Oracle® Retail POS Suite

Implementation Guide, Volume 2 – Extension Solutions

Release 14.1

E54476-02

September 2015

ORACLE®

Oracle Retail POS Suite Implementation Guide, Volume 2 - Extension Solutions, Release 14.1

E54476-02

Primary Author: Bernadette Goodman

reconfigurations, reassembly or reverse assembly, re-engineering or reverse engineering and recompilations or reverse compilations of the VAR Applications or any derivatives of the VAR Applications. You acknowledge that it shall be a breach of the agreement to utilize the relationship, and/or confidential information of the VAR Applications for purposes of competitive discovery.

The VAR Applications contain trade secrets of Oracle and Oracle's licensors and Customer shall not attempt, cause, or permit the alteration, decompilation, reverse engineering, disassembly or other reduction of the VAR Applications to a human perceivable form. Oracle reserves the right to replace, with functional equivalent software, any of the VAR Applications in future releases of the applicable program.

Contents

4 Point-of-Service Development Standards

7 Returns Management Extension Guidelines

8 Coding Your First Back Office or Central Office Feature

9 Frameworks

10 Manager/Technician Framework

11 User Interface Framework

12 Oracle Retail Tour Framework

13 Point-of-Service COMMEXT Framework

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Retail POS Suite Implementation Guide – Volume 2, Extension Solutions, Release 14.1

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document.

Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

> **Note:** Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the new Applications Release Online Documentation CD available on My Oracle Support and www.oracle.com. It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: retail-doc_us@oracle.com

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our web site at www.oracle.com.

Preface

Oracle Retail POS Suite Extension Solutions contain the requirements and procedures that are necessary for the retailer to extend Back Office, Central Office, Returns Management, and Point-of-Service.

Audience

The audience for this document is developers who develop code for Oracle Retail Back Office, Central Office, Returns Management, and Point-of-Service. Knowledge of the following techniques is required:

- Java Programming Language
- Object-Oriented Design Methodology (OOD)
- Extensible Markup Language (XML)

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the following Release 14.1 documentation sets:

- Oracle Retail Back Office documentation set
- Oracle Retail Central Office documentation set
- Oracle Retail Point-of-Service documentation set
- Oracle Retail Returns Management documentation set

Customer Support

To contact Oracle Customer Support, access My Oracle Support at the following URL:

https://support.oracle.com

When contacting Customer Support, please provide the following:

- Product version and program/module name
- Functional and technical description of the problem (include business impact)
- Detailed step-by-step instructions to re-create
- Exact error message received
- Screen shots of each step you take

Review Patch Documentation

When you install the application for the first time, you install either a base release (for example, 14.1) or a later patch release (for example, 14.1.1). If you are installing the base release or additional patch releases, read the documentation for all releases that have occurred since the base release before you begin installation. Documentation for patch releases can contain critical information related to the base release, as well as information about code changes since the base release.

Improved Process for Oracle Retail Documentation Corrections

To more quickly address critical corrections to Oracle Retail documentation content, Oracle Retail documentation may be republished whenever a critical correction is needed. For critical corrections, the republication of an Oracle Retail document may at times not be attached to a numbered software release; instead, the Oracle Retail document will simply be replaced on the Oracle Technology Network Web site, or, in the case of Data Models, to the applicable My Oracle Support Documentation container where they reside.

This process will prevent delays in making critical corrections available to customers. For the customer, it means that before you begin installation, you must verify that you have the most recent version of the Oracle Retail documentation set. Oracle Retail documentation is available on the Oracle Technology Network at the following URL:

http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html

An updated version of the applicable Oracle Retail document is indicated by Oracle part number, as well as print date (month and year). An updated version uses the same part number, with a higher-numbered suffix. For example, part number E123456-02 is an updated version of a document with part number E123456-01.

If a more recent version of a document is available, that version supersedes all previous versions.

Oracle Retail Documentation on the Oracle Technology Network

Documentation is packaged with each Oracle Retail product release. Oracle Retail product documentation is also available on the following web site:

http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html

(Data Model documents are not available through Oracle Technology Network. These documents are packaged with released code, or you can obtain them through My Oracle Support.)

Documentation should be available on this web site within a month after a product release.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Extracting Source Code

Much of this guide deals with the structure and function of Oracle Retail Back Office code, and how you can modify and extend it to serve the needs of your organization.

The source code is downloadable in a zip file.

This zip file contain the following:

File Name	Comments
cmnotes.txt	Configuration Management notes. Describe how to set up and build the source.
<application>-<*release_number*>_source.zip	The application source, where <application> is: ■ ORBO (Back Office) ■ ORCO (Central Office) ■ ORPOS (Point-of-Service) ■ ORRM (Returns Management)
ORSSS-<*release_number*>_data_model.zip	Data Model (database schema) documentation.
README.html	Release Notes.

Using pkzip, WinZip or similar utilities, you can extract <application>-<*release_number*> onto your local hard disk. Choose the option to preserve the directory structure when you extract. All the source files are placed into the following directory:

```
<Path to disk root>/<application>-<release_number>_source
```

From this point on, this directory is referred to as:

- `<BO_SRC_ROOT>` for Back Office.
- `<CO_SRC_ROOT>` for Central Office.
- `<POS_SRC_ROOT>` for Point-of-Service.
- `<RM_SRC_ROOT>` for Returns Management.

The following is the first-level directory structure under the directory:

Directory	Comments
applications	Contains application-specific code for applications.
build	Files used to compile, assemble and run functional tests.
clientinterfaces	Interface definitions, between different code modules.

Directory	Comments
commerceservices	Commerce Services code.
DIMP	Includes XSD and XML files used to map data import.
installer	Files used by the installer.
modules	A collection of various code modules some of which are the foundation for Commerce Services. The utility module contains SQL files used for database creation and pre-loading.
thirdparty	Executable (mostly .jar files) from third-party providers.
webapp	Web-based user interface code. Also contains the Application Managers.

In subsequent chapters, all path names of a code file are made relative to one of these directories. You must add the directory name (for example, <source_directory>) to the file path to get its actual location on disk.

2

Development Environments

This chapter describes how to set up development environments.

Back Office and Central Office Development Environment

This chapter describes how to set up a single-user development environment for Oracle Retail Back Office or Central Office. The setup enumerates the files, tools, and resources necessary to build and run the Back Office or Central Office application.

When you complete the steps in this chapter, you will have a local development workspace with the ability to build the application, and an application server installation to which you can deploy the Back Office or Central Office application.

This chapter assumes that you are using WebLogic Application Server and Oracle database; together, they form the officially supported platform for the current release of Back Office or Central Office.

Your development environment may use different tools, and you may develop variations on this procedure. Specific property file settings, in particular, may need to be modified in your environment.

For more information about product versions, see the *Oracle Retail Back Office Installation Guide* or *Oracle Retail Central Office Installation Guide*.

Using the Apache Ant Build Tool

Oracle Retail uses the Apache Ant build tool to compile and build executable products from source. Ant uses build information defined in various build.xml files, which in turn read from properties files. Each top-level directory in the product's source contains a build.xml file that specifies a variety of targets, or build tasks, for use by Ant.

Since each code module depends on other modules, the top-level build directory has a build.xml file which contains targets designed to build the entire system. You can build modules individually if you built them in the correct dependency order.

Properties files (such as build.properties) contain values that are used by Ant when it processes tasks. Individual properties can exist in multiple files. The first setting processed by Ant is the one that is used; properties are like constants which cannot be changed once set.

If your system does not already have Ant, you can use the version shipped with Back Office or Central Office located at:

```
<source_directory>\thirdparty\apache\ant
```

Note: Make sure that the Ant bin directory is included in your workstation's PATH.

Prerequisites for the Development Environment

The following software resources must be installed and configured before you set up the Back Office or Central Office development environment as described in the next section. Where a software version is specified, use only the specified version.

- The Back Office or Central Office source code, on a local (or network) hard disk. See Chapter 1 for details on how to extract the code.
- A database server and database. You should have access to the database server; you need its connection URL, user name and password. Depending on your organization's preferences, you may need to install the database server yourself, get a qualified database administrator to install it for you, or you may access a database server installed on another machine. The instructions in this chapter work for a local or remote database.
- JDK 7 or later in the Java 7 code line.

  Downloads and instructions are available at the following web site:

  `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

  The JAVA_HOME environment variable needs to be set in your operating system and the `%JAVA_HOME%\bin` directory needs to be added to the path.

Install Weblogic Application Server

Install Weblogic Application Server (WL) under any directory you choose. Follow the instructions that come with the Application Server product. This chapter refers to this directory as <WL_ROOT>.

Build the Back Office or Central Office Application

Do the following to build the Back Office or Central Office application:

1. CD to the Back Office or Central Office build directory.
2. Edit setenv.sh (or setenv.bat on Windows). Make sure that ANT_HOME is set correctly for your system.
3. Execute setenv.sh (or setenv.bat).
4. Run the following, where `<application>` is either `backoffice` or `centraloffice`:

   ```
   ant -Denv=<application> clean.build.assemble
   ```

   This command will take several minutes to execute. If successful, it puts the J2EE-compatible .ear file in `applications/<application>/assemble/assemble.working.dir/<application>.ear`, where `<application>` is either `backoffice` or `centraloffice`.
5. Obtain the installer in `install/dist/<application>-<`*release_number*`>.zip`, where `<application>` is either `ORBO` or `ORCO`.
6. Run the installer. See the *Oracle Retail Back Office Installation Guide* or the *Oracle Retail Central Office Installation Guide* for more information.

Point-of-Service Development Environment

A development environment for Point-of-Service includes all files, tools and resources necessary to build and run the Point-of-Service application. While development environments may vary depending on the choice of IDE, database, and version control system, configuration of the development environment involves some common steps. This document addresses components that various development environments have in common.

Preparation

The following software resources must be installed and configured before the Point-of-Service development environment can be set up. Ensure that the following are in place:

Version control system

The Point-of-Service source code must be available from a source control system.

OracleRetailStore database

The OracleRetailStore database should be installed.

Eclipse version 3.0 or another IDE

If installing Eclipse, downloads and instructions are available from http://www.eclipse.org/downloads/.

JDK 7 or later in the Java 7 code line

Downloads and instructions are available at the following web site:

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Setup

Setting up the development environment requires installing the Point-of-Service application, populating the database, creating a sandbox, configuring the IDE, and configuring the version control system.

Install Point-of-Service

Install Point-of-Service using the installation script. While running the Point-of-Service installation script, accept the default options even when nothing is selected, except for the options discussed in the following table.

Table 2–1 lists some Point-of-Service installation options.

Table 2–1 Point-of-Service Installation Options

Option	Instruction
Tier Type	Choose the Tier Type from the following options. Client and Store Server—Choose both of these options to run client and server components on the same machine in separate JVMs.
Database Information	Specify the database type and its location.
JRE Location	Location where the JRE is installed.

Build the Database

Open a command prompt in the Point-of-Service installer directory and use the following command-lines:

- To reset the store database: `install.cmd ant install-database`
- To reset the scratchpad database: `install.cmd ant install-scratchpad`
- To reset both: `install.cmd ant install-database install-scratchpad`

The install-database command uses the settings in the ant.install.properties file, so the dataset specified by the input.install.database property is loaded. The values can be:

- no – no action taken
- schema – only install the schema, no data
- minimum – schema and minimum required data
- sample – schema, minimum, and sample data

> **Note:** If the same installer directory is used for installing the Point-of-Service client, the ant.install.properties file is overwritten with the new settings. Then the ant.install.properties file cannot be used for building the database.

To reset the scratchpad database, the ant.install.properties file needs to have the scratchpad database information as well as input.install.scratchpad.database set to true.

Create a Sandbox

If you plan to retrieve all the source code with the version control system, create a local sandbox with only one directory such as the following:

```
C:\mySandbox\
```

Configure the Version Control System

Each file from the source code repository should be retrieved to the proper location in your sandbox. To do this, set the workfile location of the root of each of the product components displayed in the version control system. Each workfile location should be set to the local sandbox. For example, if your sandbox is named C:\mySandbox, the root of the product components should point to C:\mySandbox.

Run Point-of-Service

To verify the setup, run the Point-of-Service application using the following steps:

1. Start the OracleRetailStore database.
2. Build the project.
3. Run Point-of-Service from the IDE.

3

General Development Standards

The following standards have been adopted by Oracle Retail product and service development teams. These standards are intended to reduce bugs and increase the quality of the code. The chapter covers basic standards, architectural issues, and common frameworks. These guidelines apply to all Oracle Retail applications.

Note: See the *Oracle Retail POS Suite Security Guide* for more information about specific security features and implementation guidelines for the POS Suite products.

Basics

The guidelines in this section cover common coding issues and standards.

Java Recommendations

The following are guidelines for what to avoid when writing Java code.

- Do use polymorphism.
- Do have only one return statement per function or method; make it the last statement.
- Do use constants instead of literal values when possible.
- Do import only the classes necessary instead of using wildcards.
- Do define constants at the top of the class instead of inside a method.
- Do keep methods small, so that they can be viewed on a single screen without scrolling.
- Do not have an empty catch block. This destroys an exception from further down the line that might include information necessary for debugging.
- Do not concatenate strings. Oracle Retail products tend to be string-intensive and string concatenation is an expensive operation. Use StringBuilder instead.
- Do not use function calls inside looping conditionals (for example, while (i <=name.len())). This calls the function with each iteration of the loop and can affect performance.
- Do not use a static array of strings.
- Do not use public attributes.
- Do not use a switch to make a call based on the object type.

Avoiding Common Java Bugs

The following fatal Java bugs are not found at compile time and are not easily found at runtime. These bugs can be avoided by following the recommendations in the following table.

Table 3–1 lists some fatal Java bugs and their preventative measures.

Table 3–1 Common Java Bugs

Bug	Preventative Measure
null pointer exception	Check for null before using an object returned by another method.
boundary checking	Check the validity of values returned by other methods before using them.
array index out of bounds	When using a value as a subscript to access an array element directly, first verify that the value is within the bounds of the array.
incorrect cast	When casting an object, use instanceof to ensure that the object is of that type before attempting the cast.

Formatting

Follow these formatting standards to ensure consistency with existing code.

- Indenting/braces—Indent all code blocks with four spaces (not tabs). Put the opening brace on its own line following the control statement and in the same column. Statements within the block are indented. Closing brace is on its own line and in same column as the opening brace. Follow control statements (if, while, and so on) with a code block with braces, even when the code block is only one line long.
- Line wrapping—If line breaks are in a parameter list, line up the beginning of the second line with the first parameter on the first line. Lines should not exceed 120 characters.
- Spacing—Include a space on both sides of binary operators. Do not use a space with unary operators. Do not use spaces around parenthesis. Include a blank line before a code block.
- Deprecation—Whenever you deprecate a method or class from an existing release is deprecated, mark it as deprecated, noting the release in which it was deprecated, and what methods or classes should be used in place of the deprecated items; these records facilitate later code cleanup.
- Header—The file header should include the tag for revision and log history.

Example 3–1 Header Sample

```
/* ===========================================================================
*    Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
 * ===========================================================================
 * $Header:$
 * ===========================================================================
 * NOTES
 * <other useful comments, qualifications, etc.>
 *
 * MODIFIED    (MM/DD/YY)
 *    username  01/04/10 - update header date
 *
 * ===========================================================================
package oracle.retail.stores.samples;
```

```
// Import only what is used and organize from lowest layer to highest.
import oracle.retail.stores.common.utility.Util;

//---------------------------------------------------------------------------
/**
     This class is a sample class. Its purpose is to illustrate proper
     formatting.
     @version $Revision$
**/
//---------------------------------------------------------------------------
public class Sample extends AbstractSample
implements SampleIfc
{
    // revision number supplied by configuration management tool
    public static String revisionNumber = "$Revision$";
    // This is a sample data member.
    // Use protected access since someone may need to extend your code.
    // Initializing the data is encouraged.
    protected String sampleData = "";

    //-------------------------------------------------------------------------
    /**
        Constructs Sample object.
        Include the name of the parameter and its type in the javadoc.
        @param initialData String used to initialize the Sample.
    **/
    //-------------------------------------------------------------------------
    public Sample(String initialData)
    {
        sampleData = initialData;
        // Declare variables outside the loop
        int length = sampleData.length();
        BigDecimal[] numberList = new BigDecimal[length];

        // Precede code blocks with blank line and pertinent comment
        for (int i = 0; i < length; i++)
        {
            // Sample wrapping line.
            numberList[i] = someInheritedMethodWithALongName(Util.I_BIG_DECIMAL_
ONE,
 sampleData,
 length - i);
        }
    }
}
```

Javadoc

- Make code comments conform to Javadoc standards.
- Include a comment for every code block.
- Document every method's parameters and return codes, and include a brief statement as to the method's purpose.

Naming Conventions

Names should not use abbreviations except when they are widely accepted within the domain (such as the customer abbreviation, which is used extensively to distinguish customized code from product code).

Table 3–2 lists some additional naming conventions.

Table 3–2 Naming Conventions

Element	Description	Example
Package Names	Package names are entirely lower case and should conform to the documented packaging standards.	oracle.retail.stores.packagename com.mbs.packagname
Class Names	Mixed case, starting with a capital letter. Exception classes end in Exception; interface classes end in Ifc; unit tests append Test to the name of the tested class.	DatabaseException DatabaseExceptionTest FoundationScreenIfc
File Names	File names are the same as the name of the class.	DatabaseException.java
Method Names	Method names are mixed case, starting with a lowercase letter. Method names are an action verb, where possible. Boolean-valued methods should read like a question, with the verb first. Accessor functions use the prefixes get or set.	isEmpty() hasChildren() getAttempt() setName()
Attribute Names	Attribute names are mixed case, starting with a lowercase letter.	lineItemCount
Constants	Constants (static final variables) are named using all uppercase letters and underscores.	final static int NORMAL_SIZE = 400
EJBs—entity	Use these conventions for entity beans, where 'Transaction' is a name that describes the entity.	TransactionBean TransactionIfc TransactionLocal TransactionLocalHome TransactionRemote TransactionHome
EJBs—session	Use these conventions for session beans, where 'Transaction' is a name that describes the session.	TransactionService TransactionAdapter TransactionManager

SQL Guidelines

The following general guidelines apply when creating SQL code:

- Keep SQL code out of client/UI modules. Such components should not interact with the database directly.
- Table and column names must be no longer than 18 characters.
- Comply with ARTS specifications for new tables and columns. If you are creating something not currently specified by ARTS, strive to follow the ARTS naming conventions and guidelines.
- Document and describe every object, providing both descriptions and default values so that we can maintain an up-to-date data model.
- Consult your data architect when designing new tables and columns.
- Whenever possible, avoid vendor-specific extensions and strive for SQL-92 compliance with your SQL.
- While database-specific extensions are common in the code base, do not introduce currently unused extensions, because they must be ported to the DataFilters and JdbcHelpers for other databases.

- All SQL commands should be uppercase because the DataFilters currently only handle uppercase.
- If database-specific code is used in the source, move it into the JdbcHelpers.
- All JDBC operations classes must be thread-safe.

Do the following to avoid errors:

- Pay close attention when cutting and pasting SQL.
- Always place a carriage return at the end of the file.
- Test your SQL before committing.

The subsections that follow describe guidelines for specific database environments.

Table 3–3 provides some examples of common syntax problems which cause Oracle to produce errors.

Table 3–3 Oracle SQL Code Problems

Problem	Problem Code	Corrected Code

Blank line in code block causes error.	`CREATE TABLE BLAH`
	`(`
	`FIELD1 INTEGER,`
	`FIELD2 VARCHAR(20)`
	
	`);`
		`CREATE TABLE BLAH`
		`(`
		`FIELD1 INTEGER,`
		`FIELD2 VARCHAR(20)`
		`);`

When using NOT NULL with a default value, NOT NULL must follow the DEFAULT statement.	`CREATE TABLE BLAH`
	`(`
	`FIELD1 INTEGER NOT NULL DEFAULT 0,`
	`FIELD2 VARCHAR(20)`
	`);`
		`CREATE TABLE BLAH`
		`(`
		`FIELD1 INTEGER DEFAULT 0 NOT NULL,`
		`FIELD2 VARCHAR(20)`
		`);`

In a CREATE or INSERT, do not place a comma after the last item.	`CREATE TABLE BLAH`
	`(`
	`FIELD1 INTEGER,`
	`FIELD2 VARCHAR(20),`
	`);`
		`CREATE TABLE BLAH`
		`(`
		`FIELD1 INTEGER,`
		`FIELD2 VARCHAR(20)`
		`);`

Unit Testing

For details on how to implement unit testing, see separate guidelines on the topic. Some general notes apply:

- Break large methods into smaller, testable units.
- Although unit testing may be difficult for tour scripts, apply it for Java components within Point-of-Service code.
- If you add a new item to the codebase, make sure your unit tests prove that the new item can be extended.
- In unit tests, directly create the data/preconditions necessary for the test (in a setup() method) and remove them afterwards (in a teardown() method). JUnit expects to use these standard methods in running tests.

Architecture and Design Guidelines

This section provides guidelines for making design decisions which are intended to promote a robust architecture.

AntiPatterns

An AntiPattern is a common solution to a problem which results in negative consequences. The name contrasts with the concept of a pattern, a successful solution to a common problem.

Table 3–4 identifies AntiPatterns which introduce bugs and reduce the quality of code.

Table 3–4 Common AntiPatterns

Pattern	Description	Solution
Reinvent the Wheel	Sometimes code is developed in an unnecessarily unique way that leads to errors, prolonged debugging time and more difficult maintenance.	The analysis process for new features provides awareness of existing solutions for similar functionality so that you can determine the best solution. There must be a compelling reason to choose a new design when a proven design exists. During development, a similar pattern should be followed in which existing, proven solutions are implemented before new solutions.
Copy-and-paste Programming, classes	When code needs to be reused, it is sometimes copied and pasted instead of using a better method. For example, when a whole class is copied to a new class when the new class could have extended the original class. Another example is when a method is being overridden and the code from the super class is copied and pasted instead of calling the method in the super class.	Use object-oriented techniques when available instead of copying code.

Table 3–4 (Cont.) Common AntiPatterns

Pattern	Description	Solution
Copy-and-paste Programming, XML	A new element (such as a Site class or an Overlay XML tag) can be started by copying and pasting a similar existing element. Bugs are created when one or more pieces are not updated for the new element. For example, a new screen might have the screen name or prompt text for the old screen.	If you copy an existing element to create a new element, manually verify each piece of the element to ensure that it is correct for the new element.
Project Mismanagement/ Common Understanding	A lack of common understanding between managers, Business Analysts, Quality Assurance and developers can lead to missed functionality, incorrect functionality and a larger-than-necessary number of defects. An example of this is when code does not match requirements, including details like maximum length of fields and dialog message text.	Read the requirement before you code. If there is disagreement with content, raise an issue with the Product Manager. Before you consider code for the requirement finished, all issues must be resolved and the code must match the requirements.
Stovepipe	Multiple systems within an enterprise are designed independently. The lack of commonality prevents reuse and inhibits interoperability between systems. For example, a change to till reconcile in Back Office may not consider the impact on Point-of-Service. Another example is a making change to a field in the Oracle Retail database for a Back Office feature without handling Point-of-Service effects.	Coordinate technologies across applications at several levels. Define basic standards in infrastructures for the suite of products. Only mission-specific functions should be created independently of the other applications within the suite.

Designing for Extension

This section defines how to code product features so that they may be easily extended.

> **Note:** See the *Oracle Retail POS Suite Security Guide* for more information about specific security features and implementation guidelines for the POS Suite products.

- Separate external constants such as database table and column names, JMS queue names, port numbers from the rest of the code. Store them in (in order of preference):
  - Configuration files
  - Deployment descriptors
  - Constant classes/interfaces
- Make sure the SQL code included in a component does not touch tables not directly owned by that component.

- Make sure there is some separation from DTO and ViewBean type classes so we have abstraction between the service and the presentation.
- Consider designing so that any fine grained operation within the larger context of a coarse grain operation can be factored out in a separate algorithm class, so that it can be replaced without reworking the entire activity flow of the larger operation.

Common Frameworks

This section provides guidelines which are common to the Oracle Retail POS Suite applications.

Internationalization

Internationalization is the process of creating software that can be translated easily. Changes to the code are not specific to any particular market. Oracle Retail POS Suite has been internationalized to support multiple languages. This section describes configuration settings and features of the software that ensure that the base application can handle multiple languages.

