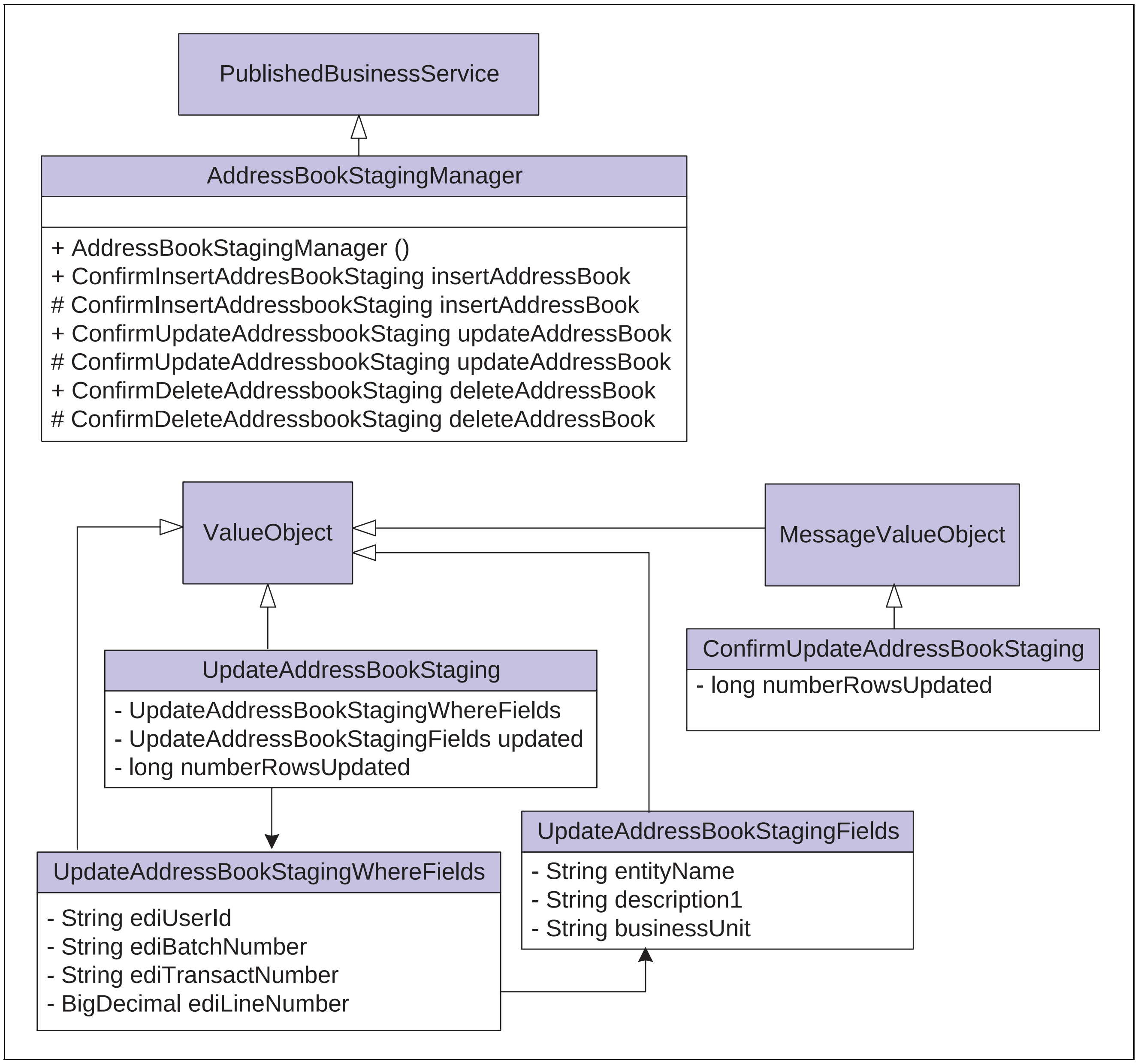
The naming convention for an update value object is to use the verb *update* to preface the type of data to be processed, for example, *UpdateAddressBookStaging*. The naming convention for an output value object is to use the verb phrase *ConfirmUpdate* to preface the information that is processed, for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

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.

Class Diagram

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


PublishedBusinessService
AddressBookStagingManager
+ AddressBookStagingManager ()
+ ConfirmInsertAddresBookStaging insertAddressBook
# ConfirmInsertAddressbookStaging insertAddressBook
+ ConfirmUpdateAddressbookStaging updateAddressBook
# ConfirmUpdateAddressbookStaging updateAddressBook
+ ConfirmDeleteAddressbookStaging deleteAddressBook
# ConfirmDeleteAddressbookStaging deleteAddressBook
ValueObject
MessageValueObject
UpdateAddressBookStaging
- UpdateAddressBookStagingWhereFields
- UpdateAddressBookStagingFields updated
- long numberRowsUpdated
ConfirmUpdateAddressBookStaging
- long numberRowsUpdated
UpdateAddressBookStagingFields
- String entityName
- String description1
- String businessUnit
UpdateAddressBookStagingWhereFields
- String ediUserId
- String ediBatchNumber
- String ediTransactNumber
- BigDecimal ediLineNumber

Published business service, UpdateAddressBookStaging, class diagram.

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:

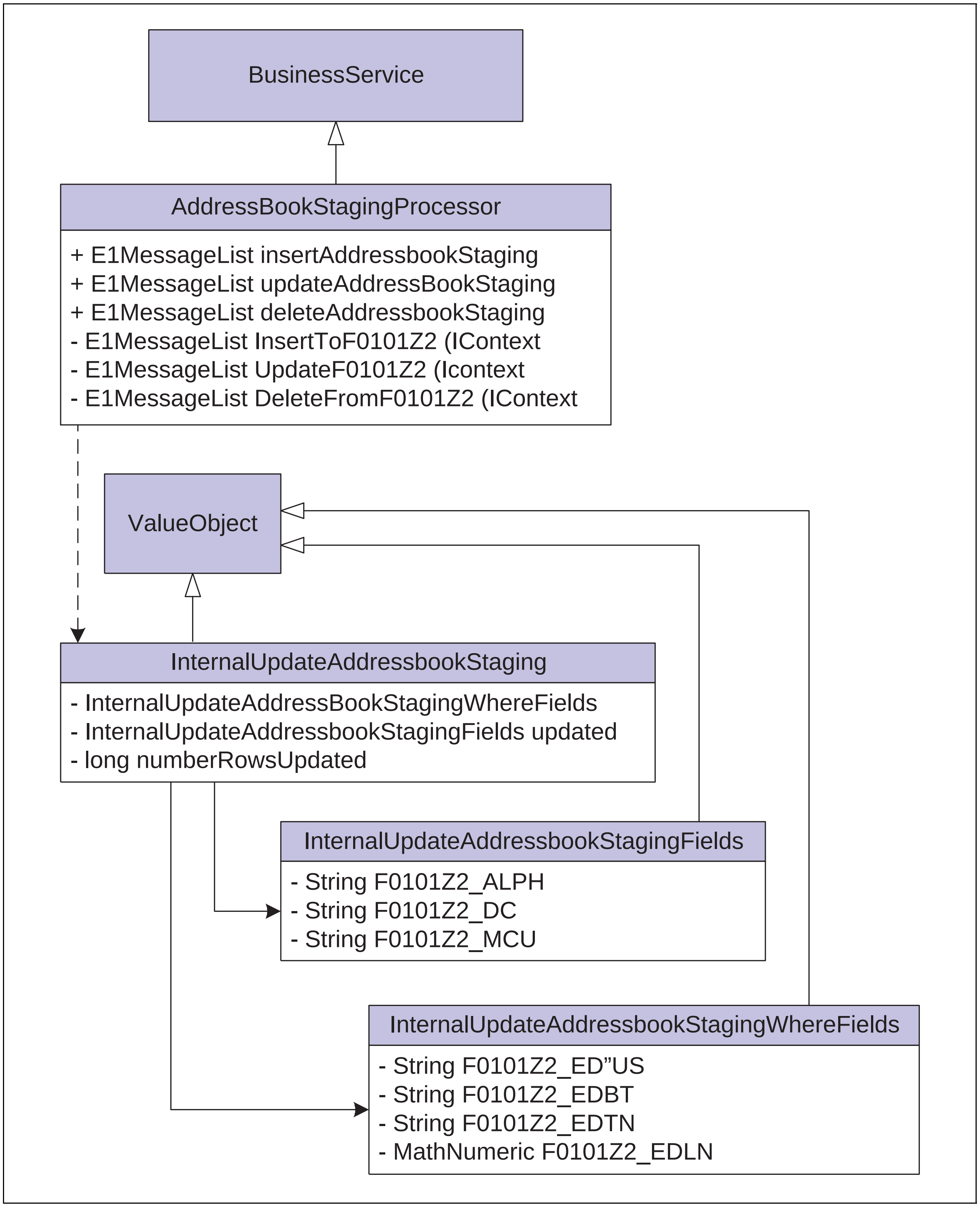
```
public class InternalUpdateAddressBookStaging extends ValueObject {
   /**
    * Internal VO representing the where clause for updating the
    * F0101Z2 table.
    */
   private InternalUpdateAddressBookStagingWhereFields
updateWhereFields = new InternalUpdateAddressBookStagingWhereFields();
  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


BusinessService
AddressBookStagingProcessor
+ E1MessageList insertAddressbookStaging
+ E1MessageList updateAddressBookStaging
+ E1MessageList deleteAddressbookStaging
- E1MessageList InsertToF0101Z2 (IContext
- E1MessageList UpdateF0101Z2 (Icontext
- E1MessageList DeleteFromF0101Z2 (IContext
ValueObject
InternalUpdateAddressbookStaging
- InternalUpdateAddressBookStagingWhereFields
- InternalUpdateAddressbookStagingFields updated
- long numberRowsUpdated
InternalUpdateAddressbookStagingFields
- String F0101Z2_ALPH
- String F0101Z2_DC
- String F0101Z2_MCU
InternalUpdateAddressbookStagingWhereFields
- String F0101Z2_ED”US
- String F0101Z2_EDBT
- String F0101Z2_EDTN
- MathNumeric F0101Z2_EDLN

Business service, UpdateASddressBookStaging, class diagram.

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.

Published Value Object for Delete

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

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*.

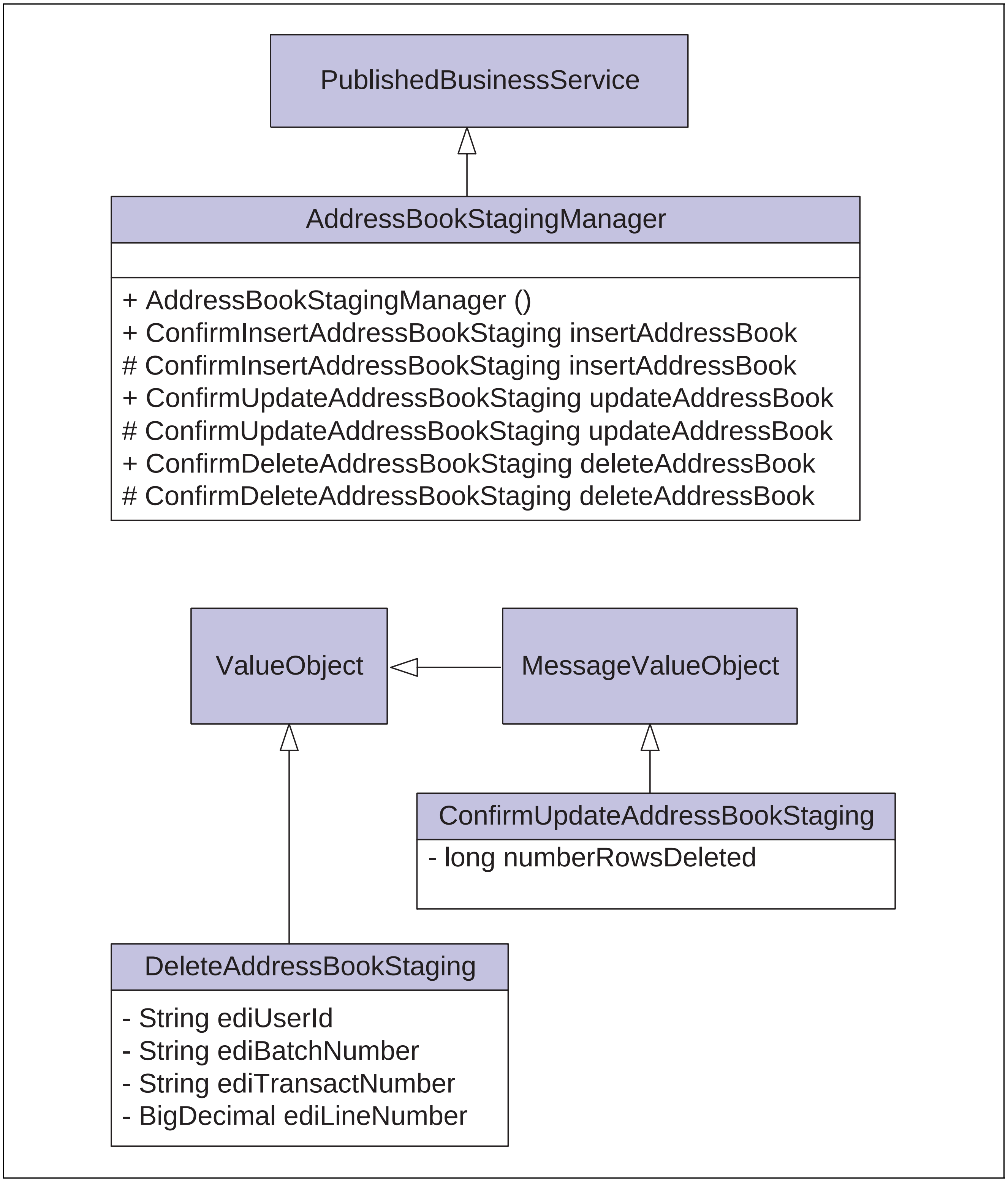
Data Types and Structure

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.

Class Diagram

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

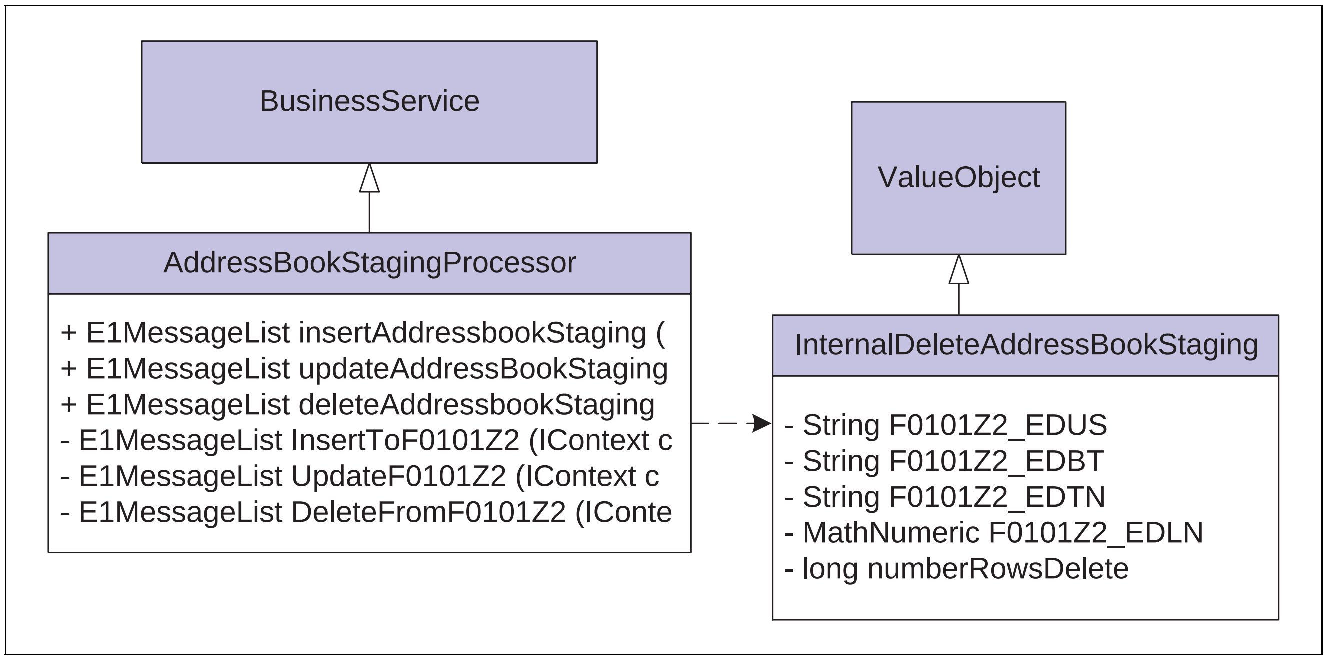

PublishedBusinessService
AddressBookStagingManager
+ AddressBookStagingManager ()
+ ConfirmInsertAddressBookStaging insertAddressBook
# ConfirmInsertAddressBookStaging insertAddressBook
+ ConfirmUpdateAddressBookStaging updateAddressBook
# ConfirmUpdateAddressBookStaging updateAddressBook
+ ConfirmDeleteAddressBookStaging deleteAddressBook
# ConfirmDeleteAddressBookStaging deleteAddressBook
ValueObject
MessageValueObject
ConfirmUpdateAddressBookStaging
- long numberRowsDeleted
DeleteAddressBookStaging
- String ediUserId
- String ediBatchNumber
- String ediTransactNumber
- BigDecimal ediLineNumber

Published business service, DeleteAddressBookStaging, class diagram.

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:


BusinessService
AddressBookStagingProcessor
+ E1MessageList insertAddressbookStaging (
+ E1MessageList updateAddressBookStaging
+ E1MessageList deleteAddressbookStaging
- E1MessageList InsertToF0101Z2 (IContext c
- E1MessageList UpdateF0101Z2 (IContext c
- E1MessageList DeleteFromF0101Z2 (IConte
ValueObject
InternalDeleteAddressBookStaging
- String F0101Z2_EDUS
- String F0101Z2_EDBT
- String F0101Z2_EDTN
- MathNumeric F0101Z2_EDLN
- long numberRowsDelete

Business service, DeleteAddressBookStaging, class diagram.

CHAPTER 6

Understanding Transaction Processing

This chapter discusses:

- Transaction processing
- Default transaction processing behavior
- Explicit transaction processing behavior

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

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.

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.

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.

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.

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");
        }
  }
```