Translation

Translation is the process of interpreting and adapting text from one language into another. Although the code itself is not translated, components of the application that are translated may include the following, among others:

- Graphical user interface (GUI)
- Error messages

The following components are not usually translated:

- Documentation (for example, Online Help, Release Notes, Installation Guide, User Guide, Operations Guide)
- Batch programs and messages
- Log files
- Configuration Tools
- Reports
- Demo data
- Training Materials

The user interface for Oracle Retail POS Suite has been translated into:

- Chinese (Simplified)
- Chinese (Traditional)
- Croatian
- Dutch
- French
- German
- Greek
- Hungarian
- Italian

- Japanese
- Korean
- Polish
- Portuguese (Brazilian)
- Russian
- Spanish
- Swedish
- Turkish

Logging

Oracle Retail POS Suite applications use Log4J for logging. When writing log commands, use the following guidelines:

- Use calls to Log4J rather than System.out from the beginning of your development. Unlike System.out, Log4J calls are naturally written to a file, and can be suppressed when desired.
- Log exceptions where you catch them, unless you are going to rethrow them. This is preserves the context of the exceptions and helps reduce duplicate exception reporting.
- Use the correct logging level:
  - FATAL—exceptions that cause the application to fail
  - ERROR—nonfatal, unhandled exceptions (there should be few of these)
  - INFO—life cycle/heartbeat information
  - DEBUG—information for debugging purposes

The following sections provide additional information on guarding code, when to log, and how to write log messages.

Guarding Code

Testing shows that logging takes up very little of a system's CPU resources. However, if a single call to your formatter is abnormally expensive (stack traces, database access, network IO, large data manipulations, and so forth), you can use Boolean methods provided in the Logger class for each level to determine whether you have that level (or better) currently enabled; Jakarta calls this a code guard:

Example 3–2 Wrapping Code in a Code Guard

```
if (log.isDebugEnabled()) {
log.debug(MassiveSlowStringGenerator().message());
}
```

An interesting use of code guards, however, is to enable debug-only code, instead of using a DEBUG flag. Using Log4J to maintain this functionality lets you adjust it at runtime by manipulating Log4J configurations.

For instance, you can use code guards to simply switch graphics contexts in your custom swing component:

Example 3–3 Switching Graphics Contexts via a Logging Level Test

```
protected void paintComponent(Graphics g) {

    if (log.isDebugEnabled()) {
        g = new DebugGraphics(g, this);
    }

    g.drawString("foo", 0, 0);
}
```

When to Log

There are three main cases for logging:

- Exceptions—Should be logged at an error or fatal level.
- Heartbeat/Life cycle—For monitoring the application; helps to make unseen events clear. Use the info level for these events.
- Debug—Code is usually littered with these when you are first trying to get a class to run. If you use System.out, you have to go back later and remove them to keep. With Log4J, you can simply raise the log level. Furthermore, if problems pop up in the field, you can lower the logging level and access them.

Writing Log Messages

When Log4J is being used, any log message might be seen by a user, so the messages should be written with users in mind. Cute, cryptic, or rude messages are inappropriate. The following sections provide additional guidelines for specific types of log messages.

Exception Messages

A log message should have enough information to give the user a good shot at understanding and fixing the problem. Poor logging messages say something opaque like “load failed.”

Take this piece of code:

```
try {
    File file = new File(fileName);
    Document doc = builder.parse(file);

    NodeList nl = doc.getElementsByTagName("molecule");
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        // something here
    }

} catch {
   // see below
}
```

and these two ways of logging exceptions:

```
} catch (Exception e){
    log.debug("Could not load XML");
}


} catch (IOException e){
```

```
    log.error("Problem reading file " + fileName, e);
} catch (DOMException e){
    log.error("Error parsing XML in file " + fileName, e);
} catch (SAXException e){
    log.error("Error parsing XML in file " + fileName, e);
}
```

In the first case, you'll get an error that just tells you something went wrong. In the second case, you're given slightly more context around the error, in that you know if you can't find it, load it, or parse it, and you're given that key piece of data: the file name.

The log lets you augment the message in the exception itself. Ideally, with the messages, the stack trace, and type of exception, you'll have enough to be able to reproduce the problem at debug time. Given that, the message can be reasonably verbose.

For instance, the fail() method in JUnit really just throws an exception, and whatever message you pass to it is in effect logging. It's useful to construct messages that contain a great deal of information about what you are looking for:

Example 3–4 JUnit

```
if (! list.contains(testObj)) {

    StringBuffer buf = new StringBuffer();
    buf.append("Could not find object " + testObj + " in list.\n");
    buf.append("List contains: ");
    for (int i = 0; i < list.size(); i++) {
        if (i > 0) {
             buf.append(",");
        }
        buf.append(list.get(i));
    }
    fail(buf.toString());
}
```

Heartbeat or Life Cycle Messages

The log message here should succinctly display what portion of the life cycle is occurring (login, request, loading, etc.) and what apparatus is doing it (is it a particular EJB are there multiple servers running, etc.)

These message should be fairly terse, since you expect them to be running all the time.

Debug Messages

Debug statements are going to be your first insight into a problem with the running code, so having enough, of the right kind, is important.

These statements are usually either of an intra-method-life cycle variety:

```
    log.debug("Loading file");

    File file = new File(fileName);
    log.debug("loaded.  Parsing...");
    Document doc = builder.parse(file);
    log.debug("Creating objects");
    for (int i ...
```

or of the variable-inspection variety:

```
log.debug("File name is " + fileName);

log.debug("root is null: " + (root == null));
log.debug("object is at index " + list.indexOf(obj));
```

Exception Handling

The key guidelines for exception handling are:

- Handle the exceptions that you can (File Not Found, etc.)
- Fail fast if you can't handle an exception
- Log every exception with Log4J, even when first writing the class, unless you are rethrowing the exception
- Include enough information in the log message to give the user or developer a fighting chance at knowing what went wrong
- Nest the original exception if you rethrow one

Types of Exceptions

The EJB specification divides exceptions into the following categories:

JVM Exceptions

You cannot recover from these; when one is thrown, it's because the JVM has entered a kernel panic state that the application cannot be expected to recover from. A common example is an Out of Memory error.

System Exceptions

Similar to JVM exceptions, these are generally, though not always, "non-recoverable" exceptions. In the commons-logging parlance, these are "unexpected" exceptions. The canonical example here is NullPointerException. The idea is that if a value is null, often you don't know what you should do. If you can simply report back to your calling method that you got a null value, do that. If you cannot gracefully recover, say from an IndexOutOfBoundsException, treat as a system exception and fail fast.

Application Exceptions

These are the expected exceptions, usually defined by specific application domains. It is useful to think of these in terms of recoverability. A FileNotFoundException is sometimes easy to rectify by simply asking the user for another file name. But something that's application specific, like JDOMException, may still not be recoverable. The application can recognize that the XML it is receiving is malformed, but it may still not be able to do anything about it.

Avoid java.lang.Exception

Avoid throwing the generic Exception; choose a more specific (but standard) exception.

Avoid Custom Exceptions

Custom exceptions are rarely needed. The specific type of exception thrown is rarely important; do not create a custom exception if there is a problem with the formatting of a string (ApplicationFormattingException) instead of reusing IllegalArgumentException.

The best case for writing a custom exception is if you can provide additional information to the caller which is useful for recovering from the exception or fixing the

problem. For example, the JPOSExceptions can report problems with the physical device. An XML exception could have line number information embedded in it, allowing the user to easily detect where the problem is. Or, you could subclass NullPointer with a little debugging magic to tell the user what method of variable is null.

Catching Exceptions

The following sections provide guidelines on catching exceptions.

Keep the Try Block Short The following example, from a networking testing application, shows a loop that was expected to require approximately 30 seconds to execute (since it calls sleep(3000) ten times):

Example 3–5 Network Test

```
    for (int i = 0; i < 10; i++) {
        try {
            System.out.println("Thread " + Thread.currentThread().getName() + "
requesting number " + i);
            URLConnection con = myUrl.openConnection();
            con.getContent();
            Thread.sleep(3000);
        } catch (Exception e) {
            log.error("Error getting connection or content", e);
        }
    }
```

The initial expectation was for this loop to take approximately 30 seconds, since the sleep(3000) would be called ten times. Suppose, however, that con.getContent() throws an IOException. The loop then skips the sleep() call entirely, finishing in 6 seconds. A better way to write this is to move the sleep() call outside of the try block, ensuring that it is executed:

Example 3–6 Network Test with Shortened Try Block

```
    for (int i = 0; i < 10; i++) {

        try {
            System.out.println("Thread " + Thread.currentThread().getName() + "
requesting number " + i);
            URLConnection con = myUrl.openConnection();
            con.getContent();
        } catch (Exception e) {
            log.error("Error getting connection or content", e);
        }
        Thread.sleep(3000);
    }
```

Avoid Throwing New Exceptions When you catch an exception, then throw a new one in its place, you replace the context of where it was thrown with the context of where it was caught.

A slightly better way is to throw a wrapped exception:

Example 3–7 Wrapped Exception

```
1:   try {
2:       Class k1 = Class.forName(firstClass);
```

```
3:       Class k2 = Class.forName(secondClass);
4:       Object o1 = k1.newInstance();
5:       Object o2 = k2.newInstance();
6:
7:   } catch (Exception e) {
8:      throw new MyApplicationException(e);
9:   }
```

However, the onus is still on the user to call getCause() to see what the real cause was. This makes most sense in an RMI type environment, where you need to tunnel an exception back to the calling methods.

The better way than throwing a wrapped exception is to simply declare that your method throws the exception, and let the caller figure it out:

Example 3–8 Declaring an Exception

```
public void buildClasses(String firstName, String secondName)
       throws InstantiationException, ... {

       Class k1 = Class.forName(firstClass);
       Class k2 = Class.forName(secondClass);
       Object o1 = k1.newInstance();
       Object o2 = k2.newInstance();
   }
```

However, there may be times when you want to deal with some cleanup code and then rethrow an exception:

Example 3–9 Clean Up First, then Rethrow Exception

```
try {
        someOperation();
    } catch (Exception e) {
        someCleanUp();
        throw e;
    }
```

Catching Specific Exceptions There are various exceptions for a reason: so you can precisely identify what happened by the type of exception thrown. If you just catch Exception (rather than, say, ClassCastException), you hide information from the user. On the other hand, methods should not generally try to catch every type of exception. The rule of thumb is the related to the fail-fast/recover rule: catch as many different exceptions as you are going to handle.

Favor a Switch over Code Duplication The syntax of try and catch makes code reuse difficult, especially if you try to catch at a granular level. If you want to execute some code specific to a certain exception, and some code in common, you're left with either duplicating the code in two catch blocks, or using a switch-like procedure. The switch-like procedure, shown below, is preferred because it avoids code duplication:

Example 3–10 Using a Switch to Execute Code Specific to an Exception

```
    try{
        // some code here that throws Exceptions...
    } catch (Exception e) {
        if (e instanceof LegalException) {
            callPolice((LegalException) e);
        } else if (e instanceof ReactorException) {
```

```
            shutdownReactor();
        }
        logException(e);
        mailException(e);
        haltPlant(e);
    }
```

This example is preferred, in these relatively rare cases, to using multiple catch blocks:

Example 3–11 Using Multiple Catch Blocks Causes Duplicate Code

```
try{
        // some code here that throws Exceptions...
    } catch (LegalException e) {
        callPolice(e);
        logException(e);
        mailException(e);
        haltPlant(e);
    } catch (ReactorException e) {
        shutdownReactor();
        logException(e);
        mailException(e);
        haltPlant(e);
    }
```

Exceptions tend to be the backwater of the code; requiring a maintenance developer, even yourself, to remember to update the duplicate sections of separate catch blocks is a recipe for future errors.

4

Point-of-Service Development Standards

The following standards specific to the Point-of-Service architecture have been adopted by Oracle Retail product and service development teams. These standards are intended to reduce bugs and increase the quality of the code.

Note: See the *Oracle Retail POS Suite Security Guide* for more information about specific security features and implementation guidelines for the POS Suite products.

Screen Design and User Interface Guidelines

Avoid creating new screen beans and screen models for every new screen. Look for ways to reuse existing or generic beans, such as the Data Input Bean, to avoid complicating the code base.

Tour Framework

This section includes general guidelines as well as subsections on specific tour components.

For more information, see Chapter 12.

Tour Architectural Guidelines

Consult these guidelines when making architecture decisions in tour framework designs.

- Services—When designing services, consider their size and reusability. Services that are overlarge create additional work when a portion must be extended.
- Utility Manager—Put methods used by multiple services in this manager so they can be easily extended.
- If the reusable behavior contains flow-dependent behavior, then it is best implemented as a Site and the Site action can be reused within a Service or across Services.
- Large bodies of reusable behavior can be implemented as Managers and Technicians. This pattern is especially useful if the user might offload the processing to a separate CPU.

General Tour Guidelines

- Code that uses bus resources must reside in a Site action, Lane action, Signal or Shuttle.
- Never mail a letter from a Road. This causes unpredictable results.
- Never define local data in a Site, Aisle, Road or Signal. Local data is not guaranteed when processing across multiple tiers. Sites and Lanes must be stateless. This is the purpose of Cargo.
- Traffic Signals should not modify Cargo. Signals should only be used to evaluate a condition as true or false. Anything else is a side effect, reducing the maintainability of the system.
- Never implement just one Signal. Always implement Signals when there is more than one Road that responds to the same letter, or when there is an Aisle and a Road that respond to the same letter. See Signals.
- Send letters at the end of methods. If the choice of which letter to send depends on conditions which occur during the method, store the method name and mail it at the end of the method.
- Do not mail letters from depart() and undo() in Sites, backup() and traverse() in Roads, roadClear() in Signals, and load() and unload() in Shuttles. Letters can be mailed from traverse() in Aisles.
- Define Shuttles in the calling Service package. If they are reusable Shuttles, define them in a common package.

Table 4–1 provides naming conventions for Tour components.

Table 4–1 Tour Naming Conventions

Element	Description	Example
Service	description of the related functionality	Login
Site element	VerbNoun—indicating the action taking place at the Site	EnterID
Site class	The same as the Site name, with Site as a suffix	EnterIDSite.java
Road element	NounVerb—indicating the event that caused the Road to be taken	IDEntered
Road class	The same as the Road name, with Road as a suffix	IDEnteredRoad.java
Aisle element	NounVerb- indicating the event that caused the Aisle to be taken	PasswordEntered
Aisle class	The same as the Aisle name, with Aisle as a suffix	PasswordEnteredAisle.java
Cargo	ServiceNameCargo	LoginCargo.java

Table 4–1 (Cont.) Tour Naming Conventions

Element	Description	Example
Letter	One word action name indicating the event; see list defined in commonLetterIfc.java	■ Success ■ Failure ■ Continue ■ Next ■ Cancel ■ OK ■ Retry ■ Invalid ■ Add ■ Yes ■ No ■ Undo ■ Done
Transfer Station element	NestedServiceNameStation	FindCustomerStation
Shuttle class	NestedServiceNameLaunchShuttle	FindCustomerLaunchShuttle.java
	NestedServiceNameReturnShuttle	FindCustomerReturnShuttle.java
Traffic Signal class	IsCondition.java-indicating the condition being tested	IsAuthRequiredSignal.java

Foundation

- The best reuse in the Foundation engine takes place at the Service level. Sites require extra thought because they can affect flow. Lane actions can be reused without flow implications. Signals and Shuttles are very well suited to reuse especially when interfaces are developed for accessing Cargo.
- If validation and database lookup are coded in Aisles, they may be good candidates for reuse in several Sites as well as in multiple Services.
- All component pieces need to be designed with care for reuse: they must be context insensitive or must do a lot of checking to make sure that the managers they access exist for the bus that is active, the Cargo contains the data they need, etc.
- Trying to maximize reuse can result in confusing code with too many discrete parts. If the reusable unit consists of one or two lines of code, consider whether there is sufficient payoff in reusing the unit of code. If the code contains a complex calculation that is subject to change over time, then isolating this logic in one place may be well worth the effort.

Tours and Services

- There is often a one-to-one mapping between a Use Case and a Service. The Service should provide the best opportunity for reuse. If you design for reuse, it should be focused at the Service level. This is where you get your best return on investment.
- Maintenance is a matter of choosing a style and implementing it consistently within a Service and sometimes within an entire application. When you are comfortable with how TourCam works, maintaining TourCam Services is easy.

- Aisles help reduce the total number of Sites in a Service, but they may be harder to see because they are contained within a Site.
- When making choices, give making an application as consistent and easy to maintain as possible the top priority.
- Consider the performance costs of using TourCam or creating additional Sites when designing a Service.
- A Service can often be simplified by reducing the number of individual Sites. You can do this by using Aisles to replace Sites; Sites with one exit Road can be good candidates, and Aisles are good candidates for reuse. However, Aisles are less visible than Roads.

Sites

- Reusing a Site has flow implications. Site classes can be reused whenever the exit conditions are identical. Reusable Sites should be packaged in a common package as opposed to one of the packages that use them. A reusable Site must refer to a reusable Cargo or a common Cargo interface.
- Treat the sending of a letter like a return code: put it at the end of your arrive() or traverse() method. Sending letters in the middle of the arrive() method may cause duplicate letters (with unpredictable results), or no letters (with no results).
- Do not try to store state information in instance variables. Pass in state information through arguments.
- Do not put a lot of functionality in arrive(), traverse() methods. Decompose them into logical methods that each have one job. For methods not called from outside the package, protect the methods.

Managers and Technicians

- There is a high degree of reuse of Managers and Technicians across the applications. For example, the DataTransactions and DataActions are reusable. By design, it is the DataOperations that change with different database implementations. The UIManager and UITechnician expect a lot of reuse of beans, adapters, and specification objects. In fact, the UISubsystem looks in the UI Script for most of the configuration information that effects changes in screen layout, bean interactions and even bean composition.
- Utility methods can be useful for capturing behavior that is used by many Services, but does not lend itself to Site or Aisle behavior. Put Utility methods in a UtilityManager so they can be easily extended. The Point-of-Service application contains an example of this called the POSUtilityManager. Service developers can access these methods through the POSUtilityManagerIfc. The UtilityManager and UtilityManagerIfc classes can be extended and the new class is specified through the Conduit Script. For general-purpose behavior that can be called from a Site, Lane, or even from a Signal, use utility methods to capture the common reusable behavior rather than extending a common Site.
- Large bodies of reusable behavior can be implemented as Managers and Technicians. This pattern is especially useful if the user might off-load the processing to a separate CPU.

Roads

It is sometimes useful to define multiple Roads from an origin Site to the same destination if they capture different Road traversal conditions.

Do not trap and change the name of a letter just to reduce the number of Roads in a Service. This is a poor use of system resources and also hides useful information from the reader of the Tour Script. Do not rename letters except as noted in Renaming Letters.

For example, the Return Transaction Service has two Roads with the same origin (LookupItem) and the same destination (EnterReturnItemInformation), but the letters that invoke these two Roads are different.

The use of Road actions is dependent on a number of factors: use of TourCam, developer conventions for an application, number of classes generated, and maintainability.

Use Road actions for outcome-specific behavior. If you need to store some data in Cargo on the sending of a specific letter, do the Cargo storage in the traverse() method of the Road that is associated with that letter. If the data must be stored in Cargo before leaving a Site, put the logic in the Site's depart() method. Code in a Site or Aisle's depart() method should not check to see what letter was sent before taking an action; use a Road in that case.

Aisles

Aisles are used to implement behavior that occurs within a Site. When there is interaction with an external source (e.g. user, database) use a Site. When you are doing business validation which may keep you in the same screen, use an Aisle.

While it makes sense to create Roads without corresponding Road actions, Aisles are useless without an Aisle action. The important thing about an Aisle is that it is not part of a transition from one Site to another, so the only code that gets executed in an Aisle is the traverse() method. The arrive() and depart() methods are never executed on a Site when an Aisle is processed. The Aisle can initiate an action that causes a transition to another Site, but it cannot transition itself.

Aisle actions can be used to validate data, compute values, provide looping behavior, and do database lookups. Aisle actions are useful for capturing repeatable behavior that can occur while the bus is still in a Site.

For example, suppose you define a Site that gathers data from the user. The data validation is implemented as an Aisle. Because it is an Aisle, the user can repeat the process of entering data, validating, and re-entering until the data is correct, with little system overhead. The Aisle behavior can be triggered over and over without calling the arrive() method on the Site (a Road back to the Site calls the arrive() method).

Aisles are also useful for looping through a list of items when each item may require error handling. This is done by placing the loop index in the Cargo.

Signals

You cannot use a signal alone; they must be used in groups of two or more. If there is more than one Lane that responds to the same letter, each Lane must implement a Signal. The logic in the Signals must be mutually exclusive; there should be only one valid Road that can be traversed at any time; otherwise, unexpected (and difficult to debug) behavior could occur.

When there are more than two Signals, each of the Signals should evaluate in such a way that only one Signal is green at any given time. But the presence of more than two Signals should raise a red flag. Track down the source of the following issues; determine if the UI or other letter generator needs to be sending more unique letters.

- Why are there so many Signals?
- What are they checking?
- Is the same letter being sent for many different conditions?

Use a Signal only to decide which road to take when you could go to two different places (such as Sites) with the same Letter, based on Cargo information. It should not be used to update cargo. The road you take after making a decision at the Signal should do the updating.

Choosing Among Sites, Aisles, and Signals

There are many times when an Aisle can do the same work as a Site. Sometimes a Signal can contain behavior that could be implemented in an Aisle. Sometimes a separate Service does the work that was once a Site if the Site needs to be reused or becomes too complicated. Consult the guidelines for your application development team in order to be consistent with the rest of your team.

If you have the following customer requirement:

- Display a UI screen that gathers search criteria to be used in a database lookup (for example, customer lookup). After the user enters the data, validate the data. Once the data has been validated, do the database lookup.

you have the following design choices:

- Implement as separate Sites and take advantage of TourCam to back up when the data is invalid or database lookup results are not correct.
- Implement as one Site with Aisles that do the validation and lookup.

The database lookup may result in a success or failure letter whether it is coded as a Site or an Aisle. When using an Aisle for database lookup, the failure letter triggers another Aisle that could display an error message but allow the user to re-enter the data and retry the lookup. This can occur without exiting the original Site. When using a Site, the failure condition can trigger a flow change to back up through the lookup Site back to the data entry Site.

If the validation and database lookup are coded in Aisles, they may be good candidates for reuse in several Sites as well as in multiple Services. Reusing the Site is also possible, especially if the TourCam's ability to back up to the last indexed Site is used. But there may be more considerations involving flow when trying to reuse a Site.

Renaming Letters

Use the following guidelines when deciding whether to rename letters:

- Do rename Letters when the application developer does not have power over the Letter that is mailed and there is more than one event associated with a single Letter.

  For example: a single Letter is sent from a button on the UI (such as dialog box OK), but the content of the retrieved data associated with the UI signals a different event notification (such as error message notification).

- Do rename Letters when a common exit Letter from a nested Service is needed.

- Do not rename Letters to reduce the number of Roads in a Service.

Shuttles

If you are creating a sub-tour (that is, a tour called from other tours using a Station) from scratch, use only the following final letters:

- Success
- Failure
- Cancel
- Undo

If you need to provide a reason for a Failure or need to return data to the calling service on a Success, use the Return Shuttle to update the calling service's cargo. Do not use letters to reflect sub-tour results.

Shuttle Type	Launch Shuttle	Return Shuttle
Description	Used to send parameter data to a sub-service	Used to return data to the parent service.
Methods	load()—can only see the parent Service's Cargo unload()—can only see the sub-service's Cargo	load()—can only see the sub-service's Cargo unload()—can only see the parent service's Cargo

Within the Tour Framework, Shuttles are used to transfer data in and out of Services. Shuttles are good candidates for reuse given a common Cargo interface.

Cargo

All Cargo classes should implement the CargoIfc interface.

Log Entry Format

This section describes the format and layout of log entries for the Point-of-Service application.

Log Entry Description

Log entries adhere to the following format:

```
LLLLL yyyyy-mm-dd hh:mm:ss,ttt bbbbbbb (<classname>):
      [<classname>.<methodname>(<filename>:<linenumber>)]
      <Log entry content>
```

Fixed Length Header

The entry begins with a fixed length record header (38 bytes) that adheres to the following layout:

```
LLLLL yyyyy-mm-dd hh:mm:ss,ttt bbbbbbb
12345678901234567890123456789012345678
```

LLLLL is the log message level and consists of one of the substrings in the following table.

Table 4–2 provides log message levels and their descriptions.

Table 4–2 Log Message Level

Log Message Level	Description
ERROR	Highest severity entry; critical
WARN	Application warning; serious
INFO	For information only
DEBUG	For developer use (not displayed by default application configuration

`yyyy-mm-dd` is the date.

`hh:mm:ss,ttt bbbbbbb` is the time stamp of the entry, comprised of the sub-fields described in the following table.

Table 4–3 provides time stamp fields and their descriptions.

Table 4–3 Time Stamp Fields

Field	Description
hh	Time of entry in hours, in 24-hour format
mm	Minutes past the full hour
ss	Seconds past the last full minute
ttt	Milliseconds past the last full second
bbbbbbb	Milliseconds since the application was started. Left justified and blank filled on the right, out to 7 places

Additional Logging Information

The fixed length record header is followed by a blank space followed by the parenthesized, fully qualified class name of the logging entity followed by a colon followed by a carriage return/line feed pair.

```
(<classname>):<cr><lf>
```

The next line in a log entry begins with 6 blank spaces and a square-bracketed sequence containing the following information:

```
<classname>.<methodname>(<filename>:<linenumber>)
```

Parentheses are included in the sequence. This sequence reflects the fully qualified name of the method invoking the logging action and the source line number in the file where the logging call was made.

The next lines in a log entry are the log entry content. The content is comprised of freeform text supplied by the calling routine. The content reflected in the freeform text may be multiple lines in length.

The next log entry is delineated with another 38 byte fixed length header beginning in column one of the text log file.

Example Log Entry

```
INFO  2004-09-02 11:12:41,253 23697
(main:oracle.retail.stores.foundation.manager.gui.DefaultBeanConfigurator):
```

```
[oracle.retail.stores.foundation.manager.gui.DefaultBeanConfigurator.applyProperti
es(DefaultBeanConfigurator.java:198)]
      Applying property cachingScheme to Class: DialogBean (Revision 1.9)
@12076742
```

5

Point-of-Service Extension Guidelines

Customers who purchase Point-of-Service extend the product to meet their particular needs. These guidelines speed implementation and simplify the upgrade path for future work.

Developers on customer projects should also refer to the Development Standards. The Development Standards address how to code product features to make them less error-prone and more easily maintained. They are especially important if code from the customer implementation may be rolled back into the base product.

Note: See the *Oracle Retail POS Suite Security Guide* for more information about specific security features and implementation guidelines for the POS Suite products.

Conventions

This section describes conventions used throughout this chapter.

Terms

The following definitions are used throughout the chapter:

- Product source tree — A directory tree that contains the Oracle Retail product code. The contents of this tree do not change, with the exception of product patches. In production code, these files are accessed as external .jar files.
- Customer source tree — A directory tree separate from the product code that contains customer-specific files. Some of these files are new files for customer-specific features; others are extensions or replacements of files from the product source tree. The customer tree should not contain packages from the product tree.
- Customer abbreviation — A short name that represents the customer. For example, a company named My Bike Store might use MBS as their customer abbreviation. The MBS example is used throughout this chapter; replace MBS with the customer abbreviation for your own project when writing code. The customer abbreviation is added to filenames to clarify that the file is part of the customized code, and is used as part of the package name in the customer source tree.

Filename Conventions

Filenames in the customer source tree usually include the customer abbreviation. Name files according to the following rules:

- If a class in the customer source tree extends or replaces a class in the product source tree, use the customer abbreviation followed by the original filename as the new filename (for example, SaleReceipt.java becomes MBSSaleReceipt.java).
- New Java classes should also begin with the customer abbreviation.
- Script or properties file names that are hard-coded in Foundation classes must use the same filename in the customer source tree as was used in the product source tree (for example, posfoundation.properties).

Modules

The Point-of-Service system is divided into a number of different modules, and each module corresponds to a project in an integrated development environment (IDE). When setting up a development environment for modifying code, building Point-of-Service, and testing changes, you must configure your system to make MBSpos dependent on all the other modules.

Do the following to set up your development environment:

1. Check out each of the required customer modules as shown in the following table.
2. Reference each of the standard modules as external .jar files.
3. Add the required modules to your CLASSPATH environment variable in the order shown in the following table, with all of the customer modules preceding the set of standard modules.

Table 5–1 identifies required customer and standard modules in their respective dependency orders.

Table 5–1 Required Modules in Dependency Order

Customer Modules	Standard Modules
MBS pos (root, src, locales and other language directories)[1]	pos (root, src, locales and other language directories)
MBS domain (root and src)	domain (root and src)
MBS commerceservices	common
MBS common	commerceservices
MBS 3rd-party	foundation
MBS utility	3rd party
MBS exportfile	utility
	exportfile

[1] Directory names in parentheses must be specified individually in the classpath.

Directory Paths

Paths given in this chapter are relative, starting either with the module or with the source code, as follows:

- Paths beginning with a module name start from the module location. pos\config refers to the config directory within the pos module, wherever that module is located on your system.
- Paths beginning with com refer to source code. Source code paths are nested within modules, in \src directories. Multiple \src\com file hierarchies are built together into one file structure during compilation. For example, a reference to `oracle\retail\stores\pos\services\tender` can be found in the pos module's

src directory. If your pos module is in `c:\workspace\OracleRetailStore`, then the full path is:

```
C:\workspace\OracleRetailStore\applications\pos\src\oracle\retail\stores\pos\services\tender
```

POS Package

This section addresses extension of files in the POS package.

> **Note:** The POS module might be nested within an OracleRetailStore directory in the source code control system.

Tour

You extend tours mainly by editing proprietary XML scripts developed by Oracle Retail. This section describes how to customize tours, beginning with replacing the Tour Map, and continuing with customization of individual tours or parts of tours.

Tour Map

The product code references tours at transfer stations by logical names, so that you can change a single tour without having to update references to that tour in various tour scripts. Tour maps tell the system the specific tour files to use for each logical name.

The tour map also enables overlays of tour classes. If a tour script does not need to be customized, but some of the Java classes do, the tour map can specify individual classes to customize. Note that any class files must still use their own unique names (such as MBScashSelectedAisle.java for a new Aisle used in place of CashSelectedAisle.java).

Typically, the base product Tour Map file, tourmap.xml, does not change. Instead, you create a custom Tour Map for your project. This Tour Map file contains only the differences it adds to the base Tour Map.

Do the following to add new Tour Map files:

1. Create one custom Tour Map file in the pos\config directory of the customer source tree. Initially, this Tour Map file may be empty; as you customize tour components, you can add tags.
2. Copy the pos\config\posfoundation.properties file to the customer source tree. Modify the tourmap.files property in this file, adding the names of the new Tour Map files. Do not rename the posfoundation.properties file, since this filename is referenced by Foundation classes. It is important to keep the customized tour map files after the product tour map file in the list, since the files listed later override earlier files.

Example 5–1 posfoundation.properties: Adding new Tour Maps

```
# comma delimited list of tourmap files to overlay
tourmap.files=tourmap.xml, MBStourmap.xml
```

3. Refer to the procedures that follow to modify tour scripts and Java components of a tour.

Tour Scripts

If you need to change the workflow of a tour, you must replace the tour script; you cannot extend a tour script. To replace a tour script, follow these steps:

1. Create a new XML tour script in the customer source tree.
2. Modify the tour map in the customer source tree to specify the correct package and filename for the new tour script. The logical tour name must stay the same.

Example 5–2 MBStourmap.xml: Replacing one tour script

```
<tour name="tender">
       <file>classpath://com/mbs/pos/services/tender/tender.xml</file>
</tour>
```

3. Copy and modify sites, roads, aisles, shuttles and signals.

Site

Extending siteactions in the traditional object-oriented sense is not recommended; letters mailed in the original arrive method would conflict with the arrive method in the extended class. Since siteactions represent relatively small units of code, they should be replaced instead of extended. Perform these steps:

1. Create a new siteaction class in the customer source tree, such as MBScashSelectedSite.java.
2. If you are overlaying a siteaction class, but not modifying the tour script, then all letters that were mailed from the product version of the siteaction class should also be mailed from the new version. Do not mail new letters that are not handled by the product code, unless the tour script and related Java classes are also modified.
3. Edit the appropriate Tour Map, using the replacewith property in the <SITEACTION> tag to define the new package and filename for the siteaction class.

Example 5–3 MBStourmap.xml: Replacing a siteaction

```
<tour name="tender">
   <file>classpath://com/mbs/pos/services/tender</file>
   <SITE
      name="cashSelected"
      useaction="oracle.retail.stores.pos.services.tender.cashSelectedSite"/>
   <SITEACTION
      class="cashSelectedSite"
      replacewith="com.mbs.pos.services.tender.MBScashSelectedSite"/>

</tour>
```

Lane—Road or Aisle

As with siteactions, extending laneactions in the traditional object-oriented sense is not recommended, as letters from the original and extended classes could conflict. Do the following to replace laneactions instead of extending them:

1. Create a new laneaction class in the customer source tree, such as MBSOpenCashDrawerAisle.java.
2. If you are overlaying a siteaction class, but not modifying the tour script, then all letters that were mailed from the product version of the laneaction class should also be mailed from the new version. Do not mail new letters that are not handled

by the product code, unless the tour script and related Java classes are also modified.

3. Edit the appropriate Tour Map using the replacewith property in the <LANEACTION> tag to define the new package and filename for the laneaction.

Example 5–4 MBStourmap.xml: Replacing a laneaction

```
<tour name="tender”>
       <file>classpath://com/mbs/pos/services/tender</file>
       <SITE
          name="RefundDueUI"
          useaction="com.mbs.pos.services.tender.refundDueUISite">"/>
       <LANEACTION
          class="OpenCashDrawerAisle"
          replacewith="com.mbs.pos.services.tender.MBSOpenCashDrawerAisle"/>

    </tour>
```

Shuttle

Since shuttles do not mail letters, they may be extended or replaced; however extending them is recommended. Do the following to either extend or replace shuttles:

1. Modify the shuttle class.

   Create a new class in the customer source tree. If it extends or replaces the product bean class, add the customer abbreviation to the filename. For example, TenderAuthorizationLaunchShuttle.java becomes MBSTenderAuthorizationLaunchShuttle.java.

2. Edit the appropriate Tour Map using the replacewith property in the <SHUTTLE> tag to define the new package and filename for the shuttle.

Example 5–5 MBStourmap.xml: Replacing or Extending a shuttle

```
<tour name="tender”>
       <file>classpath://com/mbs/pos/services/tender</file>
       <SITE
          name="RefundDueUI"
          useaction="com.mbs.pos.services.tender.refundDueUISite">"/>
       <SHUTTLE
          class="TenderAuthorizationLaunchShuttle"
replacewith="com.mbs.pos.services.tender.MBSTenderAuthorizationLaunchShuttle"/>
    </tour>
```

3. Modify the calling and nested tour scripts as necessary to adjust to the change.

Signal

Extending signals in the traditional object-oriented sense is not recommended. This is because signals are typically so small that extending an original signal class makes them overly complex.

The REPLACEWITH tag of the TourMap does not work for Signals. The tour script must be customized to refer to the package and filename of the new signal. Perform these steps:

1. Create a new signal class in the customer source tree. For example, create a replacement for IsAuthorizationRequiredSignal.java in the Tender service by creating a class file com\mbs\pos\services\tender\MBSIsAuthorizationRequiredSignal.java.

2. Customize the appropriate tour script.

Example 5–6 MBStender.xml: Tender tour script with customized signal

```
<SERVICECODE>
... non-signal declarations omitted...
     <SIGNAL class="IsReturnTransactionSignal" />
     <SIGNAL class="IsSaleTransactionSignal" />
     <SIGNAL class="IsNotVoidTransactionSignal" />
     <SIGNAL class="IsAuthNotRequiredSignal" />
     <SIGNAL class="MBSIsAuthorizationRequiredSignal"
package="com.mbs.pos.services.tender" />
     <SIGNAL class="IsRemoveTenderSignal" />
     <SIGNAL class="IsNoRemoveTenderSignal" />
     <SIGNAL  class="IsValidDriverLicenseSignal" />
     <SIGNAL  class="IsInvalidDriverLicenseSignal" />
... more declarations omitted...
</SERVICECODE>
... code omitted...
<ROAD name="AuthorizationRequested"
                     letter="Next"
                     destination="AuthorizationStation"
                     tape="ADVANCE"
                     record="OFF"
                     index="OFF">
                    <LIGHT signal="MBSIsAuthorizationRequiredSignal"/>
```

Cargo

Since cargos do not mail letters, they may be extended or replaced. Cargo classes are typically part of a hierarchy of classes. Perform these steps:

1. Modify the cargo class by doing one of the following:
   - To extend the cargo, create a new class in the customer source tree that extends the cargo in the product source tree. Be sure to extend from the lowest-level subclass. Add the customer abbreviation to the beginning of the filename.
   - To replace the cargo, create a new cargo class in the customer source tree.
2. Edit the appropriate Tour Map for the locale, using the replacewith property in the <CARGO> tag to define the new package and filename for the cargo.

Example 5–7 tourmap_CA.xml: Replacing a Cargo

```
<tour name="tender”>
       <file>classpath://com/mbs/pos/services/tender</file>
       <SITE
          name="RefundDueUI"
          useaction="com.mbs.pos.services.tender.refundDueUISite">"/>
       <CARGO
          class="TenderCargo"
          replacewith="com.mbs.pos.services.tender.MBSTenderCargo"/>

    </tour>
```

3. Modify the tour map and/or tour scripts and shuttles of the calling and nested tours to adapt to the cargo modifications. Be sure to address:
   - Classes in the same tour as the modified cargo
   - All tours for which this tour is a nested tour

- All tours which are called by this tour

UI Framework

The UIManager and UITechnician classes are provided by Foundation. They are configurable through the Conduit Script and should not be modified directly. This section describes customization to the default UI configuration and individual screens.

Default UI Config

The product file <source_directory>\applications\pos\src\oracle\retail\stores\pos\config\defaults\defaultuicfg.xml contains the building blocks for the UI (displays, templates and specs) and references to all tour-specific uicfg.xml files. If you change any UI script in the customer implementation, the defaultuicfg.xml file must be replaced. It also needs to be replaced if the displays, templates, and basic bean specs need to be replaced. Perform these steps to replace the file:

1. Copy the file defaultuicfg.xml to the pos\config\defaults directory in the customer source tree, and rename it (for example, to MBSdefaultuicfg.xml).
2. Modify the displays, templates, default screens, and specs as necessary to represent the customer's user interface.
3. Verify that the conduit script for the client tier has been customized and is located in the customer source tree.
4. Modify the client conduit script to include the new filename and package name for the MBSdefaultuicfg.xml file, in the configFilename property value in the UISubsystem section of the UITechnician tag.

Example 5–8 ClientConduit.xml: Conduit script modified to use custom UI configuration file

```
<TECHNICIAN
        name="UITechnician"
        class="UITechnician"
        package="oracle.retail.stores.foundation.manager.gui" export="Y">

        <CLASS
            name="UISubsystem"
            package="oracle.retail.stores.pos.ui"
            class="POSJFCUISubsystem">

            <CLASSPROPERTY
                    propname="configFilename"

propvalue="classpath://com/mbs/pos/config/defaults/MBSdefaultuicfg.xml"
                    proptype="STRING"/>
...additional class properties omitted...
</CLASS>
</TECHNICIAN>
```

UI Script

A UI script changes if the overlays or unique bean specifications of one or more screens in a tour need to be modified. Perform these steps:

1. Create a new UI script in the customer source tree. For example, copy the tenderuicfg.xml file from the product source tree to the customer source tree and rename it MBStenderuicfg.xml.

2. Modify the MBSdefaultuicfg.xml file in the customer source tree to refer to the new filename and package for the UI script.

Example 5–9 MBSdefaultuicfg.xml: Customized Default UI Configuration File

```
... other include statements omitted...
<INCLUDE
filename="classpath://oracle/retail/stores/pos/services/sale/saleuicfg.xml"/>
<INCLUDE filename="classpath://com/mbs/pos/services/tender/MBStenderuicfg.xml"/>
<INCLUDE
filename="classpath://oracle/retail/stores/pos/services/tender/capturecustomerinfo
/capturecustomerinfouicfg.xml"/>
<INCLUDE
filename="classpath://com/extendyourstore/pos/services/inquiry/inquiryoptionsuicfg
.xml"/>
... other include statements omitted...
```

Bean Model and Bean

The Point-of-Service product code provides generalized beans that are designed to be reused as-is, such as GlobalNavigationButtonBean.java for the global navigation button bar and DataInputBean.java for the work area of form layout screens. These classes are not intended to be extended for a specific implementation, though they may be extended if the general behavior or data must change in all cases.

The classes can be used for different screens within the application without changing to Java code by modifying parameter values and calling methods on the bean. Use the generalized beans whenever possible and avoid beans specialized for only one screen. However, bean and bean model classes in the product code that are specific to an individual screen, such as CheckEntryBean.java and CheckEntryBeanModel.java, may be customized. Perform these steps to modify a bean model:

1. Create a new bean model class.

   Create a new class in the customer source tree, and add the customer abbreviation to the filename.

2. Copy tour files that need to reference the new bean model into the customer source tree. Modify them to create and manipulate data for the new bean model.

Perform these steps to modify the bean:

1. Create a new bean class.

   Create a new class in the customer source tree, and add the customer abbreviation to the filename.

2. Modify the UI config scripts that reference the bean class in the customer source tree to refer to the new bean class filename and package.

Example 5–10 MBStenderuicfg.xml: Tender UI Configuration with Customized Bean Reference

```
<UICFG>

    <BEAN
        specName="TenderOptionsButtonSpec"
        configuratorPackage="oracle.retail.stores.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="com.mbs.pos.ui.beans"
        beanClassName="MBSNavigationButtonBean">
```

```
        <BUTTON
            actionName="Cash"
            enabled="true"
            keyName="F2"
            labelTag="Cash"/>
...other buttons omitted...
     </BEAN>
...other UI objects omitted...
</UICFG>
```

Other

This section covers customization of components other than the tour and the UI framework, including internationalization and localization changes as well as conduit scripts, PLAF, receipts, and reports.

Internationalization

For more information about Internationalization of Point-of-Service, see Internationalization.

> **Note:** For internationalization, Point-of-Service can use multiple languages at any given time: either default language, user language or customer preferred language. Point-of-Service looks up resource bundles based on the selected language (store default language, user language or preferred customer language) plus the region from its default locale. If Point-of-Service cannot find this combination, it reverts back to the selected language only.

For additional internationalization support of Oracle Retail Point-of-Service, please contact Oracle Retail Services.

Conduit Scripts

The conduit scripts provided with Oracle Retail applications define a typical tier configuration and are usually replaced with customer conduit scripts for a given implementation. Conduit scripts include an XML file and a .bat and .sh file to execute the XML; both .bat and .sh versions of the batch file are provided to support Windows and Linux.

Perform these steps to set up customer conduit scripts:

1. Copy the conduit scripts (client and server) to the customer source tree.

   Copy the XML and .bat and .sh files for each type of conduit script. Rename the scripts using the customer abbreviation (ClientConduit.xml becomes MBSClientConduit.xml).

2. Edit each XML file to include only the managers and technicians that should be loaded on the given tier.

3. Modify the class and package names for any managers, technicians and configuration scripts that have been customized.

```
MBSClientConduit.xml: Customized with New Data Manager
    <MANAGER name="DataManager" class="MBSDataManager"
             package="com.mbs.foundation.manager.data">
        <PROPERTY propname="configScript"
                  propvalue="classpath://config/manager/PosDataManager.xml" />
```

```
</MANAGER>
```

4. Modify your development environment to pass in the new conduit XML file as a parameter to the TierLoader.
5. Edit the .bat and .sh files to pass the correct conduit XML files to the Java environment.

PLAF

Point-of-Service implements a pluggable look-and-feel (PLAF) so that customers may modify the look of the application including screen colors and images. To modify the PLAF, follow these steps:

1. Create a new properties file that is a copy of one of the following files. Place the file in the com\mbs\pos\config directory in the customer source tree.
   - swanplaf.properties — yellow-and-purple, text-based LAF
   - imagePlaf.properties — blue and gold image-based LAF

   Note: Each supported language has its own version of swanplaf.properties and imagePlaf.properties:

   swanplaf_fr.properties

   imagePlaf_fr.properties
2. Update the conduit scripts in the customer source tree to specify the package and filename for the new LAF file in the UI Technician tag.
3. Have new UI beans call uiFactory.configureUIComponent(this, UI_PREFIX) in the initialize() method to set the look-and-feel.

Reports

Point-of-Service has a set of reports that print on the slip printer. The report document names are specified in the tour code.

To modify existing Point-of-Service reports and receipts, the report Java files can be extended. Perform these steps:

1. For each report or receipt, do one of the following:
   - To modify an existing report or receipt, copy existing BPT and change file name in BlueprintedDocumentManager.xml.
2. Create, modify or override data and methods as necessary to modify the report or receipt.

Creating new receipts (BPT and SER)

When it is necessary to have the Point-of-Service client print a new receipt, develop a blueprint (BPT) file for the receipt. However, it is extremely difficult to do this without a serialized instance of the object being printed prior, to aid in editing the blueprint. Since the BlueprintedDocumentManager has an option to serialize the parameter beans, you can use the Point-of-Service client to serialize what is printed for you. You will then have the serialized object file and then can use it to drag and drop method calls onto your new blueprint.

1. Ensure /pos/config/manager/BlueprintedDocumentManager.xml has the persistBeansAsDataObject value set to true.

2. Edit /pos/config/manager/BlueprintedDocumentManager.xml to contain a <*RECEIPT*> element for your new receipt.
3. Enter code in a Point-of-Service site to print your receipt. For example:

```
PrintableDocumentManagerIfc pdm =
(PrintableDocumentManagerIfc)bus.getManager(PrintableDocumentManagerIfc.TYPE);
ReceiptParameterBeanIfc bean = getMyParameterBean()*;
bean.setDocumentType("MyNewReceipt");
pdm.printReceipt((SessionBusIfc)bus, bean);
```

> **Note:** There are many ways to construct a PrintableDocumentParameterBeanIfc for printing. You can potentially use a method from PrintableDocumentManagerIfc or you can instantiate or use a subclass of PrintableDocumentParameterBean to create one of your own.

4. Run the code and navigate through the Point-of-Service site.

   A MyNewReceipt.ser file will appear in the /pos/receipts dir.

> **Caution:** It is likely the Point-of-Service will fail here because the desired receipt bpt will not exist.

5. In Eclipse, go to **File > New > Other > type**. Select **Receipt**.

   You can also go to **Oracle-Retail > Receipt Blueprint**.
6. Name your file **MyNewReceipt.bpt**.
7. Attach MyNewReceipt.ser to the Receipt Data Object view.

> **Note:** For more information, see the *Oracle Retail Point-of-Service Receipt Builder Tool User Guide*.
>
> This document is available through My Oracle Support. Access My Oracle Support at the following URL:
>
> https://support.oracle.com
>
> *Oracle Retail Point-of-Service Receipt Builder Tool User Guide* (Doc ID: 1595733.1)

Alternate Bean Creation (SER) As an alternative to using Point-of-Service to create the serialized file, it is possible to use the Object Inspector plugin to create an instance of your receipt parameter bean if it has a zero-arg constructor. This may be more convenient than running Point-of-Service, but the resulting serialized object does not contain any values to use when printing. You can use Object Inspector to execute simple set methods on your bean instance. Be sure your class implements PrintableDocumentParameterBeanIfc.

1. In Eclipse, go to **File > New > Other > type** and select **Object**.

   You can also go to **Java > Serialized Object**.
2. Name your file **MyNewReceipt.ser** or a name that matches the name of the .bpt this file is used with.

3. Pick the class that implements PrintableDocumentParameterBeanIfc.
4. Click **Finish**.
5. Edit the value of this object as desired.
6. Open the Receipt Builder editor and attach your new file.

Note: For more information, see the *Oracle Retail Point-of-Service Receipt Builder Tool User Guide*.

This document is available through My Oracle Support. Access My Oracle Support at the following URL:

https://support.oracle.com

Oracle Retail Point-of-Service Receipt Builder Tool User Guide (Doc ID: 1595733.1)

Using Person.java to Create a Receipt If you are using the sample class provided in Step 3. to create a receipt, the following steps create a serialized file and blueprint with which to print. In Eclipse, do the following to create a data object and blueprint using the Person.java downloaded from the link in Step 3.

1. If no Java Project exists, create one. Go to **File > New > Java Project**.
2. Name your project , such as **receiptbuilder**. The default settings will be fine.
3. Click **Finish**.
4. Access the project and select the **src** folder.
5. Go to **File > New > Package** and create a package named **com.demo**.
6. Right-click on the com.demo package and select **Properties**.
7. Using Windows Explorer, go to the com.demo package directory and Paste Person.java into the package.
8. Right-click on the com.demo package and select **Refresh** to make Person.java appear in Eclipse.
9. Right-click the com.demo package and go to **New > Folder** and create a folder called **objects**.
10. Right-click on the objects folder and go to **New > Other > Java > Serialized Object**.
11. Click **Next**.
12. Click **Browse**.
13. Type **Person**. Click **OK** to select com.demo.Person class.
14. Click **Finish**.
15. In a new editor window, go to **com.demo.Person > Person > init()**. This will provide some values.
16. Go to setBirthDate(Date). Specify a date and click **OK**.
17. Go to getBirthDate() to see your value.
18. Click **Save**.