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.

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.

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.

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;
  }
```

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;
  }
```

CHAPTER 7

Understanding Logging

This chapter discusses logging.

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.

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
```

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
```

CHAPTER 8

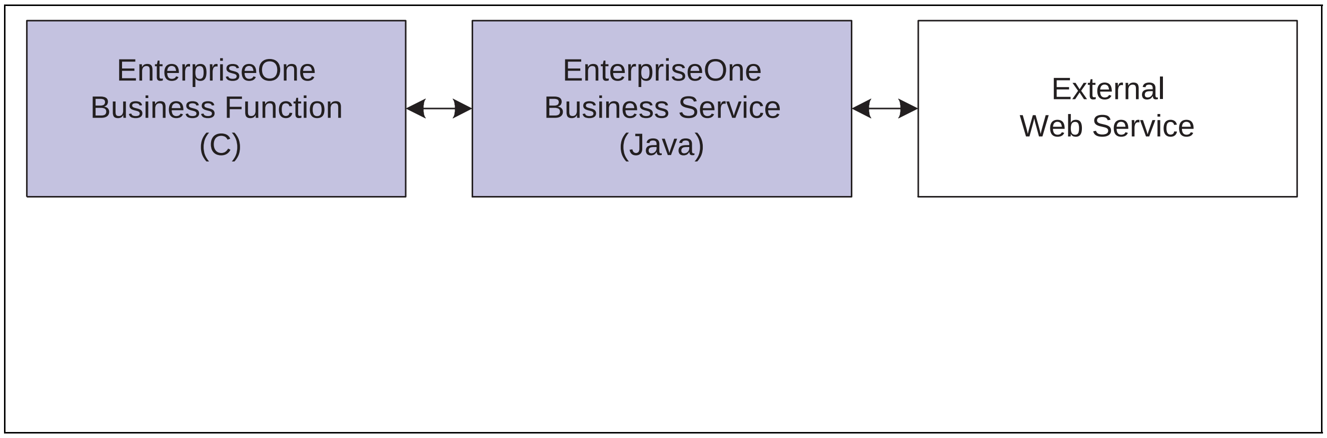
Understanding JD Edwards EnterpriseOne as a Web Service Consumer

This chapter discusses:

- JD Edwards EnterpriseOne as a web service consumer.
- C business function calling a business service.
- Creating a business service for JD Edwards EnterpriseOne as a web service consumer.
- Using Softcoding.
- Testing the business service for JD Edwards EnterpriseOne as a web service consumer.

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.


EnterpriseOne Business Function (C)
EnterpriseOne Business Service (Java)
External Web Service

Process flow for JD Edwards EnterpriseOne as a web service consumer.

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.

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.