19. Right-click on the project and go to **New > Folder** and create a folder called **receipts**.
20. Right-click on receipts folder and go to **New > Other... > Oracle-Retail > Receipt Blueprint**.
21. Click **Next**.
22. Click **Finish**.
23. Attach the serialized Person.java to the Receipt Data Object view. Do this by clicking **Attach RDO** button or drop-down menu option.
24. Type **object.ser**. Select **OK** to attach the object.
25. Rename the element with the word **text** in it to **Name:**
    - Do this by double-clicking the element and editing directly, or select the element and in the Properties view, edit th on the General tab.
26. Drag Person.getName() from the RDO view to the right of **Name:**.
27. Go to **com.demo.Person > Person > getName()**. Select this item.
28. Drag this tree item to the right of the Name: element and drop it.
29. Click **Print**. The output should look like this:

```
----------------------------------------
Blueprint@12291679
----------------------------------------
Name: John Smith
```

30. Click **Save**.
31. Continue to edit the receipt with the palette on the right side of the editor.
32. Create a receipt with output that looks like the following:

```
----------------------------------------
Blueprint@12291679
----------------------------------------
(8/13/08)                          11:35 AM
Id: 1234                Name: John Smith
Height: 0.0
Age: Voided 12,346
Sex: male
Salary: 80,000.00
Nick name: "Johnny"
Birth Date: Jul 18, 2008
Spouse: Sara Smith
 ________________________________________

Relatives
 ---------------------------------------
    Name: Joe Smith
     Age: 84E0
    Name: Sally Smith
    Name: Sandy Smith
```

Domain Package

This section addresses customization of files in the domain package. The domain package can be found in the `\OracleRetailStore\domain` directory in your source control system.

Retail Domain

The Retail Domain provides a retail-specific implementation of business objects. These objects are easily extended to meet customer's requirements.

DomainObjectFactory

If any Retail Domain Objects (RDOs) are added or extended, the DomainObject Factory must be extended. This needs to be done only one time for the application. The extended class must include getXinstance() methods for all new and extended RDOs, where X is the name of the RDO. Perform these steps:

1. Create a new Java class that extends DomainObjectFactory.java. It should be named with the customer abbreviation in the filename MBSDomainObjectFactory.java and be located in the customer source tree.
2. Copy the domain.properties file to the domain\config directory of the customer source tree. Modify the setting for the DomainObjectFactory to refer to the new package and class name created in the previous step.

   ```
   DomainObjectFactory=com.acmebrick.domain.factory.MBSDomainObjectFactory;
   ```

3. Add getXInstance() methods as necessary for new Retail Domain Objects.

Retail Domain Object (RDO)

Perform these steps to create or extend an RDO:

1. Complete one of the following steps:
   - To create a new RDO, create a Java class in the customer source tree in the appropriate subdirectory of domain\src\com\mbs\domain. Extend an appropriate superclass from the product code. At a minimum, the new class must extend EYSDomainIfc.java.
   - To modify an existing RDO, create a Java class in the customer source tree that extends an RDO in the product code.

   Include the customer abbreviation in the filename; for example, you might name your class file MBSCustomer.java.
2. Add data attributes and methods required by the customer-specific functionality.
3. Create setCloneAttributes(), equals() and toString() methods to address the new data attributes and then reference the corresponding superclass method.
4. Complete one of the following steps:
   - For a new RDO, add a new getXInstance() method to MBSDomainObjectFactory.java for the new RDO.
   - For an extended RDO, override the existing getXInstance() method in MBSDomainObjectFactory.java to return an object of the new class type.
5. Access the new RDO data and methods from tours located in the customer source tree. If product tours need to access the new RDO data and methods, the tours must be modified.

6. If the RDO data is represented on a screen, modify the UI script, bean and bean model classes to reflect the change.
7. If the RDO is saved to the database, modify the data operation to save the new data attributes.

Database

This section details how to extend database behavior through changes to the data operations. The architecture of the Data Technician simplifies this somewhat, because changes to data operations can be implemented without changes to the Point-of-Service application code.

Data Manager and Technician Scripts

The Data Manager and Data Technician Scripts, PosDataManager.xml and DefaultDataTechnician.xml, are routinely customized when transactions, data actions, and data operations are customized. See the next section for details.

Data Actions and Operations

When a new or modified RDO contains data that need to be saved to the database, a data operation class must be created or extended. A Data Action must be modified if a unit of database work is changed.

1. Create class files.

   Create new class files for each new or modified item in the customer source tree. If an item extends a product class, add the customer abbreviation to the filename.
2. If a customized version of PosDataManager.xml does not already exist, copy it to the customer source tree and give it a new name, such as MBSPosDataManager.xml.
3. For customized transactions with new filenames, modify the transaction name.
4. If a customized version of DefaultDataTechnician.xml does not already exist, copy it to the customer source tree and give it a new name, such as MBSDefaultDataTechnician.xml.
5. Edit the customized MBSDefaultDataTechnician.xml file, updating package and class names for data actions and data operations that have been modified.

Example 5–11 MBSDefaultDataTechnician.xml: Customizing a Data Operation

```
<OPERATION class="JdbcSaveTenderLineItems"
       package="com.mbs.domain.arts"
       name="MBSSaveTenderLineItems">
       <COMMENT>
           This operation saves all tender line items associated
           with the transaction.
       </COMMENT>
   </OPERATION>
```

6. Modify the conduit scripts to reference the new package and/or filename of the technician script.

Example 5–12 ClientConduit.xml: Customizing the Data Technician

```
<TECHNICIAN name="LocalDT" class="DataTechnician"
               package="com.mbs.foundation.manager.data"
```

```
            export="Y">
        <PROPERTY
            propname="dataScript"
            propvalue="classpath://config/manager/MBSDefautlDataTechnician.xml"
        />
    </TECHNICIAN>
```

Data Transactions

Data transactions are the valet classes that carry requests from the client to the server. A data transaction factory implements the factory pattern for data transaction classes. The application code asks the factory for a transaction object and the factory determines which Java class is used to create the object. To create or extend a data transaction class, follow these steps:

1. Create new or modified data transactions.

   Create a Java class in the customer source tree and prepend the customer abbreviation to the filename. If you are modifying an existing transaction, have the class extend the transaction class in the product code, and overwrite the methods you are modifying.

2. Copy POSDataManager.xml to the customer source tree.
3. For customized transactions with new filenames, modify the transaction name.
4. Copy DefaultDataTechnician.xml to the customer source tree.
5. Modify package and class names for data actions and data operations that have been modified.
6. If not already done, modify the conduit scripts to reference the new package and/or filename of the technician script.
7. Extend DataTransactionKeys.java as MBSDataTransactionKeys.java in the customer source tree to add or modify the static final String for each transaction (the file serves as a list of string constants).

Example 5–13 MBSDataTransactionKeys.java: Adding Strings

```
public static final String DATA_MAINTENANCE_TRANSACTION=”data.transaction.DATA_
MAINTENANCE_TRANSACTION
public static final String PLU_RETURN_TRANSACTION” =data.transaction.PLU_RETURN_
TRANSACTION”
```

8. Update domain.properties in the customer source tree to add or modify the name/value pairs for each transaction.

Example 5–14 domain.properties: Sample Modified and New Data Transactions

```
# Registry of DataTransactionIfc implementations
# (try to keep in alphabetical order)
#

data.transaction.ADVANCED_PRICING_DATA_
TRANSACTION=oracle.retail.stores.domain.arts.AdvancedPricingDataTransaction
...code omitted here...
data.transaction.REGISTER_STATUS_
TRANSACTION=com.MBS.domain.data.transactions.RegisterStatusTransaction
data.transaction.REGISTRY_DATA_
TRANSACTION=oracle.retail.stores.domain.arts.RegistryDataTransaction
data.transaction.STORE_LOOKUP_DATA_
```

```
TRANSACTION=com.MBS.domain.data.transactions.StoreLookupDataTransaction


MBSdata.transaction.DATA_MAINTENANCE_
TRANSACTION=com.MBS.domain.data.transactions.DataMaintenanceTransaction
MBSdata.transaction.PLU_RETURN_
TRANSACTION=com.MBS.domain.data.transactions.ReturnPluTransaction
```

6

Back Office and Central Office Extension Guidelines

This chapter describes the various extension mechanisms available in the Commerce Services framework. There are multiple forces driving each extension that determine the correct strategy in each case.

The product has four distinct layers of logic:

UI layer
Struts/Tiles implementation utilizing Actions for processing UI requests and JSP pages with Tiles providing layout.

Application Manager
Session facade for the UI (or external system) that models application business methods. May or may not be reusable between applications. Provides for remote accessibility.

Commerce Service
Session facade for the service that models coarse-grained business logic that should be reusable between applications.

Persistence
Entity beans that are fine-grained, and data access objects (DAO), consumed by the service. The entities are local to the service that controls them.

Note: See the *Oracle Retail POS Suite Security Guide* for more information about specific security features and implementation guidelines for the POS Suite products.

Audience

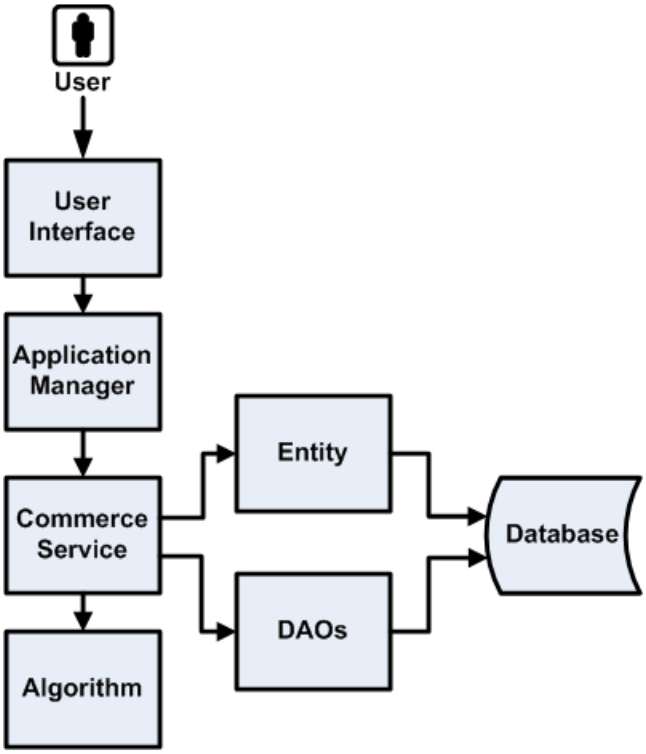
This section provides guidelines for extending the Oracle Retail Enterprise applications. The guidelines are designed for three audiences:

- Members of customer architecture and design groups can use this chapter as the basis for their analysis of the overall extension of the systems.
- Members of Oracle Retail's Technology and Architecture Group can use this chapter as the basis for analyzing the viability of the overall extension strategy for enterprise applications.
- Developers on the project teams can use this chapter as a reference for code-level design and extension of the product for the solution that is released.

Application Layers

Figure 6–1 describes the general composition of the enterprise applications. The following sections describe the purpose and responsibility of each layer.

Figure 6–1 Application Layers



User Interface

The user interface (UI) framework consists of Struts Actions, Forms, and Tiles, along with Java server pages (JSPs).

- Struts configuration
- Tiles definition
- Cascading style sheets (CSS)
- JSP pages
- Resource bundles for internationalization (I18N)

Application Manager

The Application Manager components are coarse-grained business objects that define the behavior of related Commerce Services based on the application context.

- Session beans
- View beans for the UI

Commerce Service

A commerce service is a fine grained component of reusable business logic.

- Session beans
- Data transfer objects (DTO)

Algorithm

An SPI-like interface defined to enable more fine-grained pieces of business functionality to be replaced without impacting overall application logic.

Entity

Fine-grained entity beans owned by the commerce service. The current strategy for creating entity beans in the commerce service layer is BMP.

Data Access Objects

Provides an abstract interface to the database, providing specific operations without exposing details of the database.

Database

The Oracle Retail enterprise applications support the ARTS standard database schema. The same tables referenced by Central Office and Back Office are a superset of the tables that support Point-of-Service.

Extension and Customization Scenarios

The following are extension and customization scenarios.

Style and Appearance Changes

This should only present minor changes to the UI layer of the application. These types of changes, while extremely common, should represent minimal impact to the operation of the product. Typical changes could be altering the style of the application (fonts/colors/formatting) or the types of messages that are displayed.

Application impact:

- Struts configuration (flow)
- Tile definition
- Style sheet
- Minor JSP changes, such as moving fields
- Changing static text through resource bundles

Additional Information Presented to User

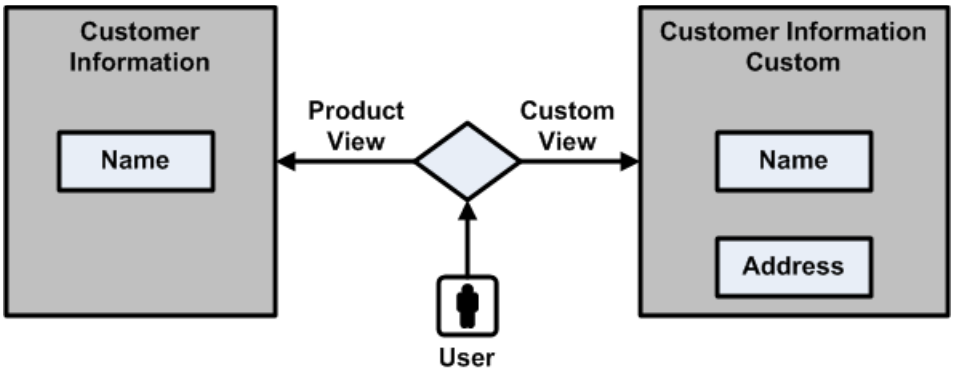
This is one of the more common extensions to the base product: enabling the full life cycle management of information required by a particular customer that is not represented in the base product.

If the information is simply presented and persisted then we can choose a strategy that simply updates the UI and persistence layers and passes the additional information through the service layer.

However, if the application must use the additional information to alter the business logic of a service, then each layer of the application must be modified accordingly.

This scenario generally causes the most pervasive changes to the system; it should be handled in a manner that can preserve an upgrade path.

Figure 6–2 Managing Additional Information



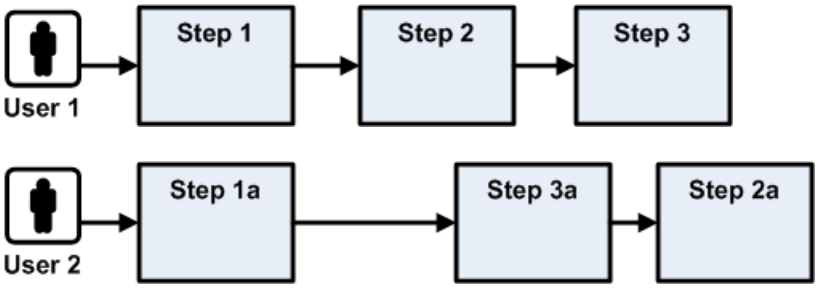
Application impact for this change:

- JSP pages
- View beans
- Struts configuration
- UI actions
- UI forms
- Application manager
- Commerce service
- Entity or DAO
- Database schema

Changes to Application Flow

Sometimes a multi-step application flow can be rearranged or customized without altering the layers of the application outside of the UI. These changes can be accomplished by changing the flow of screens with the struts configuration.

Figure 6–3 Changing Application Flow



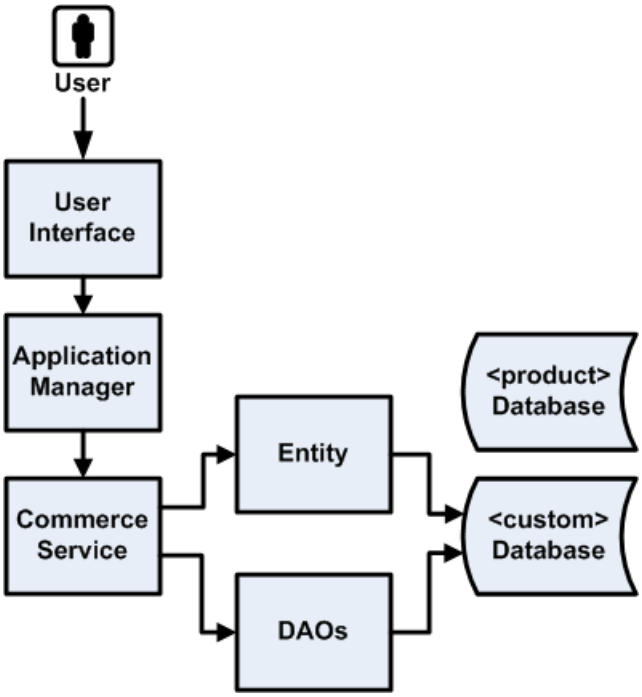
Application impact for this change:

- Struts configuration

Access Data From a Different Database

This customization describes accessing the same business data from a different database schema. No new fields are added or joined unless for deriving existing interface values. This scenario would most likely not be found isolated from the other scenarios.

Figure 6–4 Accessing Data from a Different Database



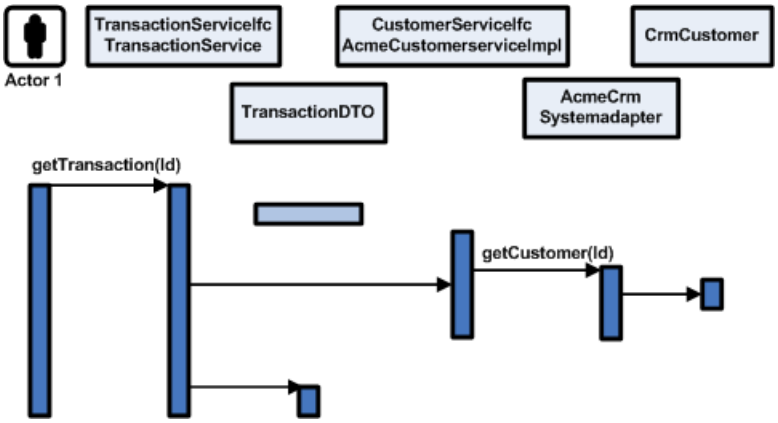
Application impact for this change:

- Entity beans and DAO
- Database schema

Access Data From External System

This customization involves replacing an entire Commerce Service with a completely new implementation that accesses an external system.

Figure 6–5 Accessing Data from an External System



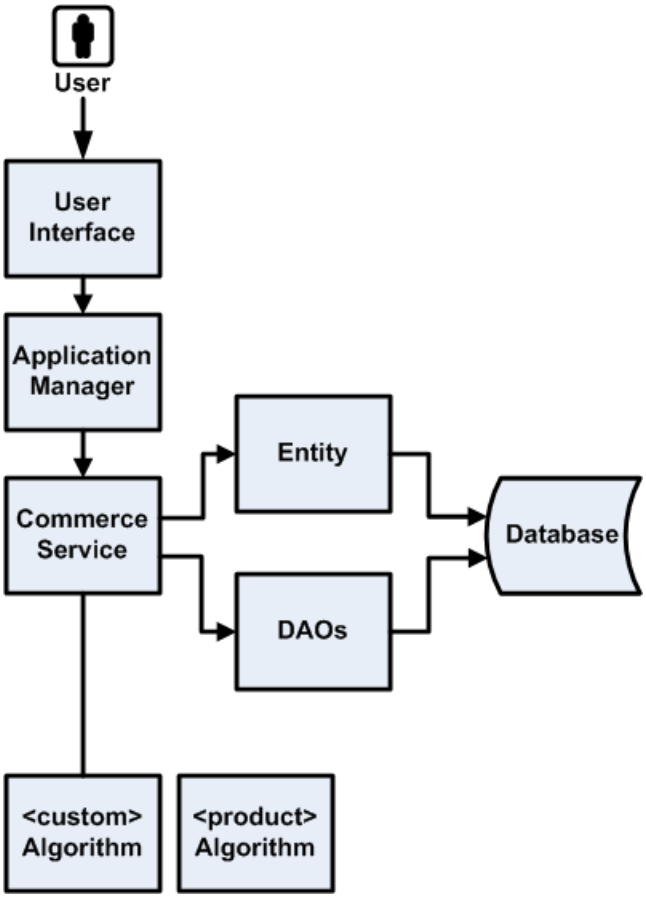
Application impact for this change:

- Deployment Configuration – replacing Commerce Service implementation with custom implementation.

Change an Algorithm Used By a Service

Assuming the UI is held constant, but values such as net totals or other attributes are derived with different calculations, it is advantageous to replace simply the algorithm in question, as the logic flow through the current service does not change.

Figure 6–6 Application Layers



Application impact for this change:

- Algorithm
- Application configuration

Extension Strategies

Refer to Figure 6–7 as a subset of classes for comparison purposes.

Figure 6–7 Sample Classes for Extension—Entity Bean

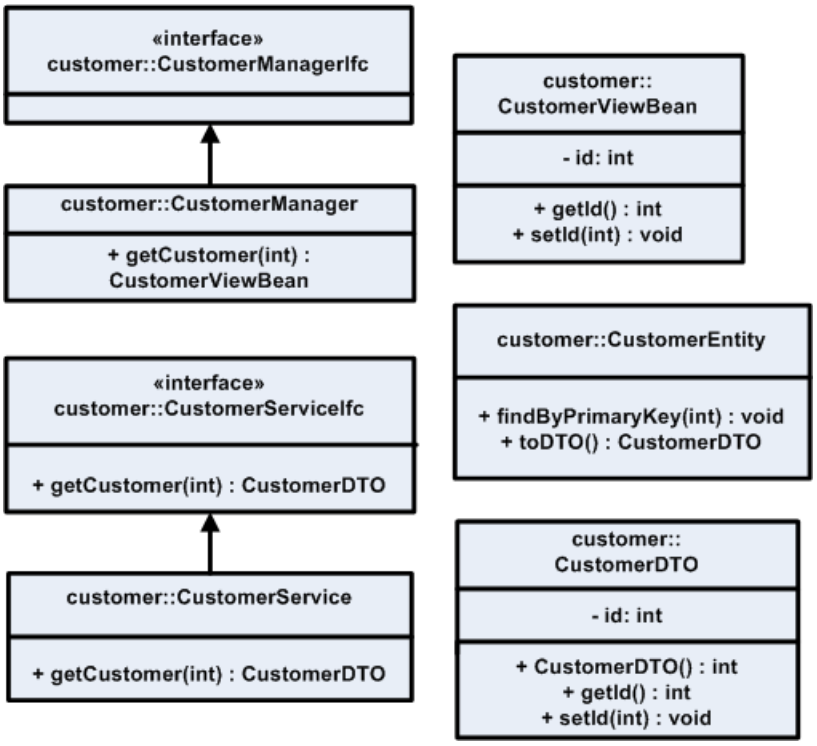
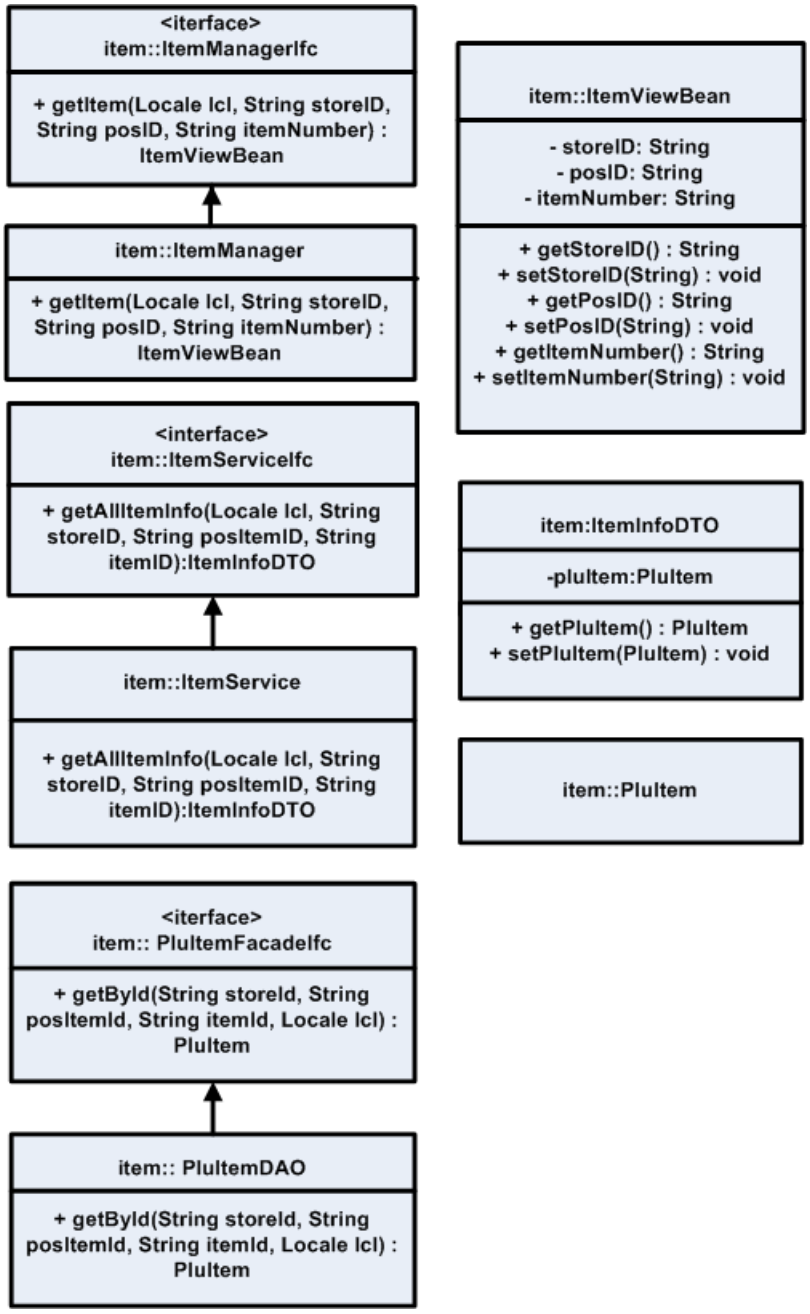


Figure 6–8 Sample Classes for Extension—DAO



Extension with Inheritance

This strategy involves changing the interfaces of the service itself, perhaps to include a new finder strategy or data items unique to a particular implementation. For instance, if the customer information contained in base product does not contain data relevant to the implementation, call it CustomField1.

All of the product code would be extended (the service interface, the implementation, the DTO and view beans utilized by the service, the UI layers and the application manager interface and implementation) to handle access to the new field.

Figure 6–9 Extension with Inheritance: Class Diagram—Entity Bean

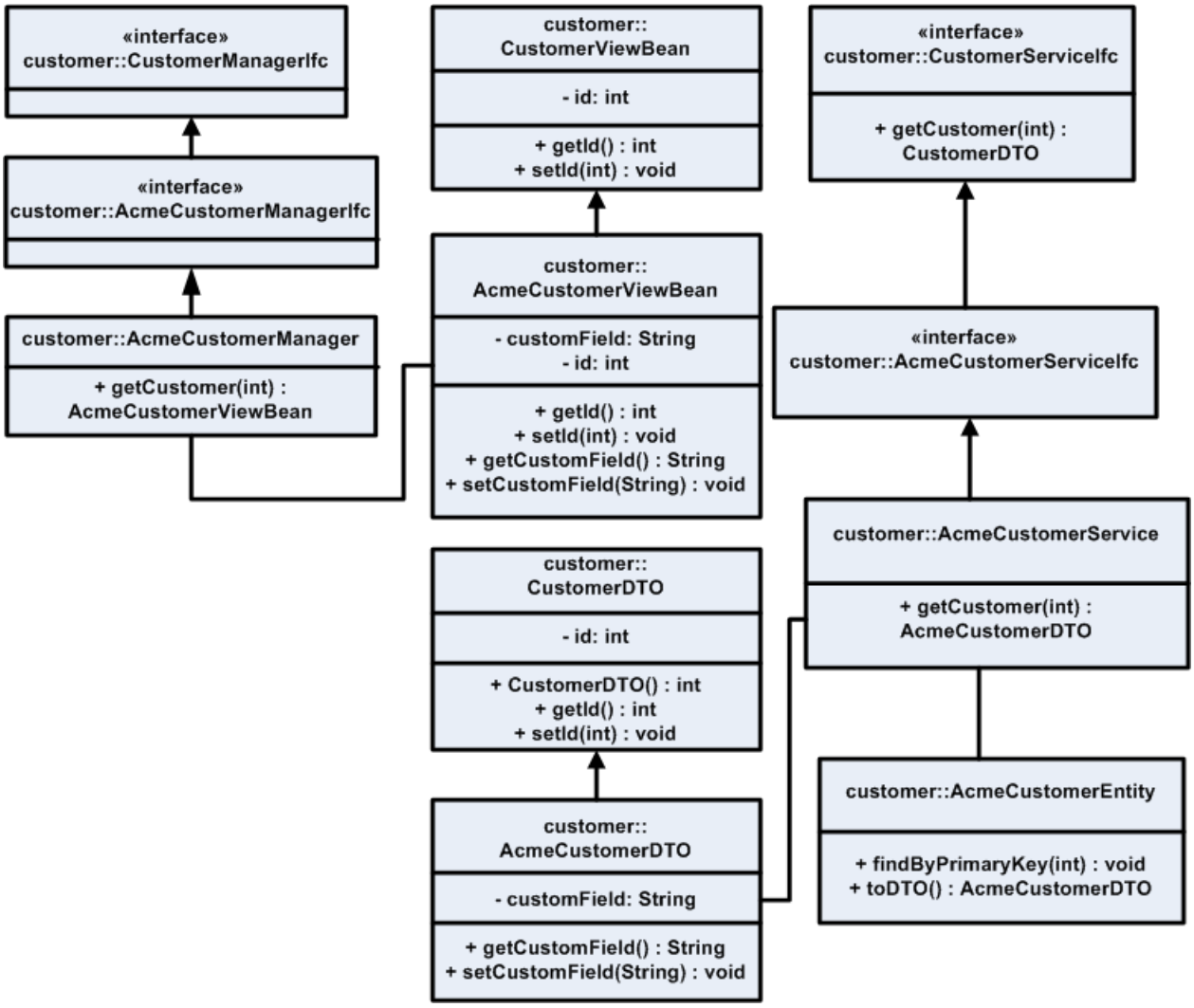
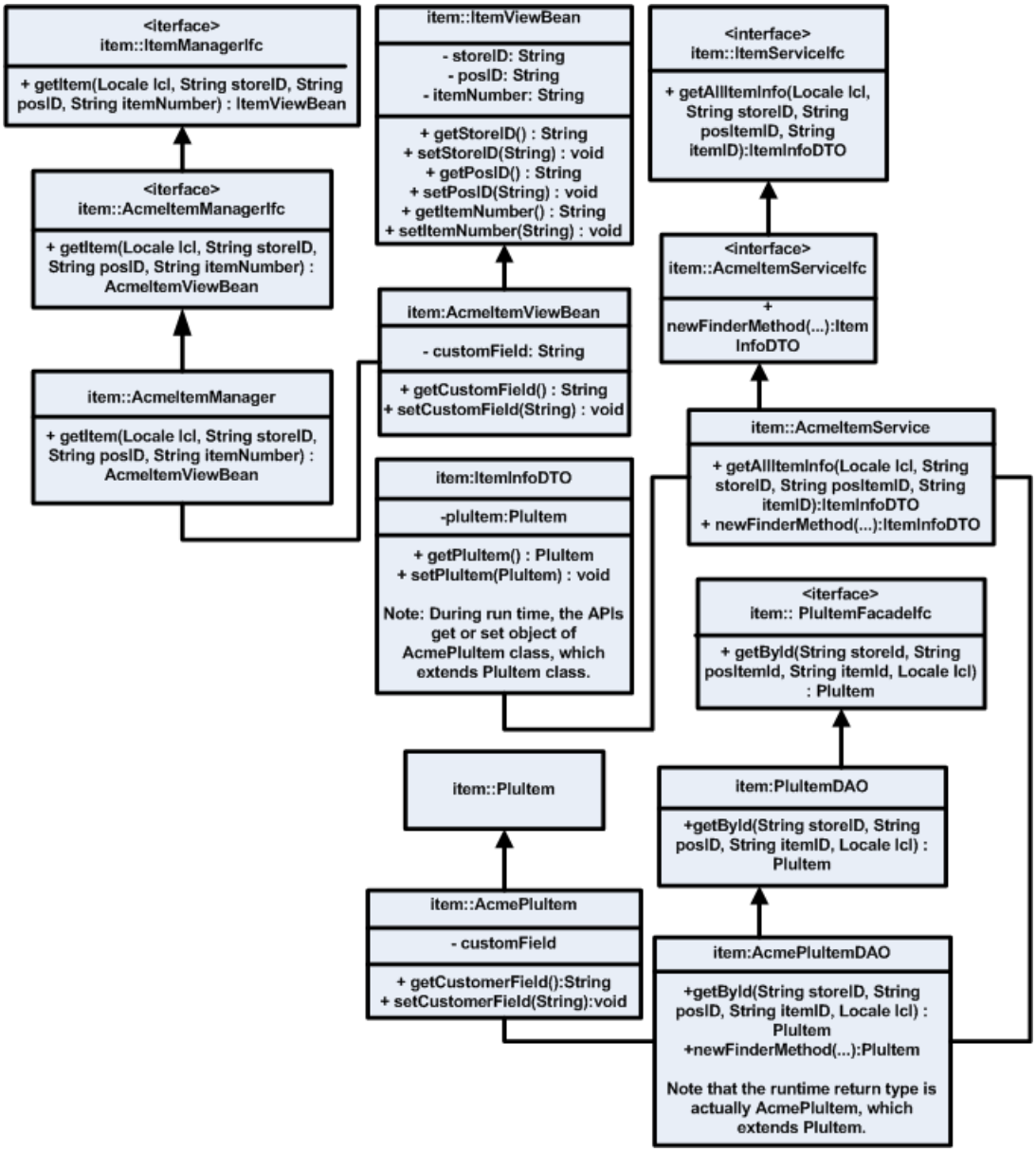


Figure 6–10 Extension with Inheritance: Class Diagram—DAO



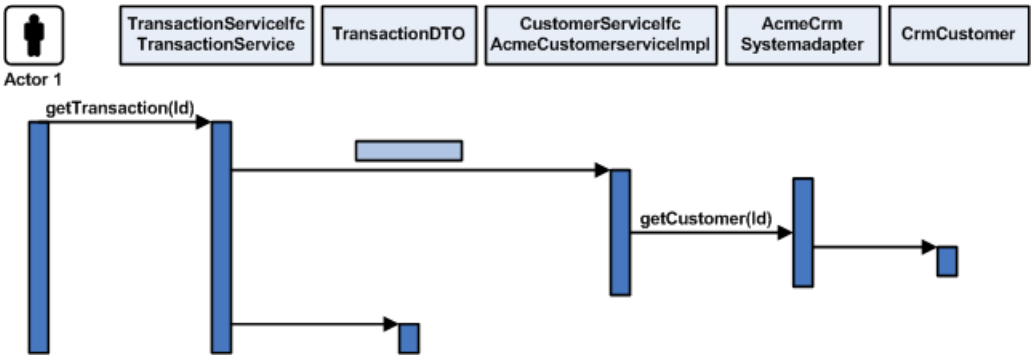
Replacement of Implementation

This strategy involves keeping the existing product interfaces to the service intact, but utilizing a new implementation. This strategy is suggested for when the entire persistence layer for a particular service is changed or delegated to an existing system.

Figure 6–11 demonstrates the replacement of the product Customer Service implementation with an adapter that delegates to an existing CRM solution (the system of record for customer information for the retailer).

This provides access to the data from the existing services that depend on the service interface.

Figure 6–11 Replacement of Implementation



Service Extension with Composition

This method is preferred adding features and data to the base product configuration. This is done with composition instead of inheritance.

For specific instances when you need more information from a service that the base product provides, and you wish to control application behavior in the service layer, it is suggested to use this extension strategy. The composition approach to code reuse provides stronger encapsulation than inheritance. Using this method keeps explicit reference to the extended data/operations in the code that needs this information. Also, the new service contains rather than extends the base product. This allows for less coupling of the custom extension to the implementation of the base product.

Figure 6–12 Extension with Composition: Class Diagram—Entity Bean

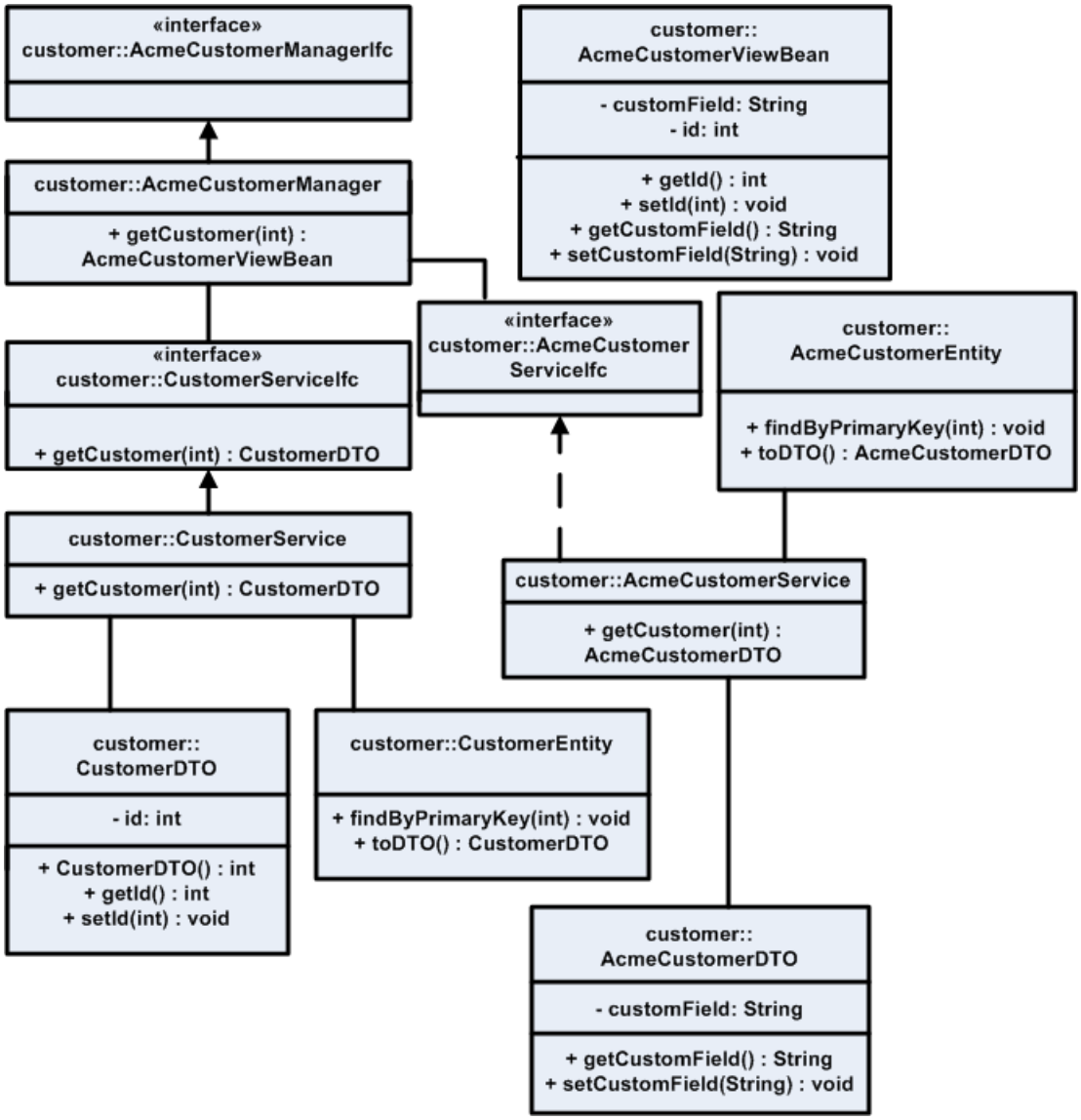


Figure 6–13 Extension with Composition: Class Diagram—DAO

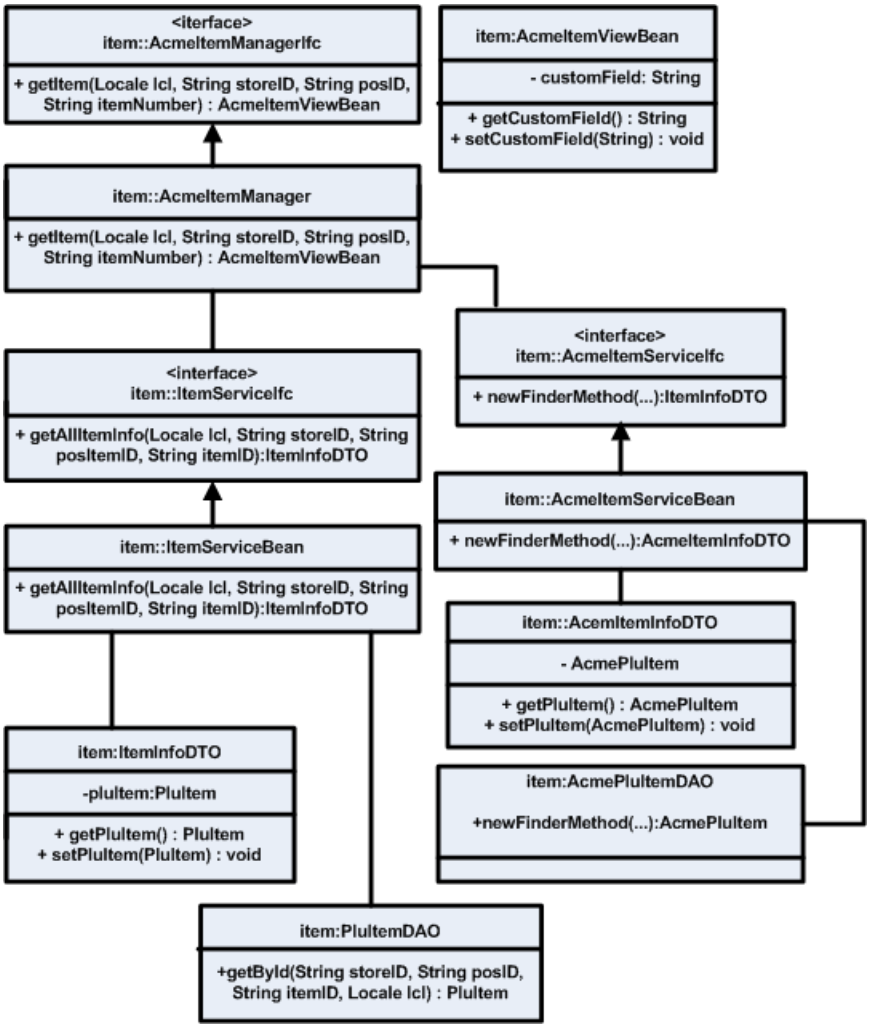
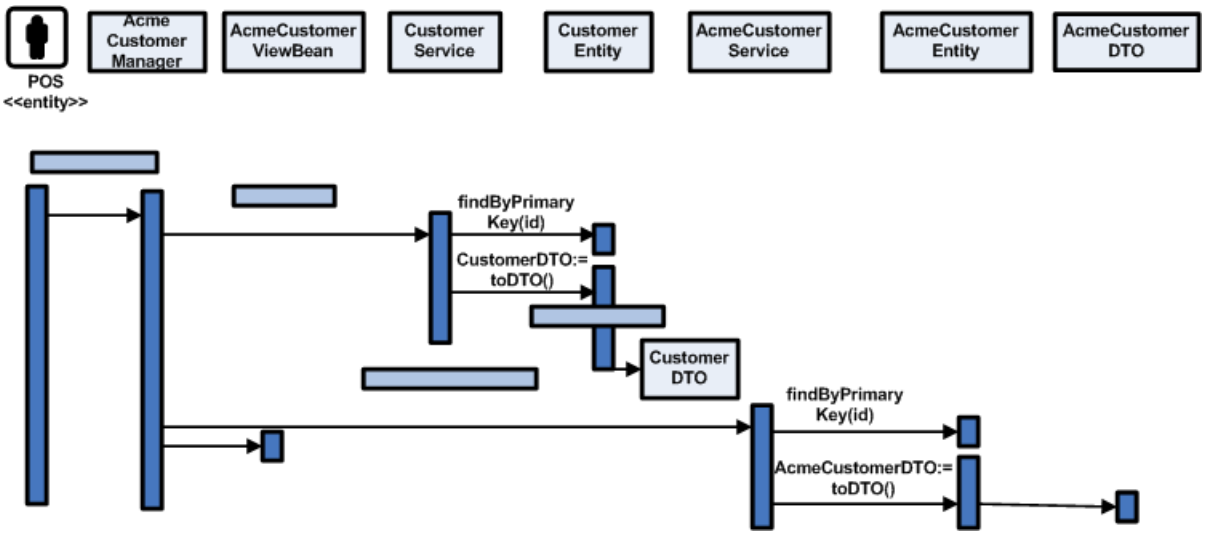


Figure 6–14 Extension Composition



Data Extension Through Composition

This strategy describes having the entity layer or DAO layer take responsibility for mapping extra fields to the database by aggregating the custom information and passing it through the service layer. This approach assumes that the extra data is presented to the user of the system and persisted to the database, but is not involved in any service layer business logic.

This scenario alters the UI layer (JSP/Action/ViewBean) and adds a new ApplicationManager method to call assemble the ViewBean from the extensible DTO provided by the replaced Entity bean or DAO.

Slight modifications to the Service session bean might be necessary to support the toDTO() and fromDTO (ExtensibleDTOIfc dto) methods on the Entity bean, depending on base product support of extensions on the particular entity bean.

1. Create the new ApplicationManager session facade.
2. Create the new ViewBeans required of the UI.
3. Create a new Entity bean or DAO that references the original data to construct a base product DTO that additionally contains the custom data using the extensible DTO pattern.
4. Create a new DTO based on the extensible DTO pattern.
5. Create new JSP pages to reference the additional data.
6. Change the deployment descriptors that describe which implementation to use for a particular Entity bean. In the case of DAO, specify the new DAO class in PersistenceContext.xml.
7. Change the new Struts configuration and Action classes that reference the customized Application Manager Session facade.
8. If necessary, change the Commerce Service Session facade to give control of the toDTO and fromDTO methods to the Entity bean and do not assemble or disassemble the DTO in this layer, as it does not give a good plug point for the Extensible DTOs.

Figure 6–15 describes the lifecycle of the data throughout the request.

Figure 6–15 Data Extension through Composition

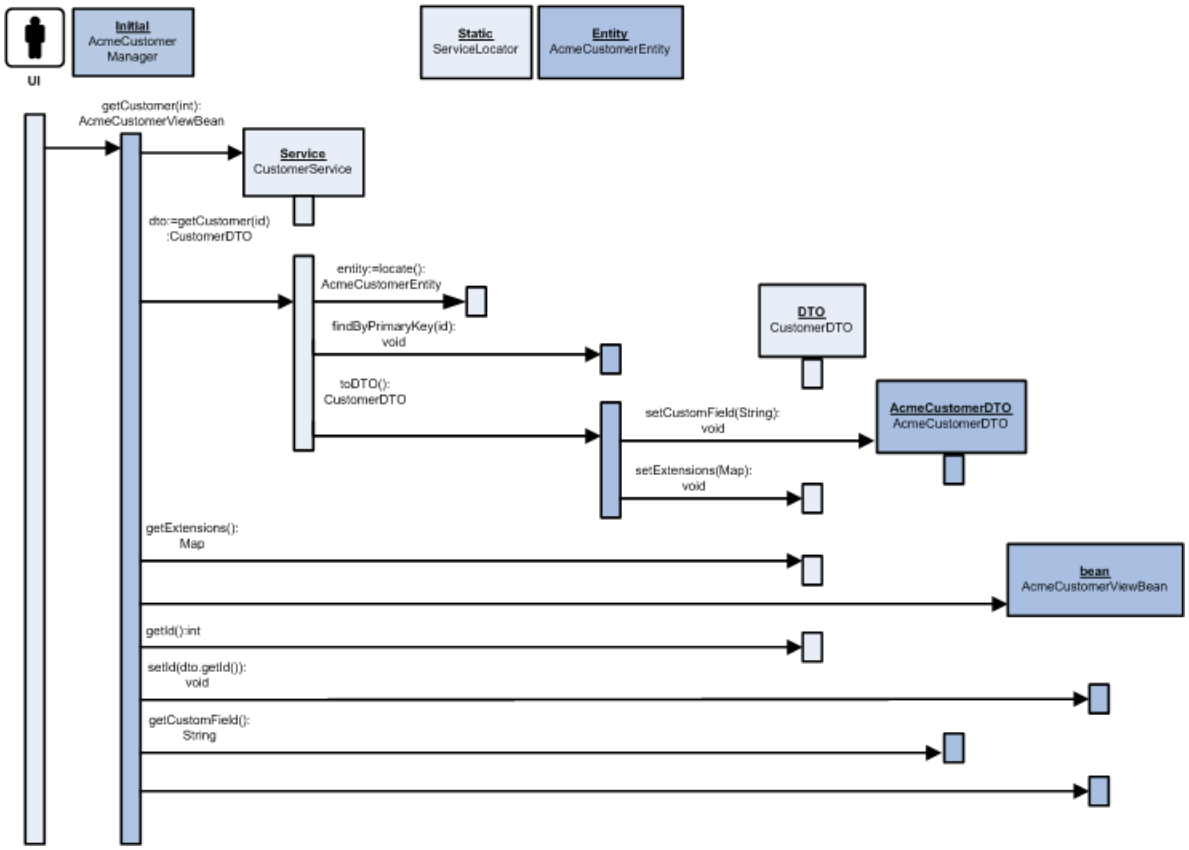


Figure 6–16 describes the various classes created.