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 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

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;
```

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.

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

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide, "Working with Softcoding," Understanding Softcoding

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 *Business Services Development Guide.*

See Also

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide, "Testing a Business Service That Consumes an External Web Service," Creating a Test Business Service

CHAPTER 9

Using Business Services with HTTP Request/Reply

This chapter provides an overview of using business services to make an HTTP POST to a third-party system and discusses:

- Using business services with HTTP Request/Reply.
- Testing the servlet.

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

JD Edwards EnterpriseOne Tools 8.98 Business Services Development Guide, "Working with HTTP Request/Response," Understanding Business Services and HTTP POST

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.

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.

APPENDIX A

Utility Business Services

This appendix provides an overview of the utility business services and discusses these business services:

- Entity Processor business service.
- GL Account Processor business service.
- Inventory Item ID Processor business service.
- Net Change Processor business service.
- Processing Version Processor business service.

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.

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.

Entity Processor Business Service

This section discusses the Entity Processor business service.

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.

Implementation Detail

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

Methods

Methods for the business service are:

- processEntity(Entity)
- processEntity(InternalEntityUtility)

Signature

The signature for the business service is:

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

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.

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.

Value Object Classes

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

Business Service Value Object

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

Published Reusable Value Object

Entity				
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

Output from Business Service to Published Value Object

InternalEntityUtility		Entity		
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

GL Account Processor Business Service

This section discusses the GL Account Processor business service.

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.

Implementation Detail

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

Methods

Methods for the business service are:

- processGLAccount(InternalGLAccountUtility)
- processGLAccount(ProcessGLAccount)

Signature

The signature for the business service is:

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

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.

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.

Value Object Classes

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

Business Service Input and Output Interface

InternalGLAccountUtility		
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.

Published Reusable Value Object

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

Published to Business Service Value Object

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

Inventory Item ID Processor Business Service

This section discusses the Inventory Item ID Processor business service.

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.

Implementation Detail

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

Methods

Methods for the business service are:

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

Signature

The signature for the business service is:

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

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.

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.

Value Object Classes

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

Business Service Value Object

InternalInventoryItemId		
Business Service VO Field Name	**Data Type**	**Input/Output**
mnShortItemNumber	MathNumeric	Input and Output

InternalInventoryItemId		
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

Business Service Value Object

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

Input Business Service Processing

From VO/BSFN/Business Service Property/Other		To BSFN/Other		
Field Name	**Data Type**	**Field Name**	**Data Type**	**Transformer**
Based on Input VO		**VerifyAndGetBranchPlant Constants (B41001390 / D41001390A)**		
szBranchPlant	String	szBranchPlant	String	Map
ItemIdsByXref-MODE				
		VerifyAndGetItemXref (B4100600 / D4100600)		
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
		getItemIdsByItemId (internal function)		
mnShortItemNumber	MathNumeric	mnShortItemNumber	MathNumeric	Map
ItemIdsByItemFreeform – MODE				
		GetItemMasterDescription UOM (B4001040 / D4001040)		
szFreeFormItemNumber	String	szPrimaryItemNumber	String	Map
szBranchPlant	String	szBranchPlant	String	Map
ItemIdsByItemId – MODE				
		GetItemMasterByShortItem (X4101 / DSDX4101B)		
mnShortItemNumber	MathNumeric	mnShortItemNumber	MathNumeric	Map

From VO/BSFN/Business Service Property/Other		To BSFN/Other		
ItemIdsbyItemFreeForm – MODE				
		GetItemMasterBy2ndItem (X4101 / DSDX4101C)		
sz2ndItemNumber	String	sz2ndItemNumber	String	Map
ItemIdsByItemCatalog – Mode				
		GetItemMasterBy3rdItem (X4101 / DSDX4101D)		
sz3rdItemNumber	String	sz3rdItemNumber	String	Map

Net Change Processor Business Service

This section discusses the Net Change Processor business service.

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.

Implementation Detail

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

Method

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

Signature

The signature for the business service is:

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

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.

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.

Method

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

Signature

The signature for the business service is:

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

Value Objects

This utility business service has no specific value objects.

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.

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.

Processing Version Processor Business Service

This section discusses the Processing Version Processor business service.

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.

Implementation Detail

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

Method

The method for the business service is getProcessingVersion.

Signature

The signature for the business service is:

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

Value Object

The value object for the business service is InternalProcessingVersion.

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.

Value Object Classes

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

Business Service Value Object

InternalProcessAddressBook			
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.

Business Service Value Object

InternalProcessAddressBook			
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 of JD Edwards EnterpriseOne Terms

Accessor Methods/Assessors	Java methods to "get" and "set" the elements of a value object or other source file.
activity rule	The criteria by which an object progresses from one given point to the next in a flow.
add mode	A condition of a form that enables users to input data.
Advanced Planning Agent (APAg)	A JD Edwards EnterpriseOne tool that can be used to extract, transform, and load enterprise data. APAg supports access to data sources in the form of rational databases, flat file format, and other data or message encoding, such as XML.
alternate currency	A currency that is different from the domestic currency (when dealing with a domestic-only transaction) or the domestic and foreign currency of a transaction. In JD Edwards EnterpriseOne Financial Management, alternate currency processing enables you to enter receipts and payments in a currency other than the one in which they were issued.
Application Server	Software that provides the business logic for an application program in a distributed environment. The servers can be Oracle Application Server (OAS) or WebSphere Application Server (WAS).
as if processing	A process that enables you to view currency amounts as if they were entered in a currency different from the domestic and foreign currency of the transaction.
as of processing	A process that is run as of a specific point in time to summarize transactions up to that date. For example, you can run various JD Edwards EnterpriseOne reports as of a specific date to determine balances and amounts of accounts, units, and so on as of that date.
Auto Commit Transaction	A database connection through which all database operations are immediately written to the database.
back-to-back process	A process in JD Edwards EnterpriseOne Supply Management that contains the same keys that are used in another process.
batch processing	A process of transferring records from a third-party system to JD Edwards EnterpriseOne. In JD Edwards EnterpriseOne Financial Management, batch processing enables you to transfer invoices and vouchers that are entered in a system other than JD Edwards EnterpriseOne to JD Edwards EnterpriseOne Accounts Receivable and JD Edwards EnterpriseOne Accounts Payable, respectively. In addition, you can transfer address book information, including customer and supplier records, to JD Edwards EnterpriseOne.
batch server	A server that is designated for running batch processing requests. A batch server typically does not contain a database nor does it run interactive applications.
batch-of-one immediate	A transaction method that enables a client application to perform work on a client workstation, then submit the work all at once to a server application for further processing. As a batch process is running on the server, the client application can continue performing other tasks. See also direct connect and store-and-forward.
best practices	Non-mandatory guidelines that help the developer make better design decisions.

BPEL	Abbreviation for *Business Process Execution Language*, a standard web services orchestration language, which enables you to assemble discrete services into an end-to-end process flow.
BPEL PM	Abbreviation for *Business Process Execution Language Process Manager*, a comprehensive infrastructure for creating, deploying, and managing BPEL business processes.
Build Configuration File	Configurable settings in a text file that are used by a build program to generate ANT scripts. ANT is a software tool used for automating build processes. These scripts build published business services.
build engineer	An actor that is responsible for building, mastering, and packaging artifacts. Some build engineers are responsible for building application artifacts, and some are responsible for building foundation artifacts.
Build Program	A WIN32 executable that reads build configuration files and generates an ANT script for building published business services.
business analyst	An actor that determines if and why an EnterpriseOne business service needs to be developed.
business function	A named set of user-created, reusable business rules and logs that can be called through event rules. Business functions can run a transaction or a subset of a transaction (check inventory, issue work orders, and so on). Business functions also contain the application programming interfaces (APIs) that enable them to be called from a form, a database trigger, or a non-JD Edwards EnterpriseOne application. Business functions can be combined with other business functions, forms, event rules, and other components to make up an application. Business functions can be created through event rules or third-generation languages, such as C. Examples of business functions include Credit Check and Item Availability.
business function event rule	See named event rule (NER).
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 artifacts	Source files, descriptors, and so on that are managed for business service development and are needed for the business service build process.
business service class method	A method that accesses resources provided by the business service framework.
business service configuration files	Configuration files include, but are not limited to, interop.ini, JDBj.ini, and jdelog.properties.
business service cross reference	A key and value data pair used during orchestration. Collectively refers to both the code and the key cross reference in the WSG/XPI based system.
business service cross-reference utilities	Utility services installed in a BPEL/ESB environment that are used to access JD Edwards EnterpriseOne orchestration cross-reference data.
business service development environment	A framework needed by an integration developer to develop and manage business services.
business services development tool	Otherwise known as JDeveloper.
business service EnterpriseOne object	A collection of artifacts managed by EnterpriseOne LCM tools. Named and represented within EnterpriseOne LCM similarly to other EnterpriseOne objects like tables, views, forms, and so on.

business service framework	Parts of the business service foundation that are specifically for supporting business service development.
business service payload	An object that is passed between an enterprise server and a business services server. The business service payload contains the input to the business service when passed to the business services server. The business service payload contains the results from the business service when passed to the Enterprise Server. In the case of notifications, the return business service payload contains the acknowledgement.
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 categorization	A way to categorize business service properties. These properties are categorized by business service.
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 service repository	A source management system, for example ClearCase, where business service artifacts and build files are stored. Or, a physical directory in network.
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.
business view	A means for selecting specific columns from one or more JD Edwards EnterpriseOne application tables whose data is used in an application or report. A business view does not select specific rows, nor does it contain any actual data. It is strictly a view through which you can manipulate data.
central objects merge	A process that blends a customer's modifications to the objects in a current release with objects in a new release.
central server	A server that has been designated to contain the originally installed version of the software (central objects) for deployment to client computers. In a typical JD Edwards EnterpriseOne installation, the software is loaded on to one machine—the central server. Then, copies of the software are pushed out or downloaded to various workstations attached to it. That way, if the software is altered or corrupted through its use on workstations, an original set of objects (central objects) is always available on the central server.

charts	Tables of information in JD Edwards EnterpriseOne that appear on forms in the software.
check-in repository	A repository for developers to check in and check out business service artifacts. There are multiple check-in repositories. Each can be used for a different purpose (for example, development, production, testing, and so on).
connector	Component-based interoperability model that enables third-party applications and JD Edwards EnterpriseOne to share logic and data. The JD Edwards EnterpriseOne connector architecture includes Java and COM connectors.
contra/clearing account	A general ledger account in JD Edwards EnterpriseOne Financial Management that is used by the system to offset (balance) journal entries. For example, you can use a contra/clearing account to balance the entries created by allocations in JD Edwards EnterpriseOne Financial Management.
Control Table Workbench	An application that, during the Installation Workbench processing, runs the batch applications for the planned merges that update the data dictionary, user-defined codes, menus, and user override tables.
control tables merge	A process that blends a customer's modifications to the control tables with the data that accompanies a new release.
correlation data	The data used to tie HTTP responses with requests that consist of business service name and method.
cost assignment	The process in JD Edwards EnterpriseOne Advanced Cost Accounting of tracing or allocating resources to activities or cost objects.
cost component	In JD Edwards EnterpriseOne Manufacturing, an element of an item's cost (for example, material, labor, or overhead).
credentials	A valid set of JD Edwards EnterpriseOne username/password/environment/role, EnterpriseOne session, or EnterpriseOne token.
cross-reference utility services	Utility services installed in a BPEL/ESB environment that access EnterpriseOne cross-reference data.
cross segment edit	A logic statement that establishes the relationship between configured item segments. Cross segment edits are used to prevent ordering of configurations that cannot be produced.
currency restatement	The process of converting amounts from one currency into another currency, generally for reporting purposes. You can use the currency restatement process, for example, when many currencies must be restated into a single currency for consolidated reporting.
cXML	A protocol used to facilitate communication between business documents and procurement applications, and between e-commerce hubs and suppliers.
database credentials	A valid database username/password.
database server	A server in a local area network that maintains a database and performs searches for client computers.
Data Source Workbench	An application that, during the Installation Workbench process, copies all data sources that are defined in the installation plan from the Data Source Master and Table and Data Source Sizing tables in the Planner data source to the system-release number data source. It also updates the Data Source Plan detail record to reflect completion.
date pattern	A calendar that represents the beginning date for the fiscal year and the ending date for each period in that year in standard and 52-period accounting.

denominated-in currency	The company currency in which financial reports are based.
deployment artifacts	Artifacts that are needed for the deployment process, such as servers, ports, and such.
deployment server	A server that is used to install, maintain, and distribute software to one or more enterprise servers and client workstations.
detail information	Information that relates to individual lines in JD Edwards EnterpriseOne transactions (for example, voucher pay items and sales order detail lines).
direct connect	A transaction method in which a client application communicates interactively and directly with a server application. See also batch-of-one immediate and store-and-forward.
Do Not Translate (DNT)	A type of data source that must exist on the iSeries because of BLOB restrictions.
dual pricing	The process of providing prices for goods and services in two currencies.
duplicate published business services authorization records	Two published business services authorization records with the same user identification information and published business services identification information.
embedded application server instance	An OC4J instance started by and running wholly within JDeveloper.
edit code	A code that indicates how a specific value for a report or a form should appear or be formatted. The default edit codes that pertain to reporting require particular attention because they account for a substantial amount of information.
edit mode	A condition of a form that enables users to change data.
edit rule	A method used for formatting and validating user entries against a predefined rule or set of rules.
Electronic Data Interchange (EDI)	An interoperability model that enables paperless computer-to-computer exchange of business transactions between JD Edwards EnterpriseOne and third-party systems. Companies that use EDI must have translator software to convert data from the EDI standard format to the formats of their computer systems.
embedded event rule	An event rule that is specific to a particular table or application. Examples include form-to-form calls, hiding a field based on a processing option value, and calling a business function. Contrast with the business function event rule.
Employee Work Center	A central location for sending and receiving all JD Edwards EnterpriseOne messages (system and user generated), regardless of the originating application or user. Each user has a mailbox that contains workflow and other messages, including Active Messages.
enterprise server	A server that contains the database and the logic for JD Edwards EnterpriseOne.
Enterprise Service Bus (ESB)	Middleware infrastructure products or technologies based on web services standards that enable a service-oriented architecture using an event-driven and XML-based messaging framework (the bus).
EnterpriseOne administrator	An actor responsible for the EnterpriseOne administration system.
EnterpriseOne credentials	A user ID, password, environment, and role used to validate a user of EnterpriseOne.