Figure 6–16 Data Extension Through Composition: Class Diagram—Entity Bean

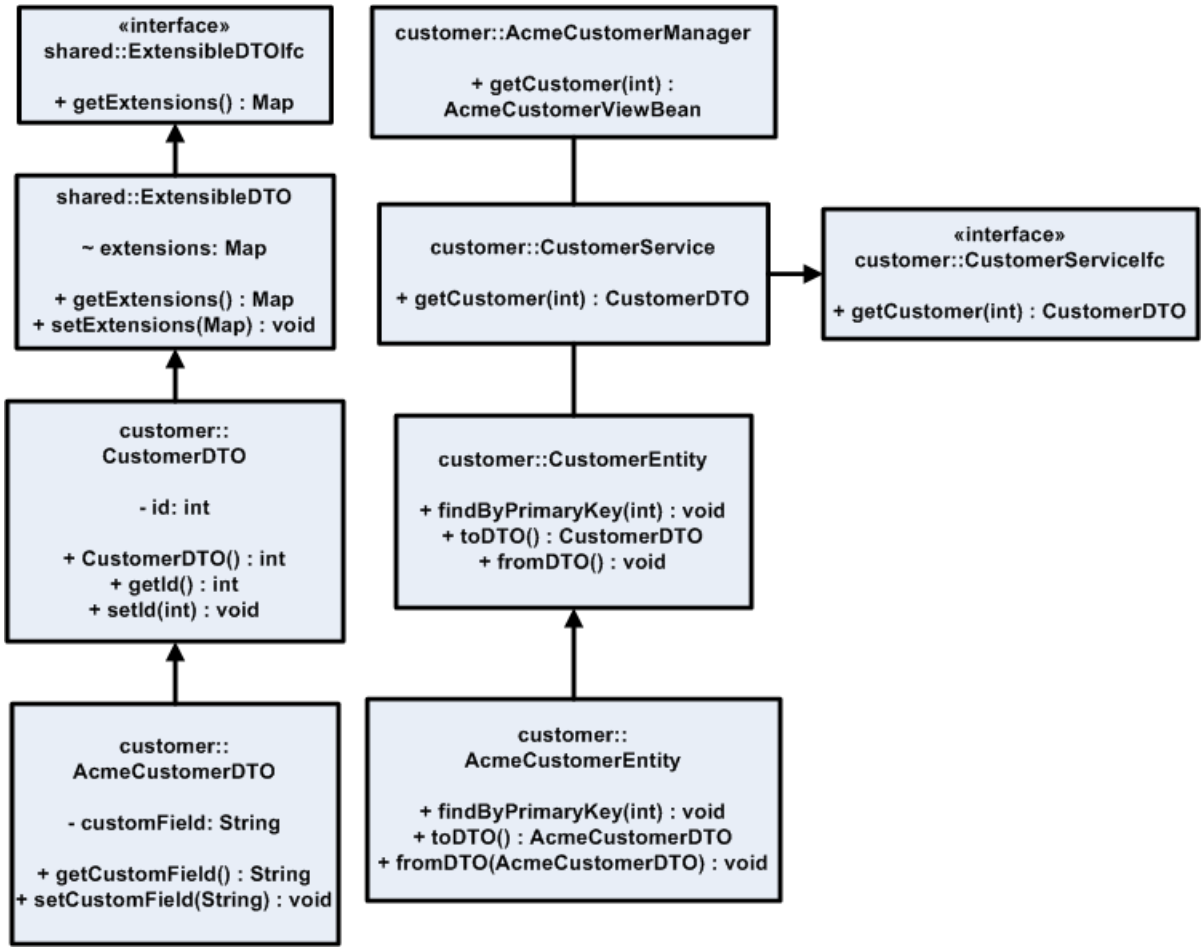
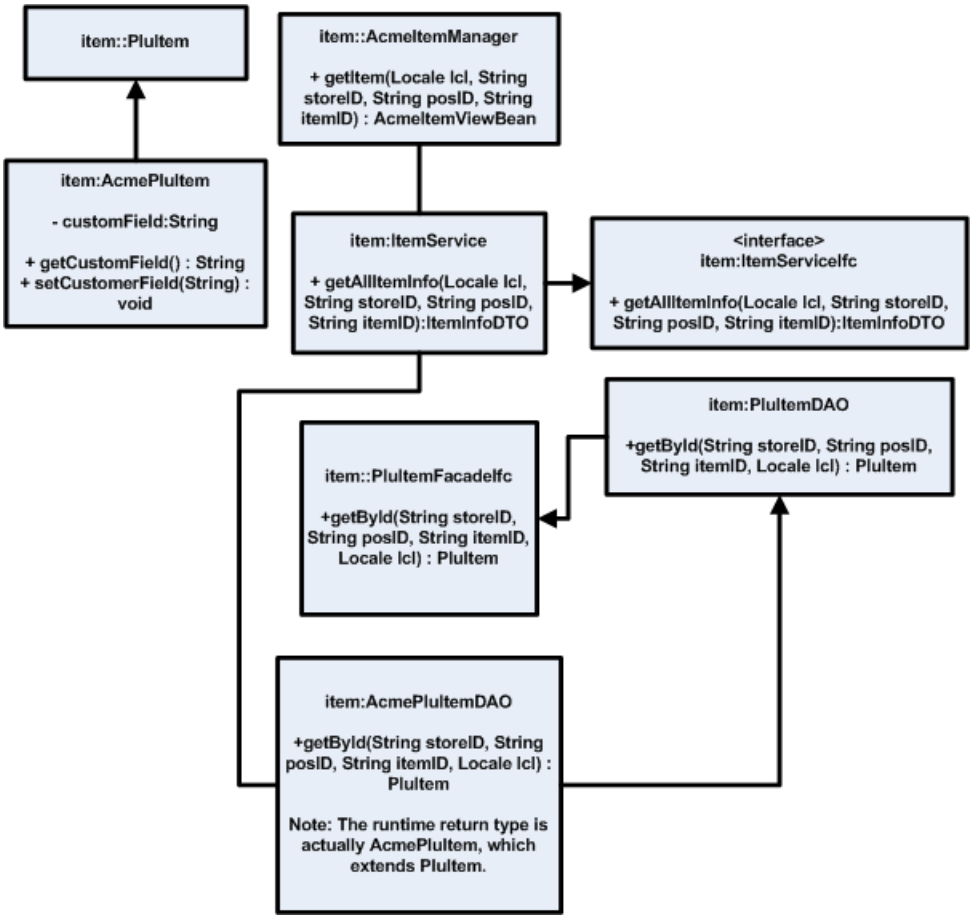


Figure 6–17 Data Extension Through Composition: Class Diagram—DAO



7

Returns Management Extension Guidelines

In order to pass arbitrary data, the schemas used for return request and return result contain an optional element called RetMsgExtDesc. This optional element contains zero or more sub-elements known as ExtensionElements that are simple name/value pairs of string data.

Element Location and Schema Definition

The RetMsgExtDesc element is attached to multiple parent elements in the two schemas. Table 7–1 summarizes the RetMsgExtDesc elements.

Table 7–1 RetMsgExtDesc Locations

Schema	Type
RetAuthDesc.xsd	ReturnRequest
	ItemReturnInfo
	ItemTransactionInfo
RetResultDesc.xsd	ReturnResult
	ItemReturnResult

The following example is a schema definition for the RetMsgExtDesc and ExtensionElement. For reference, also see the relevant section of the RetAuthDesc schema.

> **Note:** The names and values are strings.

Example 7–1 RetMsgExtDesc

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:retailDoc="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.oracle.com/retail/integration/base/bo/RetMsgExtDesc/v1"

targetNamespace="http://www.oracle.com/retail/integration/base/bo/RetMsgExtDesc/v1
"
    elementFormDefault="qualified" version="1.0">

    <xs:element name="RetMsgExtDesc">
        <retailDoc:annotation>
            <retailDoc:documentation>
                Contain information related to return Message Extensions.
            </retailDoc:documentation>
```

```
        </retailDoc:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ExtensionEntry" minOccurs="0"
maxOccurs="unbounded">
                    <retailDoc:annotation>
                        <retailDoc:documentation>
                            Return Message Extensions.
                        </retailDoc:documentation>
                    </retailDoc:annotation>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ExtensionEntry">
        <retailDoc:annotation>
            <retailDoc:documentation>
                Contains the Extension Entry information.
            </retailDoc:documentation>
        </retailDoc:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="varchar250" minOccurs="1"
maxOccurs="1">

                    <retailDoc:annotation>
                        <retailDoc:documentation>
                            Name of the extension.
                        </retailDoc:documentation>
                    </retailDoc:annotation>
                </xs:element>
                <xs:element name="value" type="varchar250" minOccurs="1"
maxOccurs="1">
                    <retailDoc:annotation>
                        <retailDoc:documentation>
                            Value of the Extension.
                        </retailDoc:documentation>
                    </retailDoc:annotation>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:simpleType name="varchar250">
        <retailDoc:annotation>
            <retailDoc:documentation>
                This type can hold a string of max length of 50 characters.
            </retailDoc:documentation>
        </retailDoc:annotation>
        <xs:restriction base="xs:string">
            <xs:maxLength value="50" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>

<xs:element ref="RetMsgExtDesc:RetMsgExtDesc" minOccurs="0" maxOccurs="1">
                    <retailDoc:annotation>
                        <retailDoc:documentation>
                            Return message extensions.
                        </retailDoc:documentation>
                    </retailDoc:annotation>
```

```
</xs:element>
```

Element Usage and Retrieval

To use this optional element, the message sent to Oracle Retail Returns Management needs to include a RetMsgExtDesc with as many ExtensionEntry elements as necessary. For example, an XML message using the extension to pass custom data might look similar to the following example:

Example 7–2 XML Message Using RetMsgExtDesc

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ReturnRequest>
… lines omitted …
    <transactionType>Return</transactionType>
    <RetMsgExtDesc>
    <ExtensionEntry>
          <name>LegacyID</name>
          <value>sun</value>
       </ExtensionEntry>
       <ExtensionEntry>
         <name>LegacyTransaction</name>
         <value>moon</value>
       </ExtensionEntry>
    </RetMsgExtDesc>
</ReturnRequest>
```

Here, the message contains two values for LegacyID and LegacyTransaction.

Once the values have been added to the message, the message is then sent to Oracle Retail Returns Management using the desired transport (such as through a web service or a direct API call). The XML is placed by JAXB into a list accessible through the appropriate element. In this case, the element is the ReturnRequest object. The user can then search through this list to find the values the user wants. The following example demonstrates searching the RetMsgExtDesc elements:

Example 7–3 Searching the RetMsgExtDesc Elements

```
        RetMsgExtDesc mesageExtDesc = request.getRetMsgExtDesc();
        if(mesageExtDesc != null)
        {
        List list = request.getRetMsgExtDesc().getExtensionEntry();
        for (Iterator iterator = list.iterator(); iterator.hasNext();) {
        ExtensionEntry entry = (ExtensionEntry) iterator.next();
        if(entry.getName().equals("LegacyID"))
        {
        //do something
        }

}
        }
```

Note: If the RetMsgExtDesc element is not included with the ReturnRequest or ReturnResult, the RetMsgExtDesc() method returns a null. If the element exists but does not have any child elements, then the list returned by getExtensionEntry() is zero length.

8

Coding Your First Back Office or Central Office Feature

This chapter describes how to add a feature to Back Office or Central Office using a specific example based on extending a search page within the application's web-based UI. The example is a simple extension of an existing search criteria page to allow it to search on additional criteria.

> **Note:** See the *Oracle Retail POS Suite Security Guide* for more information about specific security features and implementation guidelines for the POS Suite products.

Before You Begin

Before you attempt to develop new Back Office or Central Office code, set up your development environment as described in Chapter 3. Verify that you can successfully build and deploy an ear file.

Extending Transaction Search

This section explores the extension of transaction search features through the creation of a new criteria page. The changes required to implement this functionality interact with the user interface and the internals of the Central Office system. This example takes you through the process of implementing a new search criteria page, under the assumption that you have been asked to develop a page that enables a user to screen transactions according to new criteria.

> **Note:** Paths in this chapter are assumed to start from your local source code tree, checked out from the source code control system.

Item Quantity Example in Central Office

As an example of how to extend Central Office, this chapter refers to a new search criteria page called Item Quantity. This new page is an addition to the Transaction Tracker tab. The existing interface offers a side navigation bar with options to search by Item, Transaction, Sales Associate, Customer, and others. Item Quantity is a new option on this side navigation bar; it looks much like the Item page but enables the user to set a quantity value and search for transactions whose quantity of any item compares appropriately to a chosen quantity (for example, greater than, greater than or equal to, less than, and so on).

This example shows how:

- A new user interface can be created.
- Search criteria is collected from the end user.
- Data is handed off from one layer of the interface to another.
- SQL queries are handled and modified.

The following procedures offer general steps followed by specific examples.

Search by Login ID in Back Office

The existing employee search page allows the operator to search for employees by their employee ID, or by first and last name, or by role. This example will add the ability to search for employees by their login ID.

This example shows how:

- A user interface can be modified
- Search criteria is collected from the end user
- Data is handed off from one layer of the interface to another
- SQL queries are handled and modified

The following procedures offer general steps followed by specific examples.

Web UI Framework in Central Office

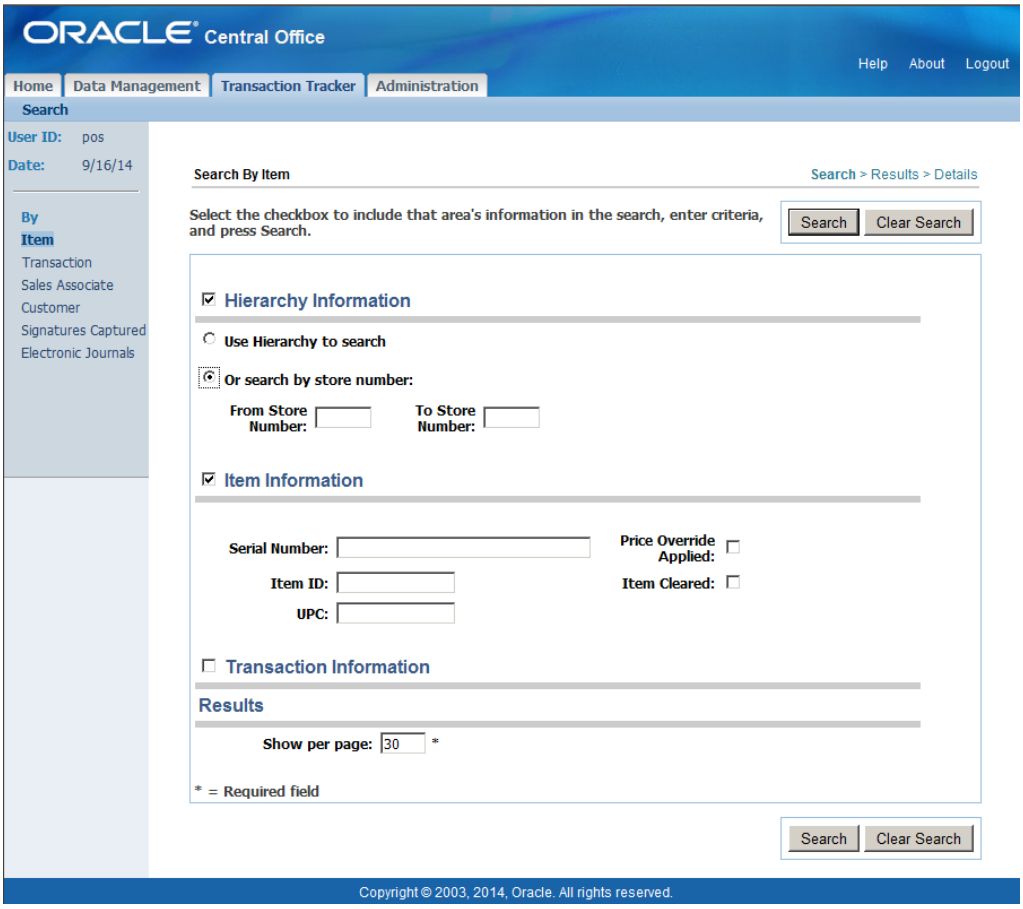
To add a new search criteria page, you must create a new JSP file for the page, edit workflow and Struts/Tiles configuration files to register the page, and add appropriate classes to handle the page.

Create a New JSP file

Create a new JSP file and edit the file content. You can start with a copy of an existing criteria page and add input fields for the new data you intend to factor into your search. Plan your string usage to reference property files for internationalization purposes.

To create ItemQuantityCriteria.jsp, make a copy of `<source_directory>\webapp\transaction-webapp\web\centralizedElectronicJournal\ItemCriteria.jsp`. Establish input fields to collect store numbers, item numbers, item quantity, and the item quantity limit operator (the operator that determines how to compare a transaction's item quantities with the item quantity criteria).

Figure 8–1 Item Quantity Criteria JSP Page Mock-Up



Add Strings to Properties Files

Add references to any new strings to appropriate properties files.

For example, to add Item Quantity Information and Item Quantity column labels, edit the `<source_directory>\webapp\i18n-webapp\src\ui\oracle\retail\stores\webmodules\i18n\en\transaction.properties` file to add the following entries:

- transaction.centej.itemquantitycrit.header=Item Quantity Information
- transaction.centej.itemquantitycrit.label.itemquantity=Item Quantity

Configure the sideNav Tile

To add the new JSP page to the side navigation bar in the Transaction Tracker tab, you configure the sideNav tile. Using Struts/Tiles conventions, edit the `<source_directory>\webapp\transaction-webapp\WEB-INF\360\tiles-transaction_tracker.xml` file, making the following edits:

- Add an entry to the <putList name="sideNav"> tag to add your new page name to the list of options on the side navigation bar.
- Set the security role for this new option by adding an element tag in the appropriate location in the <putList name="sideNavRoles"> tag. You can use the element <add value="BLANK"/> if no role has yet been defined.
- Add a destination URL to be activated when your new page name is clicked.

The following code sample shows where to add tags:

Example 8–1 transaction_tracker.xml: SideNav Option List and Roles

```
<putList name="sideNav">
    <add value="By"/>
    <add value="Item"/>
    ...add your new tag here...
    <add value="Transaction"/>
    <add value="Sales Associate"/>
    <add value="Customer"/>
</putList>
<putList name="sideNavRoles">
    <add value="BLANK"/>
    <add value="search_by_item"/>
    ...add your new tag here...
    <add value="search_by_trans"/>
    <add value="search_by_assoc"/>
    <add value="search_by_cust"/>
</putList>
<putList name="sideNavURLs">
    <add value="BLANK"/>
    <add value="centralizedElectronicJournal/ejItemSearch.do"/>
    ...add your new tag here...
    <add value="centralizedElectronicJournal/ejTransactionSearch.do"/>
    <add value="centralizedElectronicJournal/ejSalesAssociateSearch.do"/>
    <add value="centralizedElectronicJournal/ejCustomerSearch.do"/>
</putList>
```

Finally, add a set of definition tags to define your JSP page's title, help URL, and body layout. The following code sample offers an example:

Example 8–2 Example Definition Tags for tiles-transaction_tracker.xml

```
<definition name="centralizedElectronicJournal.ejItemQuantitySearch"
extends="ejournal">
        <put name="sideNavIndex" value="Item Quantity"/>
        <put name="title" value="Search By Item Quantity"/>
        <put name="helpURL"
value="centralizedElectronicJournal/help.do#searchbyitem"/>
        <put name="body"
value="centralizedElectronicJournal.ejItemQuantitySearch.layout"/>
    </definition>

<!-- the following definition defines the layout for the JSP's body, as called out
above --!>

<definition name="centralizedElectronicJournal.ejItemQuantitySearch.layout"
extends="ejournal.search.layout">
        <put name="resetSearchURL"
value="/centralizedElectronicJournal/ejItemQuantitySearch.do"/>
        <put name="searchTitle" value="Search By Item Quantity"/>
        <put name="searchAction"
value="/centralizedElectronicJournal/searchTransactionByItemQuantity.do"/>
        <put name="expandSections" value="itemQuantityCriteria"/>
        <put name="searchCriteria1"
value="/centralizedElectronicJournal/ItemQuantityCriteria.jsp"/>
        <put name="searchCriteria2"
value="/centralizedElectronicJournal/transactionCriteria.jsp"/>
        <put name="searchCriteria3"
```

```
value="/centralizedElectronicJournal/resultsCriteria.jsp"/>
    </definition>
```

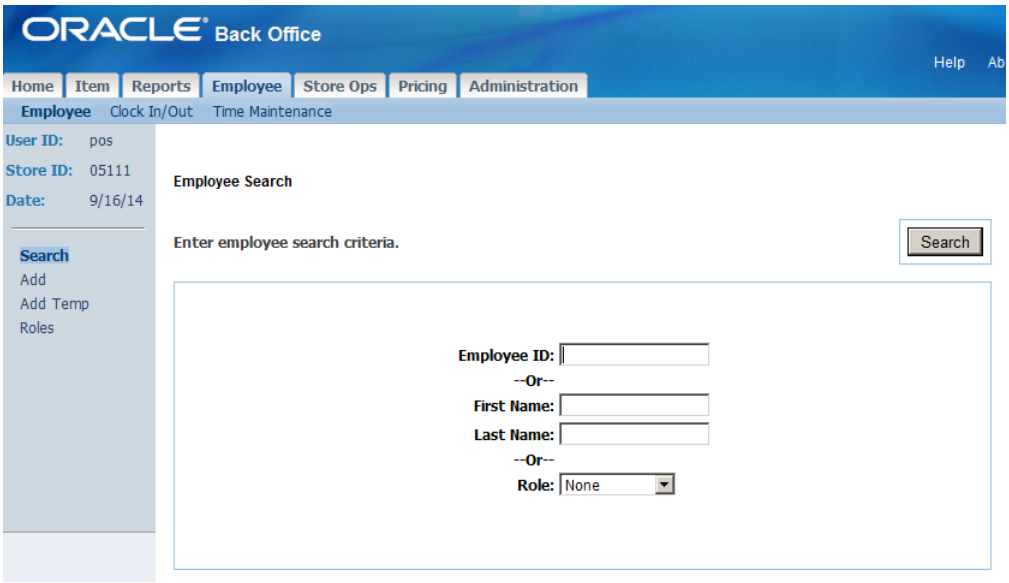
Web UI Framework in Back Office

The user interface changes require that you update the JSP page to add the additional search criteria. You also need to ensure any strings you use are properly externalized for future localization. Depending on what kind of form is used you may have additional work to perform. A review of the Struts action mapping used for employee searches will reveal what changes are required to any action form in use. Finally, the action used to search will also need to be modified.

Modify the JSP File

Modification of the JSP file is fairly straightforward. Taking the existing JSP structure into consideration, insert the new search criteria between the Employee ID field and the First and Last Name fields.

Figure 8–2 Employee Search Screen



Adding the search criteria requires that you insert the appropriate HTML and JSP tags to the `<source_directory>\webapp\employee-webapp\web\employee\employeeSearch.jsp` file. You can identify where the changes need to be placed by looking for the message tag that displays the `employee.employeeForm.employeeFirstName.label` message. The code in the following example presents the required changes to the employee search page.