EnterpriseOne object	A reusable piece of code that is used to build applications. Object types include tables, forms, business functions, data dictionary items, batch processes, business views, event rules, versions, data structures, and media objects.

EnterpriseOne development client	Historically called “fat client,” a collection of installed EnterpriseOne components required to develop EnterpriseOne artifacts, including the Microsoft Windows client and design tools.
EnterpriseOne extension	A JDeveloper component (plug-in) specific to EnterpriseOne. A JDeveloper wizard is a specific example of an extension.
EnterpriseOne process	A software process that enables JD Edwards EnterpriseOne clients and servers to handle processing requests and run transactions. A client runs one process, and servers can have multiple instances of a process. JD Edwards EnterpriseOne processes can also be dedicated to specific tasks (for example, workflow messages and data replication) to ensure that critical processes don’t have to wait if the server is particularly busy.
EnterpriseOne resource	Any EnterpriseOne table, metadata, business function, dictionary information, or other information restricted to authorized users.
Environment Workbench	An application that, during the Installation Workbench process, copies the environment information and Object Configuration Manager tables for each environment from the Planner data source to the system-release number data source. It also updates the Environment Plan detail record to reflect completion.
escalation monitor	A batch process that monitors pending requests or activities and restarts or forwards them to the next step or user after they have been inactive for a specified amount of time.
event rule	A logic statement that instructs the system to perform one or more operations based on an activity that can occur in a specific application, such as entering a form or exiting a field.
explicit transaction	Transaction used by a business service developer to explicitly control the type (auto or manual) and the scope of transaction boundaries within a business service.
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.
facility	An entity within a business for which you want to track costs. For example, a facility might be a warehouse location, job, project, work center, or branch/plant. A facility is sometimes referred to as a “business unit.”
fast path	A command prompt that enables the user to move quickly among menus and applications by using specific commands.
file server	A server that stores files to be accessed by other computers on the network. Unlike a disk server, which appears to the user as a remote disk drive, a file server is a sophisticated device that not only stores files, but also manages them and maintains order as network users request files and make changes to these files.
final mode	The report processing mode of a processing mode of a program that updates or creates data records.
foundation	A framework that must be accessible for execution of business services at runtime. This includes, but is not limited to, the Java Connector and JDBj.
FTP server	A server that responds to requests for files via file transfer protocol.
header information	Information at the beginning of a table or form. Header information is used to identify or provide control information for the group of records that follows.
HTTP Adapter	A generic set of services that are used to do the basic HTTP operations, such as GET, POST, PUT, DELETE, TRACE, HEAD, and OPTIONS with the provided URL.

instantiate	A Java term meaning "to create." When a class is instantiated, a new instance is created.
integration developer	The user of the system who develops, runs, and debugs the EnterpriseOne business services. The integration developer uses the EnterpriseOne business services to develop these components.
integration point (IP)	The business logic in previous implementations of EnterpriseOne that exposes a document level interface. This type of logic used to be called XBPs. In EnterpriseOne 8.11, IPs are implemented in Web Services Gateway powered by webMethods.
integration server	A server that facilitates interaction between diverse operating systems and applications across internal and external networked computer systems.
integrity test	A process used to supplement a company's internal balancing procedures by locating and reporting balancing problems and data inconsistencies.
interface table	See Z table.
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.
interoperability model	A method for third-party systems to connect to or access JD Edwards EnterpriseOne.
in-your-face-error	In JD Edwards EnterpriseOne, a form-level property which, when enabled, causes the text of application errors to appear on the form.
IServer service	This internet server service resides on the web server and is used to speed up delivery of the Java class files from the database to the client.
jargon	An alternative data dictionary item description that JD Edwards EnterpriseOne appears based on the product code of the current object.
Java application server	A component-based server that resides in the middle-tier of a server-centric architecture. This server provides middleware services for security and state maintenance, along with data access and persistence.
JDBNET	A database driver that enables heterogeneous servers to access each other's data.
JDEBASE Database Middleware	A JD Edwards EnterpriseOne proprietary database middleware package that provides platform-independent APIs, along with client-to-server access.
JDECallObject	An API used by business functions to invoke other business functions.
jde.ini	A JD Edwards EnterpriseOne file (or member for iSeries) that provides the runtime settings required for JD Edwards EnterpriseOne initialization. Specific versions of the file or member must reside on every machine running JD Edwards EnterpriseOne. This includes workstations and servers.
JDEIPC	Communications programming tools used by server code to regulate access to the same data in multiprocess environments, communicate and coordinate between processes, and create new processes.
jde.log	The main diagnostic log file of JD Edwards EnterpriseOne. This file is always located in the root directory on the primary drive and contains status and error messages from the startup and operation of JD Edwards EnterpriseOne.
JDENET	A JD Edwards EnterpriseOne proprietary communications middleware package. This package is a peer-to-peer, message-based, socket-based, multiprocess communications middleware solution. It handles client-to-server and server-to-server communications for all JD Edwards EnterpriseOne supported platforms.
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.
JMS Queue	A Java Messaging service queue used for point-to-point messaging.
listener service	A listener that listens for XML messages over HTTP.
local repository	A developer's local development environment that is used to store business service artifacts.
local standalone BPEL/ESB server	A standalone BPEL/ESB server that is not installed within an application server.
Location Workbench	An application that, during the Installation Workbench process, copies all locations that are defined in the installation plan from the Location Master table in the Planner data source to the system data source.
logic server	A server in a distributed network that provides the business logic for an application program. In a typical configuration, pristine objects are replicated on to the logic server from the central server. The logic server, in conjunction with workstations, actually performs the processing required when JD Edwards EnterpriseOne software runs.
MailMerge Workbench	An application that merges Microsoft Word 6.0 (or higher) word-processing documents with JD Edwards EnterpriseOne records to automatically print business documents. You can use MailMerge Workbench to print documents, such as form letters about verification of employment.
Manual Commit transaction	A database connection where all database operations delay writing to the database until a call to commit is made.
master business function (MBF)	An interactive master file that serves as a central location for adding, changing, and updating information in a database. Master business functions pass information between data entry forms and the appropriate tables. These master functions provide a common set of functions that contain all of the necessary default and editing rules for related programs. MBFs contain logic that ensures the integrity of adding, updating, and deleting information from databases.
master table	See published table.
matching document	A document associated with an original document to complete or change a transaction. For example, in JD Edwards EnterpriseOne Financial Management, a receipt is the matching document of an invoice, and a payment is the matching document of a voucher.
media storage object	Files that use one of the following naming conventions that are not organized into table format: Gxxx, xxxGT, or GTxxx.
message center	A central location for sending and receiving all JD Edwards EnterpriseOne messages (system and user generated), regardless of the originating application or user.
messaging adapter	An interoperability model that enables third-party systems to connect to JD Edwards EnterpriseOne to exchange information through the use of messaging queues.
messaging server	A server that handles messages that are sent for use by other programs using a messaging API. Messaging servers typically employ a middleware program to perform their functions.
Middle-Tier BPEL/ESB Server	A BPEL/ESB server that is installed within an application server.
Monitoring Application	An EnterpriseOne tool provided for an administrator to get statistical information for various EntepriseOne servers, reset statistics, and set notifications.

named event rule (NER)	Encapsulated, reusable business logic created using event rules, rather that C programming. NERs are also called business function event rules. NERs can be reused in multiple places by multiple programs. This modularity lends itself to streamlining, reusability of code, and less work.
nota fiscal	In Brazil, a legal document that must accompany all commercial transactions for tax purposes and that must contain information required by tax regulations.
nota fiscal factura	In Brazil, a nota fiscal with invoice information. See also *nota fiscal.*
Object Configuration Manager (OCM)	In JD Edwards EnterpriseOne, the object request broker and control center for the runtime environment. OCM keeps track of the runtime locations for business functions, data, and batch applications. When one of these objects is called, OCM directs access to it using defaults and overrides for a given environment and user.