Example 8–3 EmployeeSearch.jsp Modifications

```
<tr>
    <td align="right" class="fieldname">--<bean:message
        key="prompt.or"/>--</td>
    <td align="right"> </td>
</tr>
<tr>
    <td align="right" class="fieldname"><bean:message
        key="employee.employeeForm.employeeLoginId.label"/>: </td>
    <td align="left">
```

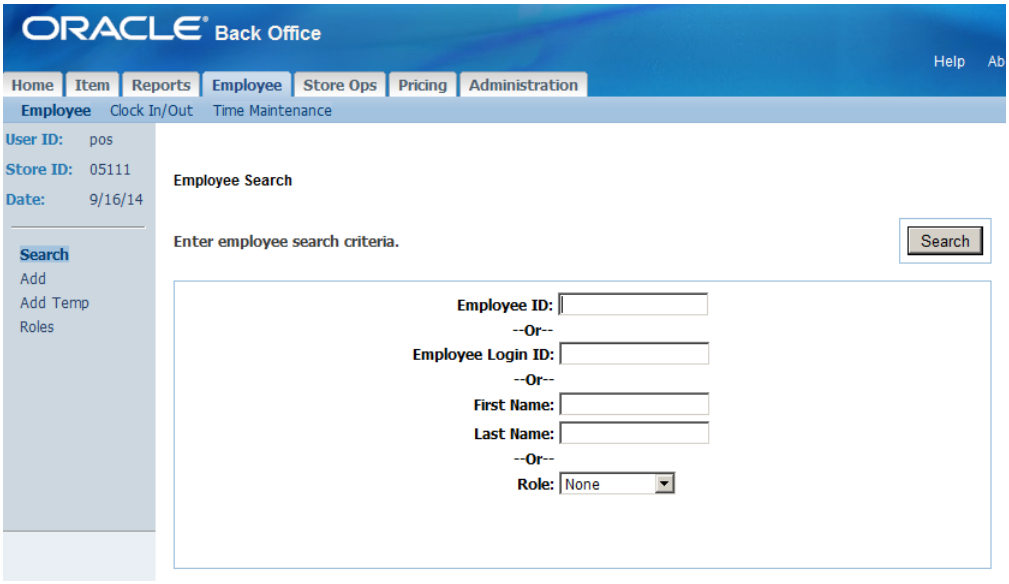
```
            <html:text styleClass="data"
                property="searchEmployeeLoginId"
                size="20" maxlength="10" tabindex="2"/>
        </td>
    </tr>
    <tr>
        <td align="right" class="fieldname">--<bean:message
            key="prompt.or"/>--</td>
        <td align="right"> </td>
    </tr>
    <tr>
        <td align="right" class="fieldname"><bean:message
            key="employee.employeeForm.employeeFirstName.label"/>: </td>
        <td align="left">
            <html:text styleClass="data"
                property="searchEmployeeFirstName" size="20"
                maxlength="16" tabindex="3"/></td>
    </tr>
```

The text in bold in this example is the new text that was added to the employeeSearch.jsp page. It introduces a new label and new text box to collect the new search criteria.

> **Note:** The tab index values were incremented for all of the remaining input fields.

The modified screen is presented in Figure 8–3.

Figure 8–3 Modified Employee Search Screen



Externalize Strings

It is very important that all user-visible strings be externalized for localization. Check to see if the label text is already defined or if you need to create it. If it exists, it is likely already in the `<source_directory>\webapp\i18n-webapp\src\ui\oracle\retail\stores\webmodules\i18n\`

en\employee.properties file. A quick examination of this file reveals that the label text already exists and there is nothing to add.

Action Mapping

The Struts action mapping defines the important details you need to know about the code that executes the employee search when the search screen submits its data. The following example contains the XML fragment that defines the action. It comes from the struts-employee_actions.xml file that is included in the application's struts-config.xml file.

Example 8–4 Action Definition from struts-employee_actions.xml

```
<action path="/employee/searchEmployee"
    type="oracle.retail.stores.webmodules.employee.ui.SearchEmployeeAction"
    name="employeeForm"
    scope="request"
    input="/employee/searchEmployeeView.do"
    validate="true">
    <forward name="success" path="employee.searchEmployeeViewDetails"/>
    <forward name="showEmployeeList" path="employee.searchEmployeeView"/>
    <forward name="failure" path="employee.searchEmployeeView"/>
</action>
```

You may need to modify the action form this action uses to transfer the data from the modified JSP to the action class. The XML fragment indicates that the form is valid for a single request and is named **employeeForm**.

Action Form

The definition of the action form is contained in another XML file, struts-employee_forms.xml. It is presented in the following example.

Example 8–5 Action Form Definition from struts-employee_forms.xml

```
<form-bean name="employeeForm"
type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="searchEmployeeId" type="java.lang.String"/>
    <form-property name="searchEmployeeLoginId" type="java.lang.String"/>
    <form-property name="searchEmployeeFirstName" type="java.lang.String"/>
    <form-property name="searchEmployeeLastName" type="java.lang.String"/>
    <form-property name="employeeName" type="java.lang.String"/>
    <form-property name="employeeFormattedName" type="java.lang.String"/>
    <form-property name="employeeFirstName" type="java.lang.String"/>
    <form-property name="employeeMiddleName" type="java.lang.String"/>
    <form-property name="employeeLastName" type="java.lang.String"/>
    <form-property name="employeeId" type="java.lang.String"/>
    <form-property name="employeeAlternateId" type="java.lang.String"/>
    <form-property name="employeeLoginId" type="java.lang.String"/>
    <form-property name="employeeRole" type="java.lang.String"/>
    <form-property name="employeeStatus" type="java.lang.String"/>
    <form-property name="socialSecurityNumber" type="java.lang.String"/>
    <form-property name="employeeStatusCode" type="java.lang.String"/>
    <form-property name="workGroupId" type="java.lang.String"/>
    <form-property name="employeeLocale" type="java.lang.String"/>
    <form-property name="groupID" type="java.lang.String"/>
    <form-property name="employeeValidity" type="java.lang.String"/>
    <form-property name="employeeType" type="java.lang.String"/>
    <form-property name="employeeActualStatusCode" type="java.lang.String"/>
    <form-property name="errorMessage" type="java.lang.String"/>
```

```
    <form-property name="employeeStoreId" type="java.lang.String"/>
    <form-property name="employeeRoleName" type="java.lang.String"/>
</form-bean>
```

The definition reveals that the employee search screen is using a DynaValidatorForm provided by Struts. All of the form properties are defined here in the XML. The line in bold was added to introduce our new search criteria. Validation of the entered data should be addressed by proper configuration of the Struts validator. Examples of how to validate form data using the Struts DynaValidatorForm can be found in the `struts-employee_validator.xml` file provided with the source code and the Struts documentation.

Action Modification

With the new search criteria added to the action form, we can now turn our attention to modifying the action class that actually performs the search from the user interface. The action class used is located at `<source_directory>\webapp\employee-webapp\src\ui\oracle\retail\stores\webmodules\employee\ui\SearchEmployeeAction.java`. The action class `execute()` method is broken into three sections using an **if-then-else** statement. The following example provides a portion of the updates required to enable the action to search for employees based on their login ID.

Example 8–6 Modifications to SearchEmployeeAction.java

```
public class SearchEmployeeAction extends Action
{
        public ActionForward execute(...) throws Exception
    {
...
        String employeeLoginId = "";
...
        try
        {
...
            employeeLoginId = dynaActionForm
                .get("searchEmployeeLoginId").toString();
...
            if (!employeeId.equals(""))...
            else if (!employeeLoginId.equals(""))
            {
                EmployeeDTO employeeDTO = employeeManager
                    .getEmployeeByLoginID(employeeLoginId);
                -- similar code to the first if condition to prepare
                -- the necessary data for display to the user since
                -- this type of search will match only a one employee
            }
            else if (employeeId.equals("")
                && !(employeeFirstName.equals("")
                && employeeLastName.equals("")))...
            else...
        }
...
    }
}
```

As the only differences between the search by employee ID and search by employee login ID are how the employee is found, the code in the first two **if** blocks are almost identical and could be refactored to share implementations. For the purposes of this

example that repeated code and possible refactoring has been omitted. The important difference is the call to the `employeeManager.getEmployeeByLoginID()` method.

Configure Action Mapping in Central Office

Configure action mapping in one of the struts configuration files so that Struts knows how to handle your new JSP page.

The following example shows how the Item Quantity page could be configured. The file is `<source_directory>\webapp\transaction-webapp\WEB-INF\360\struts-transaction_tracker_actions.xml`. The code sets up the system to request an item quantity search and forwards results to standard result routines, automatically displaying the transaction details (through showDetails.do) if only one result is returned, and otherwise displaying a standard transaction list.

Example 8–7 Struts Action Configuration for Item Quantity

```
<action path="/centralizedElectronicJournal/ejItemQuantitySearch"
type="oracle.retail.stores.webmodules.transaction.ui.StartSearchAction">
<forward name="success" path="centralizedElectronicJournal.ejItemQuantitySearch"/>
</action>

<action path="/centralizedElectronicJournal/searchTransactionByItemQuantity"
type="oracle.retail.stores.webmodules.transaction.ui.SearchTransactionByItemQuanti
tyAction"
name="searchTransactionForm"
scope="request"
input="/centralizedElectronicJournal/ejItemQuantitySearch.do">
    <forward name="oneResult"
path="/centralizedElectronicJournal/showDetails.do"/>
    <forward name="multipleResults"
path="centralizedElectronicJournal.ejTransactionSearchResults"/>
</action>
```

Add Code to Handle New Fields to Search Transaction Form

Since you have added new search fields for the Item Quantity and Item Quantity Operator, you must add code for handling these fields and their validation to the `<source_directory>\webapp\transaction-webapp\src\ui\oracle\retail\stores\webmodules\transaction\ui\SearchTransactionForm.java` file.

1. Add the instance fields for any fields you have added to the criteria page, and use the same names as the input field names you defined in your JSP page, so that the fields can be automatically populated via retrospection. Note an additional static constant for the search based on line item quantity.

Example 8–8 New Instance Fields

```
    private String itemQuantityLimitOperator;
    private int itemQuantityLimit;

    public static final String ITEM_QUANTITY_LIMIT_OPERATOR =
      "itemQuantityLimitOperator";

    public static final String ITEM_QUANTITY_LIMIT =
      "itemQuantityLimit";
    public static final String SEARCH_BY_ITEM_QUANTITY_CRITERIA =
      "searchByItemQuantityCriteria";
```

```
private Boolean searchByItemQuantityCriteria;
```

2. Define corresponding getter and setter methods for the instance fields.

Example 8–9 Getter and Setter Methods for New Instance Fields

```
public Boolean getSearchByItemQuantityCriteria()
{
    return searchByItemQuantityCriteria;
}

public void setSearchByItemQuantityCriteria(Boolean
                                  searchByItemQuantityCriteria)
{
this.searchByItemQuantityCriteria =
                            searchByItemQuantityCriteria;
}

public String getItemQuantityLimitOperator()
{
    return itemQuantityLimitOperator;
}

public void setItemQuantityLimitOperator(String
                                itemQuantityLimitOperator)
{
  this.itemQuantityLimitOperator = itemQuantityLimitOperator;
}

public int getItemQuantityLimit()
{
    return itemQuantityLimit;
}

public void setItemQuantityLimit(int itemQuantityLimit)
{
  this.itemQuantityLimit = itemQuantityLimit;
}
```

3. Add the validation for the item quantity limit value to check that the input was a valid number and was greater than zero. To do this add the following code in the validate method and then provide the method implementation. The method implementation uses an error message key to look up the actual error message description.

Example 8–10 Code to Add to Validate Method

```
    if (getSearchByItemQuantityCriteria().booleanValue())
    {
        validateSearchByItemQuantityCriteria(errors);
    }
```

Example 8–11 New Validation Method

```
private void validateSearchByItemQuantityCriteria(ActionErrors
                                                  errors)
{
if (getItemQuantityLimit() <= 0)
    {
          errors.add("searchItemQuantityLimit",
```

```
                new ActionError("error.ejournal.search.itemquantity.
                            itemquantitylimitvalue"));
        }
    }
```

4. Store any error messages for validation in the `<source_directory>\webapp\i18n-webapp\src\ui\oracle\retail\stores\webmodules\i18n\transaction.properties` file.

   In the item quantity example, you might store an error message description as follows:

   ```
   error.ejournal.search.itemquantity.itemquantitylimitvalue=Item quantity limit
   value must be a valid number and greater than zero.
   ```

Create a Struts Action Class

Create a Struts action class to act as a controller for the JSP you created.

For Item Quantity, create an action class using the filename SearchTransactionByItemQuantityAction.java, in the directory `<source_directory>\webapp\transaction-webapp\src\ui\oracle\retail\stores\webmodules\transaction\ui\`.

You can start by copying and modifying SearchTransactionByItemAction.java.

Add Method to Base Class

Add code to the base search class, SearchTransactionAction.java, to establish a get method for the new criteria:

1. Add a line to call a new method.

Example 8–12 Call a New Method to Get Item Quantity Criteria

```
searchCriteria = new SearchCriteria(getTransactionCriteria(searchTransactionForm,
request.getParameterValues("transactionType")),
getTenderCriteria(searchTransactionForm),
getSalesAssociateCriteria(searchTransactionForm),
getLineItemCriteria(searchTransactionForm),getLineItemQuantityCriteria(searchTrans
actionForm),getCustomerCriteria(searchTransactionForm),
getSignatureCaptureCriteria(searchTransactionForm));
```

2. Add the method implementation.

Example 8–13 getLineItemQuantityCriteria Method Implementation

```
    /**
     *    Returns a LineItemQuantityCriteria object based on values
     *    from a SearchTransactionForm.
     *
     */
    protected LineItemQuantityCriteria
    getLineItemQuantityCriteria(SearchTransactionForm form)
    {
        if ( form.getSearchByItemQuantityCriteria().booleanValue() )
        {
            criteria = new LineItemQuantityCriteria();

            if ( StringUtils.isNotEmpty(form.getItemNumber()) )
            {
                criteria.setItemNumber(form.getItemNumber());
```

```
            }


            if ( StringUtils.isNotEmpty(form.getItemQuantityLimitOperator()) )
            {

criteria.setItemQuantityLimitOperator(form.getItemQuantityLimitOperator());
            }

            if (form.getItemQuantityLimit() > 0)
            {
                criteria.setItemQuantityLimit(form.getItemQuantityLimit());
            }
        }

        return criteria;
    }
```

Application Services in Back Office

The Employee Manager session bean already contained the required method to search for employees by their login ID. The business interface for the employee manager is located in the file <source_directory>\webapp\employee-webapp\src\app\oracle\retail\stores\webmodules\employee\app\EmployeeManagerIfc.java. It contains the declarations of the methods available to the user interface. The method to find an employee by their login ID is presented in Example 8–14.

Example 8–14 EmployeeManagerIfc.java

```
/**
 * Finds the employee with the specified Login ID.
 * @param loginID the ID of the employee to find.
 * @return A DTO containing the employee data.
 * @throws EmployeeNotFoundException if the employee cannot be
 * found
 */
EmployeeDTO getEmployeeByLoginID(String loginID)
    throws EmployeeNotFoundException, RemoteException;
```

The bean implementation is located in <source_directory>\webapp\employee-webapp\src\app\oracle\retail\stores\webmodules\employee\app\ejb\EmployeeManagerBean.java As a façade it simply delegates the call to the Employee Service bean in the Commerce Services layer.

Verify Application Manager Implementation in Central Office

Verify that the application manager appropriately calls for information from Commerce Services. In the Item Quantity search criteria example, the <source_directory>\webapp\transaction-webapp\src\app\oracle\retail\stores\webmodules\transaction\app\ejb\EJournalManagerBean.java class is used. This class already contains the necessary method implementation for a getTransactions() method.

Commerce Services in Back Office

The Employee Service bean in the Commerce Services layer is another façade that prevents direct access to the entity beans. The implementation obtains an instance of

the EmployeeLocalHome and invokes the findByLoginID() method. Examination of the Employee entity bean will reveal that it is directly querying the database to find the user with the corresponding login ID.

Add Business Logic to Commerce Service in Central Office

Use the following to add business logic to commerce services.

Create a Class to Create the Criteria Object

You must create a new class in the Commerce Services layer to handle the creation of the new ItemQuantityCriteria object type, adding instance fields for the fields you added. The class should provide the following:

- Variables for required criteria fields
- Boolean flags to indicate (to the data layer) whether a given attribute should be included in a query
- Getter and setter methods for the new fields
- Use() and reset() methods

Add New Criteria to the Service

The new criteria you have added must be included in the class that processes search criteria. For transactions, this class is `<source_directory>\commerceservices\transaction\src\oracle\retail\stores\commerceservices\transaction\SearchCriteria.java`.

To make LineItemQuantityCriteria work, add it to the variable declarations and the constructors and add new getter and setter methods, as shown in the highlighted portions of Example 8–15:

Example 8–15 SearchCriteria.java

```
public class SearchCriteria implements Serializable
{
    private TransactionCriteria transactionCriteria;
    private TenderCriteria tenderCriteria;
    private SalesAssociateCriteria salesAssociateCriteria;
    private LineItemCriteria lineItemCriteria;
    private LineItemQuantityCriteria lineItemQuantityCriteria;    private
CustomerCriteria customerCriteria;
    private SignatureCaptureCriteria signatureCaptureCriteria;

    public SearchCriteria()
    {
        this(null, null, null, null, null,null);
    }

    public SearchCriteria(TransactionCriteria transactionCriteria,
                          TenderCriteria tenderCriteria,
                          SalesAssociateCriteria
                          salesAssociateCriteria,
                          LineItemCriteria lineItemCriteria,
                          LineItemQuantityCriteria lineItemQuantityCriteria,
                          CustomerCriteria customerCriteria)

    {
        this(transactionCriteria,
```

```
            tenderCriteria,
            salesAssociateCriteria,
            lineItemCriteria,
            lineItemQuantityCriteria,
            customerCriteria,
            null);
    }


public SearchCriteria(TransactionCriteria transactionCriteria,
                      TenderCriteria tenderCriteria,
                      SalesAssociateCriteria
                      salesAssociateCriteria,
                      LineItemCriteria lineItemCriteria,
                      LineItemQuantityCriteria
                      lineItemQuantityCriteria,
                      CustomerCriteria customerCriteria,
                      SignatureCaptureCriteria
                      signatureCaptureCriteria)

    {
        setTransactionCriteria(transactionCriteria);
        setTenderCriteria(tenderCriteria);
        setSalesAssociateCriteria(salesAssociateCriteria);
        setLineItemCriteria(lineItemCriteria);
      setLineItemQuantityCriteria(lineItemQuantityCriteria);
        setCustomerCriteria(customerCriteria);
        setSignatureCaptureCriteria(signatureCaptureCriteria);
    }

...

    public LineItemQuantityCriteria getLineItemQuantityCriteria()
    {
        return lineItemQuantityCriteria;
    }

    public void
             setLineItemQuantityCriteria(LineItemQuantityCriteria
                                         lineItemQuantityCriteria)
    {
        this.lineItemQuantityCriteria = lineItemQuantityCriteria;
    }
```

Handle SQL Code Changes in the Service Bean

The service bean creates the SQL code that pulls data from the database. Add code to the appropriate ServiceBean.java file to append new criteria to the From clause and the Where clause.

To make the Line Item Quantity Criteria work, edit the `<source_directory>\commerceservices\transaction\src\com\oracle\retail\stores\commerceservices\transaction\ejb\TransactionServiceBean.java` file as follows:

1. Add a method call to append to the From clause.

```
query.append(addToFromClause(searchCriteria.getLineItemQuantityCriteria()));
```

2. Add the method implementation for the addToFromClause() method.

Example 8–16 addToFromClause() Method

```
    /** LineItemQuantityCriteria Criteria
    *
    */
   private String addToFromClause(LineItemQuantityCriteria
                                  criteria)
   {
      StringBuffer buffer = new StringBuffer();
       if (criteria != null && criteria.use())
       {
           buffer.append(" JOIN TR_LTM_RTL_TRN ON TR_LTM_RTL_TRN.ID_STR_RT = TR_
TRN.ID_STR_RT AND TR_LTM_RTL_TRN.ID_WS = TR_TRN.ID_WS AND TR_LTM_RTL_TRN.DC_DY_BSN
= TR_TRN.DC_DY_BSN AND TR_LTM_RTL_TRN.AI_TRN = TR_TRN.AI_TRN ");
           buffer.append(" JOIN TR_LTM_SLS_RTN ON TR_TRN.ID_STR_RT = TR_LTM_SLS_
RTN.ID_STR_RT AND TR_TRN.ID_WS = TR_LTM_SLS_RTN.ID_WS AND TR_TRN.DC_DY_BSN = TR_
LTM_SLS_RTN.DC_DY_BSN AND TR_TRN.AI_TRN = TR_LTM_SLS_RTN.AI_TRN ");
           buffer.append(" JOIN AS_ITM ON TR_LTM_SLS_RTN.ID_ITM  = AS_ITM.ID_ITM
");
           buffer.append(" JOIN AS_ITM_STK ON AS_ITM.ID_ITM = AS_ITM_STK.ID_ITM
");
           buffer.append(" JOIN ID_IDN_PS ON AS_ITM.ID_ITM = ID_IDN_PS.ID_ITM  ");
       }
       return buffer.toString();
   }
```

3. Add a method call to append to the Where clause.

```
    query.append(addToWhereClause(searchCriteria.getLineItemQuantityCriteria()));
```

4. Add the method implementation for the addToWhereClause() method.

Example 8–17 addToWhereClause() Method

```
  addToWhereClause(searchCriteria.getLineItemQuantityCriteria())
  as below.

    /**
    *
    */

   private String addToWhereClause(LineItemQuantityCriteria
                                   criteria)
   {

       StringBuffer query = new StringBuffer("");
       if (criteria != null && criteria.use())
       {
           if ((criteria.getItemNumber() != null &&
criteria.getItemNumber().length() > 0))
           {
              query.append(" AND TR_LTM_SLS_RTN.ID_ITM_
POS="+criteria.getItemNumber());
           }

          if (criteria.isSearchByItemQuantity())
           {
               query.append(" AND TR_LTM_SLS_RTN.QU_ITM_LM_RTN_
SLS"+criteria.getItemQuantityLimitOperator()+"?");
           }
       }
```

```
        return query.toString();
    }
```

5. Add a call to a method to bind the variables in the SQL query.

```
n = setBindVariables(ps, n, searchCriteria.getLineItemQuantityCriteria());
```

6. Add the method implementation for the setBindVariables() method.

Example 8–18 setBindVariables() method

```
setBindVariables(ps, n,
      searchCriteria.getLineItemQuantityCriteria()) as below.


    /**
    *
    */
   private int setBindVariables(PreparedStatement statement,
                                int index,
                                LineItemQuantityCriteria criteria)
                               throws SQLException
   {
       if (criteria != null && criteria.use())
       {
           if (criteria.isSearchByItemQuantity())
           {
               if (getLogger().isDebugEnabled())
            bindVariables.add(criteria.getItemQuantityLimit()+"");
               statement.setInt(index++,
                       criteria.getItemQuantityLimit());
           }
       }
       return index;
   }
```

9

Frameworks

The Oracle Retail architecture uses a combination of technologies that make it flexible and extensible, and allow it to communicate with other hardware and software systems. The frameworks that drive the application are implemented by the Java programming language, distributed objects, and XML scripting. The User Interface, Business Object, Manager/Technician, Data Persistence, and Navigation frameworks interact to provide a powerful, flexible application framework.

Frameworks

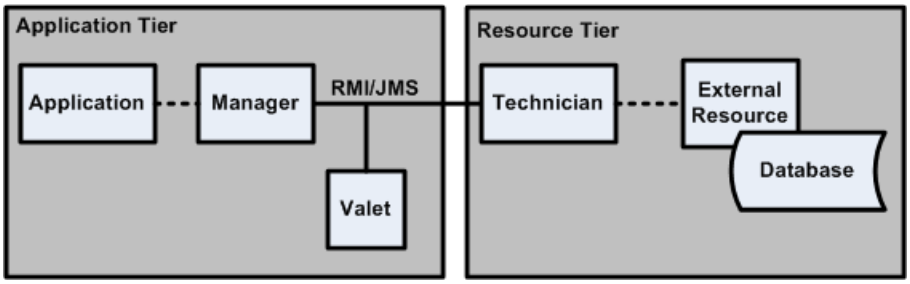
This section describes each framework.

Manager/Technician

The Manager/Technician framework is the component of Oracle Retail Platform that implements the distribution of data across a network. A Manager provides an API for the application and communicates with its Technician, which implements the interface to the external resource. The Manager is always on the same tier, or machine, as the application, while the Technician is usually on the same tier as the external resource.

Figure 9–1 shows an example of the Manager/Technician framework distributed on two different tiers.

Figure 9–1 Manager/Technician Framework



User Interface

The UI framework includes all the classes and interfaces in Oracle Retail Platform to support the rapid development of UI screens. In the application code, the developer creates a model that is handled by the UI Manager in the application code. The UI Manager communicates with the UI Technician, which accesses the UI Subsystem.

Figure 9–2 illustrates components of the UI framework.