Object Librarian	A repository of all versions, applications, and business functions reusable in building applications. Object Librarian provides check-out and check-in capabilities for developers, and it controls the creation, modification, and use of JD Edwards EnterpriseOne objects. Object Librarian supports multiple environments (such as production and development) and enables objects to be easily moved from one environment to another.
Object Librarian merge	A process that blends any modifications to the Object Librarian in a previous release into the Object Librarian in a new release.
Open Data Access (ODA)	An interoperability model that enables you to use SQL statements to extract JD Edwards EnterpriseOne data for summarization and report generation.
Output Stream Access (OSA)	An interoperability model that enables you to set up an interface for JD Edwards EnterpriseOne to pass data to another software package, such as Microsoft Excel, for processing.
package	JD Edwards EnterpriseOne objects are installed to workstations in packages from the deployment server. A package can be compared to a bill of material or kit that indicates the necessary objects for that workstation and where on the deployment server the installation program can find them. It is point-in-time snapshot of the central objects on the deployment server.
package build	A software application that facilitates the deployment of software changes and new applications to existing users. Additionally, in JD Edwards EnterpriseOne, a package build can be a compiled version of the software. When you upgrade your version of the ERP software, for example, you are said to take a package build. Consider the following context: "Also, do not transfer business functions into the production path code until you are ready to deploy, because a global build of business functions done during a package build will automatically include the new functions." The process of creating a package build is often referred to, as it is in this example, simply as "a package build."
package location	The directory structure location for the package and its set of replicated objects. This is usually \\deployment server\release\path_code\package\package name. The subdirectories under this path are where the replicated objects for the package are placed. This is also referred to as where the package is built or stored.
Package Workbench	An application that, during the Installation Workbench process, transfers the package information tables from the Planner data source to the system-release number data source. It also updates the Package Plan detail record to reflect completion.
Pathcode Directory	The specific portion of the file system on the EnterpriseOne development client where EnterpriseOne development artifacts are stored.

patterns	General repeatable solutions to a commonly occurring problem in software design. For business service development, the focus is on the object relationships and interactions. For orchestrations, the focus is on the integration patterns (for example, synchronous and asynchronous request/response, publish, notify, and receive/reply).
planning family	A means of grouping end items whose similarity of design and manufacture facilitates being planned in aggregate.
preference profile	The ability to define default values for specified fields for a user-defined hierarchy of items, item groups, customers, and customer groups.
print server	The interface between a printer and a network that enables network clients to connect to the printer and send their print jobs to it. A print server can be a computer, separate hardware device, or even hardware that resides inside of the printer itself.
pristine environment	A JD Edwards EnterpriseOne environment used to test unaltered objects with JD Edwards EnterpriseOne demonstration data or for training classes. You must have this environment so that you can compare pristine objects that you modify.
processing option	A data structure that enables users to supply parameters that regulate the running of a batch program or report. For example, you can use processing options to specify default values for certain fields, to determine how information appears or is printed, to specify date ranges, to supply runtime values that regulate program execution, and so on.
production environment	A JD Edwards EnterpriseOne environment in which users operate EnterpriseOne software.
production-grade file server	A file server that has been quality assurance tested and commercialized and that is usually provided in conjunction with user support services.
Production Published Business Services Web Service	Published business services web service deployed to a production application server.
program temporary fix (PTF)	A representation of changes to JD Edwards EnterpriseOne software that your organization receives on magnetic tapes or disks.
project	In JD Edwards EnterpriseOne, a virtual container for objects being developed in Object Management Workbench.
promotion path	The designated path for advancing objects or projects in a workflow. The following is the normal promotion cycle (path): 11>21>26>28>38>01 In this path, *11* equals new project pending review, *21* equals programming, *26* equals QA test/review, *28* equals QA test/review complete, *38* equals in production, *01* equals complete. During the normal project promotion cycle, developers check objects out of and into the development path code and then promote them to the prototype path code. The objects are then moved to the productions path code before declaring them complete.
proxy server	A server that acts as a barrier between a workstation and the internet so that the enterprise can ensure security, administrative control, and caching service.
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.
published business service identification information	Information about a published business service used to determine relevant authorization records. Published business services + method name, published business services, or *ALL.

published business service web service	Published business services components packaged as J2EE Web Service (namely, a J2EE EAR file that contains business service classes, business service foundation, configuration files, and web service artifacts).
published table	Also called a master table, this is the central copy to be replicated to other machines. Residing on the publisher machine, the F98DRPUB table identifies all of the published tables and their associated publishers in the enterprise.
publisher	The server that is responsible for the published table. The F98DRPUB table identifies all of the published tables and their associated publishers in the enterprise.
pull replication	One of the JD Edwards EnterpriseOne methods for replicating data to individual workstations. Such machines are set up as pull subscribers using JD Edwards EnterpriseOne data replication tools. The only time that pull subscribers are notified of changes, updates, and deletions is when they request such information. The request is in the form of a message that is sent, usually at startup, from the pull subscriber to the server machine that stores the F98DRPCN table.
QBE	An abbreviation for *query by example*. In JD Edwards EnterpriseOne, the QBE line is the top line on a detail area that is used for filtering data.
real-time event	A message triggered from EnterpriseOne application logic that is intended for external systems to consume.
refresh	A function used to modify JD Edwards EnterpriseOne software, or subset of it, such as a table or business data, so that it functions at a new release or cumulative update level, such as B73.2 or B73.2.1.
replication server	A server that is responsible for replicating central objects to client machines.
Rt-Addressing	Unique data identifying a browser session that initiates the business services call request host/port user session.
rules	Mandatory guidelines that are not enforced by tooling, but must be followed in order to accomplish the desired results and to meet specified standards.
quote order	In JD Edwards Procurement and Subcontract Management, a request from a supplier for item and price information from which you can create a purchase order.
	In JD Edwards Sales Order Management, item and price information for a customer who has not yet committed to a sales order.
secure by default	A security model that assumes that a user does not have permission to execute an object unless there is a specific record indicating such permissions.
Secure Socket Layer (SSL)	A security protocol that provides communication privacy. SSL enables client and server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery.
SEI implementation	A Java class that implements the methods that declare in a Service Endpoint Interface (SEI).
selection	Found on JD Edwards EnterpriseOne menus, a selection represents functions that you can access from a menu. To make a selection, type the associated number in the Selection field and press Enter.
serialize	The process of converting an object or data into a format for storage or transmission across a network connection link with the ability to reconstruct the original data or objects when needed.
Server Workbench	An application that, during the Installation Workbench process, copies the server configuration files from the Planner data source to the system-release number

	data source. The application also updates the Server Plan detail record to reflect completion.
Service Endpoint Interface (SEI)	A Java interface that declares the methods that a client can invoke on the service.
SOA	Abbreviation for *Service Oriented Architecture*.
softcoding	A coding technique that enables an administrator to manipulate site-specific variables that affect the execution of a given process.
source repository	A repository for HTTP adapter and listener service development environment artifacts.
spot rate	An exchange rate entered at the transaction level. This rate overrides the exchange rate that is set up between two currencies.
Specification merge	A merge that comprises three merges: Object Librarian merge, Versions List merge, and Central Objects merge. The merges blend customer modifications with data that accompanies a new release.
specification	A complete description of a JD Edwards EnterpriseOne object. Each object has its own specification, or name, which is used to build applications.
Specification Table Merge Workbench	An application that, during the Installation Workbench process, runs the batch applications that update the specification tables.
SSL Certificate	A special message signed by a certificate authority that contains the name of a user and that user's public key in such a way that anyone can "verify" that the message was signed by no one other than the certification authority and thereby develop trust in the user's public key.
store-and-forward	The mode of processing that enables users who are disconnected from a server to enter transactions and then later connect to the server to upload those transactions.
subscriber table	Table F98DRSUB, which is stored on the publisher server with the F98DRPUB table and identifies all of the subscriber machines for each published table.
superclass	An inheritance concept of the Java language where a class is an instance of something, but is also more specific. "Tree" might be the superclass of "Oak" and "Elm," for example.
supplemental data	Any type of information that is not maintained in a master file. Supplemental data is usually additional information about employees, applicants, requisitions, and jobs (such as an employee's job skills, degrees, or foreign languages spoken). You can track virtually any type of information that your organization needs. For example, in addition to the data in the standard master tables (the Address Book Master, Customer Master, and Supplier Master tables), you can maintain other kinds of data in separate, generic databases. These generic databases enable a standard approach to entering and maintaining supplemental data across JD Edwards EnterpriseOne systems.
table access management (TAM)	The JD Edwards EnterpriseOne component that handles the storage and retrieval of use-defined data. TAM stores information, such as data dictionary definitions; application and report specifications; event rules; table definitions; business function input parameters and library information; and data structure definitions for running applications, reports, and business functions.
Table Conversion Workbench	An interoperability model that enables the exchange of information between JD Edwards EnterpriseOne and third-party systems using non-JD Edwards EnterpriseOne tables.

table conversion	An interoperability model that enables the exchange of information between JD Edwards EnterpriseOne and third-party systems using non-JD Edwards EnterpriseOne tables.
table event rules	Logic that is attached to database triggers that runs whenever the action specified by the trigger occurs against the table. Although JD Edwards EnterpriseOne enables event rules to be attached to application events, this functionality is application specific. Table event rules provide embedded logic at the table level.
terminal server	A server that enables terminals, microcomputers, and other devices to connect to a network or host computer or to devices attached to that particular computer.
three-tier processing	The task of entering, reviewing and approving, and posting batches of transactions in JD Edwards EnterpriseOne.
three-way voucher match	In JD Edwards Procurement and Subcontract Management, the process of comparing receipt information to supplier's invoices to create vouchers. In a three-way match, you use the receipt records to create vouchers.
transaction processing (TP) monitor	A monitor that controls data transfer between local and remote terminals and the applications that originated them. TP monitors also protect data integrity in the distributed environment and may include programs that validate data and format terminal screens.
transaction processing method	A method related to the management of a manual commit transaction boundary (for example, start, commit, rollback, and cancel).
transaction set	An electronic business transaction (electronic data interchange standard document) made up of segments.
trigger	One of several events specific to data dictionary items. You can attach logic to a data dictionary item that the system processes automatically when the event occurs.
triggering event	A specific workflow event that requires special action or has defined consequences or resulting actions.
two-way authentication	An authentication mechanism in which both client and server authenticate themselves by providing the SSL certificates to each other.
two-way voucher match	In JD Edwards Procurement and Subcontract Management, the process of comparing purchase order detail lines to the suppliers' invoices to create vouchers. You do not record receipt information.
user identification information	User ID, role, or *public.
User Overrides merge	Adds new user override records into a customer's user override table.
value object	A specific type of source file that holds input or output data, much like a data structure passes data. Value objects can be exposed (used in a published business service) or internal, and input or output. They are comprised of simple and complex elements and accessories to those elements.
variance	In JD Edwards Capital Asset Management, the difference between revenue generated by a piece of equipment and costs incurred by the equipment. In JD Edwards EnterpriseOne Project Costing and JD Edwards EnterpriseOne Manufacturing, the difference between two methods of costing the same item (for example, the difference between the frozen standard cost and the current cost is an engineering variance). Frozen standard costs come from the Cost Components table, and the current costs are calculated using the current bill of material, routing, and overhead rates.

versioning a published business service	Adding additional functionality/interfaces to the published business services without modifying the existing functionality/interfaces.
Version List merge	The Versions List merge preserves any non-XJDE and non-ZJDE version specifications for objects that are valid in the new release, as well as their processing options data.
visual assist	Forms that can be invoked from a control via a trigger to assist the user in determining what data belongs in the control.
vocabulary override	An alternate description for a data dictionary item that appears on a specific JD Edwards EnterpriseOne form or report.
wchar_t	An internal type of a wide character. It is used for writing portable programs for international markets.
web application server	A web server that enables web applications to exchange data with the back-end systems and databases used in eBusiness transactions.
web server	A server that sends information as requested by a browser, using the TCP/IP set of protocols. A web server can do more than just coordination of requests from browsers; it can do anything a normal server can do, such as house applications or data. Any computer can be turned into a web server by installing server software and connecting the machine to the internet.
Web Service Description Language (WSDL)	An XML format for describing network services.
Web Service Inspection Language (WSIL)	An XML format for assisting in the inspection of a site for available services and a set of rules for how inspection-related information should be made.
web service proxy foundation	Foundation classes for web service proxy that must be included in a business service server artifact for web service consumption on WAS.
web service softcoding record	An XML document that contains values that are used to configure a web service proxy. This document identifies the endpoint and conditionally includes security information.
web service softcoding template	An XML document that provides the structure for a soft coded record.
Where clause	The portion of a database operation that specifies which records the database operation will affect.
Windows terminal server	A multiuser server that enables terminals and minimally configured computers to display Windows applications even if they are not capable of running Windows software themselves. All client processing is performed centrally at the Windows terminal server and only display, keystroke, and mouse commands are transmitted over the network to the client terminal device.
wizard	A type of JDeveloper extension used to walk the user through a series of steps.
workbench	A program that enables users to access a group of related programs from a single entry point. Typically, the programs that you access from a workbench are used to complete a large business process. For example, you use the JD Edwards EnterpriseOne Payroll Cycle Workbench (P07210) to access all of the programs that the system uses to process payroll, print payments, create payroll reports, create journal entries, and update payroll history. Examples of JD Edwards EnterpriseOne workbenches include Service Management Workbench (P90CD020), Line Scheduling Workbench (P3153), Planning Workbench (P13700), Auditor's Workbench (P09E115), and Payroll Cycle Workbench.
work day calendar	In JD Edwards EnterpriseOne Manufacturing, a calendar that is used in planning functions that consecutively lists only working days so that component and work order scheduling can be done based on the actual number of work days available. A work

	day calendar is sometimes referred to as planning calendar, manufacturing calendar, or shop floor calendar.
workflow	The automation of a business process, in whole or in part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules.
workgroup server	A server that usually contains subsets of data replicated from a master network server. A workgroup server does not perform application or batch processing.
XAPI events	A service that uses system calls to capture JD Edwards EnterpriseOne transactions as they occur and then calls third-party software, end users, and other JD Edwards EnterpriseOne systems that have requested notification when the specified transactions occur to return a response.
XML CallObject	An interoperability capability that enables you to call business functions.
XML Dispatch	An interoperability capability that provides a single point of entry for all XML documents coming into JD Edwards EnterpriseOne for responses.
XML List	An interoperability capability that enables you to request and receive JD Edwards EnterpriseOne database information in chunks.
XML Service	An interoperability capability that enables you to request events from one JD Edwards EnterpriseOne system and receive a response from another JD Edwards EnterpriseOne system.
XML Transaction	An interoperability capability that enables you to use a predefined transaction type to send information to or request information from JD Edwards EnterpriseOne. XML transaction uses interface table functionality.
XML Transaction Service (XTS)	Transforms an XML document that is not in the JD Edwards EnterpriseOne format into an XML document that can be processed by JD Edwards EnterpriseOne. XTS then transforms the response back to the request originator XML format.
Z event	A service that uses interface table functionality to capture JD Edwards EnterpriseOne transactions and provide notification to third-party software, end users, and other JD Edwards EnterpriseOne systems that have requested to be notified when certain transactions occur.
Z table	A working table where non-JD Edwards EnterpriseOne information can be stored and then processed into JD Edwards EnterpriseOne. Z tables also can be used to retrieve JD Edwards EnterpriseOne data. Z tables are also known as interface tables.
Z transaction	Third-party data that is properly formatted in interface tables for updating to the JD Edwards EnterpriseOne database.

Index