Figure 9–2 UI Framework

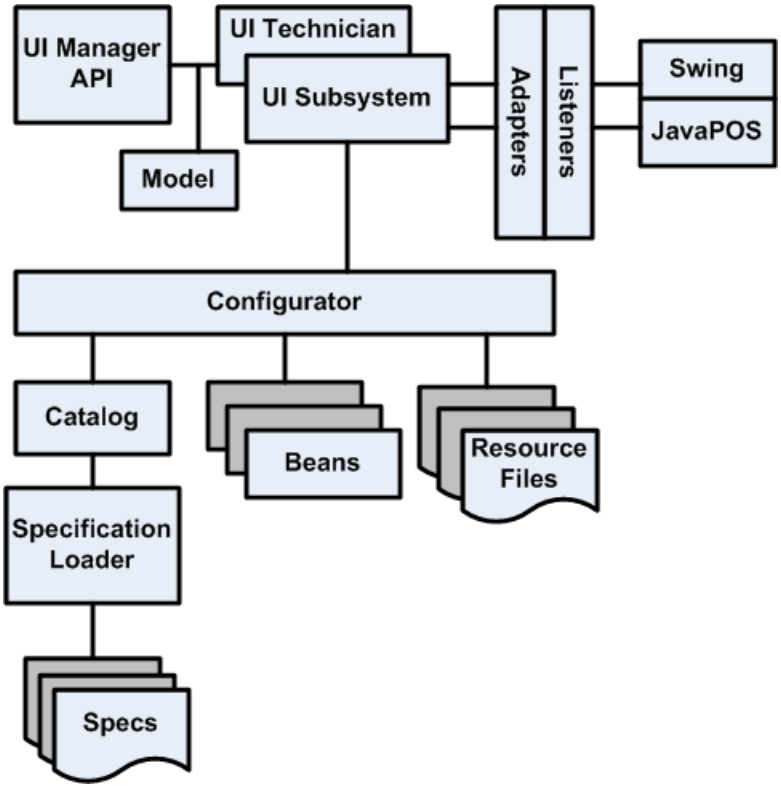


Table 9–1 describes the components of the UI framework.

Table 9–1 UI Framework Components

Component	Description
Resource Files	Resource files are text bundles that provide the labels for a screen. They are implemented as properties files. Text bundles are used for localizing the application.
Bean	Beans are reusable Java program building blocks that can be combined with other components to form an application. They typically provide the screen components and data for the workpanel area of the screen.
Specs	Specifications define the components of a screen. Display specifications define the width, height, and title of a window. Template specifications divide displays into areas. Bean specifications define classes and configurators and additional screen elements for a component. Default screen specifications map beans to the commonly used areas and define listeners to the beans. Overlay screen specifications define additional mappings of beans and listeners to default screens.
Specification Loader	Loaders find external specifications and interpret them. The loader instantiates screen specifications such as overlays, templates, and displays, and places the objects into a spec catalog.
Catalog	A Catalog provides the bean specifications by name. The UI Technician requests the catalog from the loader to simplify configurations.
Configurator	The UI framework interfaces with beans through bean configurator classes, which control interactions with beans. A configurator is instantiated for each bean specification. They apply properties from the specifications to the bean, configure a bean when initialized, reset the text on a bean when the locale changes, set the bean component data from a model, update a model from the bean component data, and set the filename of the resource bundle.

Table 9–1 (Cont.) UI Framework Components

Component	Description
Model	The business logic communicates with beans through screen models. Each bean configurator contains a screen model, and the configurator must determine if any action is to be taken on the model. Classes exist for each model.
UI Manager	The UI Manager provides the API for application code to access and manipulate user interface components. The UI Manager uses different methods to call the UI Technician.
UI Technician	The UI Technician controls the main application window or display. The UI Technician receives calls from Point-of-Service tours, locates the appropriate screen, and handles the setup of the screens through the UI Subsystem.
UI Subsystem	The UI Subsystem provides UI components for displaying and editing Point-of-Service screens. The UI subsystem enables application logic to be completely isolated from the UI components. This component is specific to the technology used, such as Swing or JSP.
Adapters	Adapters are used to provide a specialized response to bean events. Adapters can handle the events, or the event can cause the adapter to manipulate a target bean. Adapters implement listener interfaces to handle events on the UI. Adapters come from the Swing API of controls and support JavaPOS-compliant devices.
Listeners	Listeners provide a mechanism for reacting to user interface events. Listeners come from the Swing API of controls and support JavaPOS-compliant devices.

Business Object

The Commerce Services layer of the architecture contains the Business Object framework that implements the instantiation of business objects. The Business Object framework is used to create new business objects for use by Mobile Point-of-Service. The business objects contain data and logic that determine the path or option used by an application.

Figure 9–3 Business Object Framework

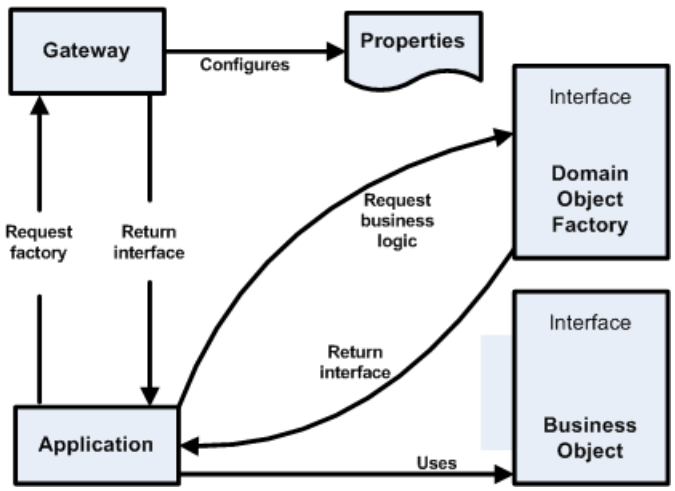


Table 9–2 describes the components in the Business Object framework.

Table 9–2 Business Object Framework Components

Component	Description
DomainGateway	The DomainGateway class provides a common access point for all business object classes.
Domain Object Factory	The Domain Object Factory returns instances of business object classes. The application requests a Factory from the DomainGateway.
Business Object	Business objects define the attributes for application data. New instances are created using the Domain Object Factory.

Data Persistence

A specific Manager/Technician pair is the Data Manager and Data Technician used for data persistence. Figure 9–4 illustrates how data gets saved to a persistent resource.

Figure 9–4 Data Persistence Framework

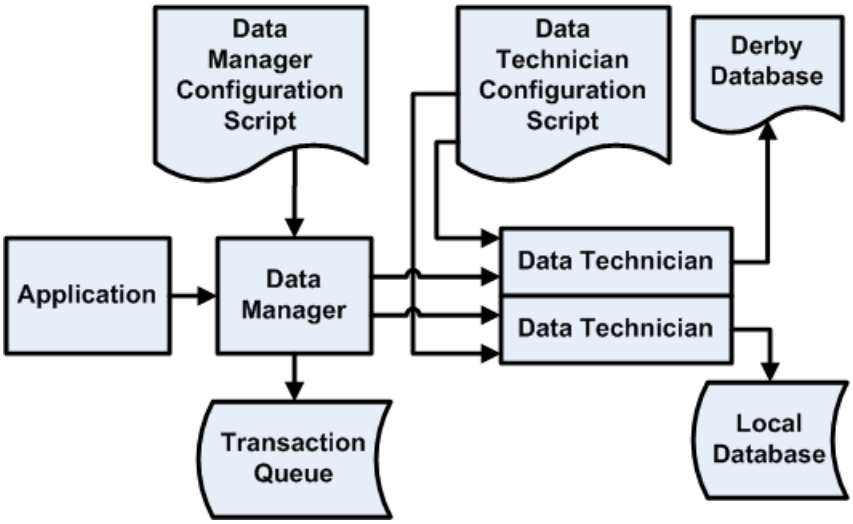


Table 9–3 describes the components in the Data Persistence framework.

Table 9–3 Data Persistence Framework Components

Component	Description
Data Manager	The Data Manager defines the application entry point into the Data Persistence Framework. Its primary responsibility is to contact the Data Technician and transport any requests to the Data Technician.
Data Manager Configuration Script	The Data Manager processes data actions from the application based on the configuration information set in the Data Manager Configuration Script. The Configuration Script defines transactions available to the application.

Table 9–3 (Cont.) Data Persistence Framework Components

Component	Description
Data Technician	The Data Technician provides the interface to the database or flat file. This class is part of the Oracle Retail Platform framework. It provides entry points for application transactions sent by the Data Manager and caches the set of supported data store operations. It also contains a pool of physical data connections used by the supported data operations.
Data Technician Configuration Script	The Data Technician Configuration Script specifies the types of connections to be pooled, the set of operations available to the application, and the mapping of an application data action to a specific data operation.
Transaction Queue	The Transaction Queue holds data transactions and offers asynchronous data persistence and offline processing for Point-of-Service. When the database is offline, the data is held in the queue and posted to the database when it comes back online. When the application is online, the Data Manager gets the information from the Transaction Queue to send to the database.

Tour

The Tour framework establishes the workflow for the application. It models application behavior as states, events and transitions. The Oracle Retail Platform engine is modeled on finite state machine behavior. A finite state machine has a limited number of possible states. A state machine stores the status of something at a given time and, based on input, changes the status or causes an action or output to occur. The Tour framework provides a formal method for defining these nested state machines as a traceable way to handle flow through an application.

Tourmap

One problem of tour scripts is that they can be difficult to customize for a particular retailer's installation. The tourmap feature allows customizations to be made more easily on existing tour scripts. Tour components and tour scripts can be referenced by logical names in the tour script and mapped to physical names in a tourmap file, making it easier to use the product tour and just change the pieces that need to be changed for a customer implementation. In addition, with tourmaps, components and scripts can be overridden based on a country, so files specific to a locale are implemented when appropriate.

The tourmap does not allow you to modify the structure of the tour, specifically the following:

- Does not allow you to add or remove sites
- Does not allow you to add or remove roads and aisles
- Does not allow you to specify a tour spanning multiple files (for example, "tour inheritance")

Of particular note is the last bullet: the tourmap does not allow you to assemble fragments of xml into one cohesive tour script. After the application is loaded, there is only be one tour script that maps to any logical name.

The functionality of tourmapping is implemented using one or more tourmap files. Multiple tourmap files can be specified using the config\tourmap.files properties. tourmap.files is a comma delimited list of tourmap files. As each file is loaded, the application checks the country property of the tourmap file. The order of files is significant because later files override any values specified in previous files. A file that overrides a similarly-named file is called an overlay.

Each tourmap file begins with a root element, tourmap, which has an optional country attribute. The tourmap elements contains multiple tour elements, each one of which describes a tour's logical name, its physical file, and any overlays to apply. For instance, a simple tourmap might look like the following:

Example 9–1 Sample Tourmap

```
<?xml version="1.0" encoding="UTF-8"?>
<tourmap
   country="CA"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="com/extendyourstore/foundation/tour/dtd/tourmap.xsd
">
    <tour name="testService">

<file>classpath://com/extendyourstore/foundation/tour/engine/tourmap.testservice.x
ml</file>
       <SITE
          name="siteWithoutAction"

useaction="oracle.retail.stores.foundation.tour.engine.actions.overlay.OverlaySite
Action"/>
       <SITEACTION
          class="SampleSiteAction"

replacewith="oracle.retail.stores.foundation.tour.engine.actions.overlay.OverlaySi
teAction"/>

    </tour>
</tourmap>
```

In this instance, the tour with the logical name **testService** references the file com\extendyourstore\foundation\tour\engine\tourmap.testservice.xml. Additionally, the values for SITE and SITEACTION are replaced.

> **Note:** Because of the country in the tourmap element, this only happens when the default locale of the application is a Canadian locale.

Tourmaps are used not only to override XML attributes, but they are used also when the workflow needs to be changed.

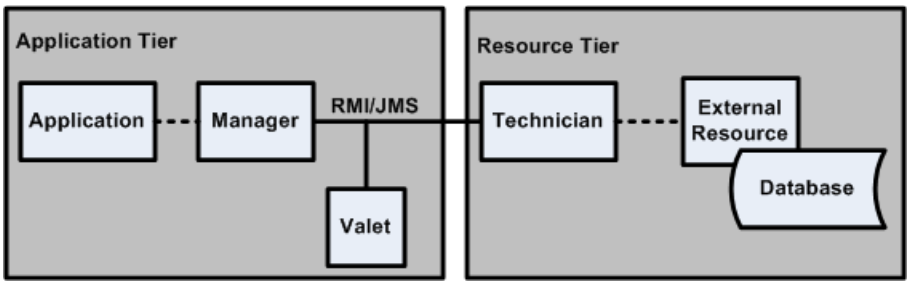
10

Manager/Technician Framework

This chapter describes the Manager/Technician pair relationship and how it is used to provide business and system services to the application. It also describes how to build a Manager and Technician and provides sample implementation and sample code.

Oracle Retail POS Suite provides the technology for distributing business and system processes across the enterprise through plug-in modules called Managers and Technicians. Manager and Technician classes come in pairs. A Manager is responsible for communicating with its paired Technician on the same or different tiers. The Technician is responsible for performing the work on its tier. By design, Managers know how to communicate with Technicians through a pass-through remote interface called a valet. The valet is the component that handles data transfer. The valet can travel across networks. It receives the instructions from the Manager and delivers them to the Technician. A valet follows the Command design pattern described in the *Oracle Retail POS Suite Implementation Guide – Volume 1, Implementation Solutions*.

Figure 10–1 Manager, Technician and Valet



There is a M:N relationship between instances of Managers and Technicians. Multiple Managers may communicate with a Technician, or one Manager may communicate with multiple Technicians. While most Managers have a corresponding Technician, there are cases such as the Utility Manager where no corresponding Technician exists.

There are three Manager/Technician categories. These types have different usages and are started differently. The three types are:

- Global—These Managers and Technicians are shared by all tours. They provide global services to applications.
- Session—These Managers and Technicians perform services for a single tour. They are started by each tour and exist for the length of the tour.
- Embedded—Thread Manager is embedded inside the Oracle Retail POS Suite engine. It is essential to the operation of the engine. This is currently the only embedded Manager.

Table 10–1 lists each of the three Manager/Technician categories, along with examples.

Table 10–1 Manager/Technician Type Examples

Manager/Technician Type	Examples
Global	▪ Data ▪ Journal ▪ Log ▪ Resource ▪ Timer ▪ Tier ▪ XML
Session	▪ Device ▪ Parameter ▪ UI ▪ FinancialNetworkManager
Embedded	Thread

Session Managers are started up by the tour bus when a tour is invoked and can only be accessed by the bus in the tour code. Global Managers, on the other hand, can be used at any time and are not specific to any tour. Each type of Manager has a specific responsibility.

Table 10–2 lists the functions of some of the Managers.

Table 10–2 Manager Names and Descriptions

Manager Name	Description
Data	The Data Manager is the system-wide resource through which the application can obtain access to persistent resources. The Data Manager tracks all data stores for the system, and is the mechanism by which application threads obtain logical connections to those resources for persistence operations.
Device	The Device Manager defines the Java interfaces that are available to an application or class for accessing hardware devices, like printers and scanners.
Journal	The Journal Manager is the interface that is used to write audit trail information, such as start transaction, end transaction, and other interesting register events.
Log	The Log Manager is the interface that places diagnostic output in a common location on one tier for an application, regardless of where the actual tours run.
Parameter	The Parameter Manager is the interface that provides access to parameters used for customization and runtime configuration of applications.
Thread	The Thread Manager is a subsystem that provides system threads as a pooled resource to the system.
Tier	The Tier Manager interface starts a tour session and mails letters to existing tour sessions. The Tier Manager enables the engine to start a tour on any tier specified in a transfer station, regardless of where that tier runs. In addition, the Tier Manager enables a bus to mail a letter to any other existing Bus in the system on any tier.

Table 10–2 (Cont.) Manager Names and Descriptions

Manager Name	Description
Timer	The Timer Manager provides timer resources to applications that require them. It does not have a Technician because all timers are local on the tier where they are used.
User Interface	The UI Manager is a mechanism for accessing and manipulating user interface components. The user interface subsystem within a state machine application must also maintain a parallel state of screens, so the appropriate screens can be matched with the application state at all times. The user interface subsystem within a distributed environment must enable application logic to be completely isolated from the user interface components.
XML	The XML Manager locates a specified XML file, parses the file, and returns an XML parse tree.

New Manager/Technician

When creating a new Manager and Technician pair, you must create a Manager and Technician class, a Valet class, and interfaces for each class. Managers are the application client to a Technician service, Technicians do the work, and the valet tells the Technicians what work to do. Managers can be considered proxies for the services provided by the Technicians. Technicians can serve as the interfaces to resources. Managers communicate with Technicians indirectly using valets. Valets can be thought of as commands to be executed remotely by the Technician. Samples for the new classes that need to be created are organized together in the next section.

Requesting services from the Managers only requires familiarity with the interface provided by Managers. However, building a new Manager/Technician pair requires implementing the interfaces for both the new Manager and Technician, as well as creating a Valet class.

Manager Class

A Manager merely provides an API to tour code. It behaves like any other method except that the work it performs may be completed remotely. The input to a Manager is usually passed on to the valet that in turn, passes it on to the Technician, which actually performs the work.

The Manager class provides methods for sending valets to the Technician. The sendValet() method makes a single attempt to send a valet to the Manager's Technician. The sendValetWithRetry() method attempts to send the valet to the Manager's Technician, and if there is an error, reset the connection to the Technician and then try again.

Managers must implement the ManagerIfc, which requires the methods in Table 10–3.

Table 10–3 ManagerIfc Methods

Method	Description
MailboxAddress getAddress()	Gets address of Manager
Boolean getExport()	Returns if this Manager is exportable
String getName()	Gets name of Manager
void setExport(Boolean)	Sets whether the Manager is exportable
void setName(String)	Sets name of Manager

Table 10–3 (Cont.) ManagerIfc Methods

Method	Description
void shutdown()	Shuts this Manager down
void startUp()	Starts this Manager

Often, a subclass of Manager can use these methods exactly as written. Unlike the Technicians, Managers seldom require special startup and shutdown methods, because most Managers have no special resources associated with them.

Manager Configuration

You can provide runtime configuration settings for each Manager using a conduit script. The Dispatcher that loads Point-of-Service configures the Managers by reading properties from the conduit script and calling the corresponding set() method using the Java reflection utility. All properties are set by the Dispatcher before the Dispatcher calls startUp() on the Manager.

Every Manager should have the following:

- Name—Tour code typically locates a Manager using its name. Often this name is the same as the name of the class and may be defined as a constant within the Manager. This is what the getName() method returns.
- Class—This is the name of the class, minus its package.
- Package—This is the Java package where the class resides.

Managers may have an additional property file defined that specifies additional information such as the definition of transaction mappings. If a separate configuration script is defined, the startup() method must read and interpret the configuration script. The following sample from *<source_directory>*\applications\pos\deploy\client\config\conduit\ClientConduit.xml shows this.

Example 10–1 CollapsedConduitFF.xml: Data Manager Configuration

```
<MANAGER name="DataManager" class="DataManager"
             package="oracle.retail.stores.foundation.manager.data">
        <PROPERTY propname="configScript"
                  propvalue="classpath://config/manager/PosDataManager.xml" />
</MANAGER>
```

Technician Class

Technicians implement functions needed by Point-of-Service to communicate with external or internal resources, such as the UI or the store database. Technicians must implement the TechnicianIfc, which requires the following methods in Table 10–4:

Table 10–4 TechnicianIfc Methods

Method	Description
MailboxAddress getAddress()	Gets address of Technician
Boolean getExport()	Checks if this Technician is exportable
String getName()	Gets name of Technician
void shutdown()	Shuts this Technician down

Table 10–4 (Cont.) TechnicianIfc Methods

Method	Description
void startUp()	Starts up Technician process

Often, a subclass of Technician can use these methods exactly as written. The most likely methods to require additional implementation are startUp() and shutdown(), which needs to handle connections with external systems.

Technician Configuration

The Technician is configured within the conduit script. Each Technician should have the following:

Name
A Manager typically locates its Technician using its name. Often this name is the same as the name of the class and may be defined as a constant within the Technician. This is what Technician.getName() returns.

Class
The name of the class, minus its package.

Package
The Java package where the class resides.

Export
This should be Y if the Technician may be accessed by an external Java process; N otherwise. The value returned by Technician.getExport() is based on this. In Technicians, it indicates whether the Technician can be remotely accessed from another tier.

Some Technicians may require complex configuration. In cases like this, it may be preferable to define an XML configuration script specific to the Technician, rather than to rely on the generic property mechanism. Therefore, Technicians may have an additional property defined that specifies additional information such as log formats or parameter validators. If a separate configuration script is defined, the startup() method must read and interpret the configuration script. The following sample from *<source_directory>*`\applications\pos\deploy\server\config\conduit\StoreServerConduit.xml` shows an additional script defined in the configuration of the Data Technician.

Example 10–2 CollapsedConduitFF.xml: Tax Technician Configuration

```
<TECHNICIAN name="RemoteDT" class = "DataTechnician"
               package = "oracle.retail.stores.foundation.manager.data"
               export = "Y" >
       <PROPERTY propname="dataScript"
                 propvalue="classpath://config/DefaultDataTechnician.xml"/>
   </TECHNICIAN>
```

Valet Class

The valet is the intermediary between the Manager and Technician. Valets act as commands and transport information back and forth between the Manager and Technician. Valets must implement ValetIfc, which contains a single method.

Table 10–5 lists the ValetIfc method.

Table 10–5 ValetIfc Method

Method	Description
Serializable execute(Object)	Executes the valet-specific processing on the object

The execute method is called by the Technician after the valet arrives at its destination as a result of the Manager's sendValet() or sendValetWithRetry() methods.

Sample Code

Example 10–3 illustrates the primary changes that need to be made to create a Manager/Technician pair. Note that interfaces also need to be created for the new Manager, Technician, and Valet classes.

Configuration

The conduit script needs to define the location of the Manager and Technician. This code would be found in a conduit script such as `config\conduit\ClientConduit.xml`. These code samples would typically be in different files on separate machines. It would include snippets like the following.

Example 10–3 Sample Manager and Technician Configuration

```
    <MANAGER name="MyNewManager"
             class="MyNewManager"
             package="oracle.retail.stores.foundation.manager.mynew">
    </MANAGER>

    <TECHNICIAN name="MyNewTechnician"
             class="MyNewTechnician"
             package="oracle.retail.stores.foundation.manager.mynew"
             export="Y" >
            <PROPERTY propname="techField" propvalue="importantVal"/>
            <PROPERTY propname="configScript"

propvalue="classpath://com/extendyourstore/pos/config/myconfigscript.xml"/>
    </TECHNICIAN>
```

Tour Code

Tour code might include a snippet like the following, which might be located in *<source_directory>*`\applications\pos\src\oracle\retail\stores\pos\services`.

Example 10–4 Sample Manager in Tour Code

```
        try
        {
            MyNewManagerIfc myManager =
(MyNewManagerIfc)bus.getManager("MyNewManager");
            myManager.doSomeClientWork("From site code ");
        catch (Exception e)
        {
            logger.info(bus.getServiceName(), e.toString());
        }
```

Manager

This is a minimal Manager class to illustrate how to create a new Manager. A new Manager interface also needs to be created for this class. Note that this class references the sample MyNewTechnician class shown in a later code sample.

Example 10–5 Sample Manager Class

```
package oracle.retail.stores.foundation.manager.mynew;

import oracle.retail.stores.foundation.manager.log.LogMessageConstants;
import oracle.retail.stores.foundation.tour.manager.Manager;
import oracle.retail.stores.foundation.tour.manager.ValetIfc;

public class MyNewManager extends Manager implements MyNewManagerIfc
{
    //--------------------------------------------------------------------------
    /**
       Constructs MyNewManager object, establishes the manager's address, and
       identifies the associated technician.
    */
    //--------------------------------------------------------------------------

    public MyNewManager()
    {
        getAddressDispatcherOptional();
        setTechnicianName("MyNewTechnician");
    }

    //--------------------------------------------------------------------------
    /**
        This method processes the input argument (via its technician).
        @param  input a String to illustrate argument passing.
        @return  a transformed String
    **/
    //--------------------------------------------------------------------------

    public String doSomeClientWork(String input)
    {
        String result = null;
        ValetIfc valet = new MyNewValet(input);
        try
        {
            result = (String)sendValetWithRetry(valet);
        }
        catch (Exception e) // usually ValetException or CommException
        {
            logger.error(LogMessageConstants.SCOPE_SYSTEM,
                         "MyNewManager.doSomeClientWork, " +
                         "could not sendValetWithRetry: Exception = {0}", e);
        }
        logger.debug(LogMessageConstants.SCOPE_SYSTEM,
                     "MyNewManager.doSomeClientWork, returns {0}", result);
        return result;
    }
}
```

Valet

The following code defines a valet to send input to MyNewTechnician.

Example 10–6 Sample Valet Class

```
package oracle.retail.stores.foundation.manager.mynew;

import oracle.retail.stores.foundation.tour.manager.ValetIfc;
import java.io.Serializable;

public class MyNewValet implements ValetIfc
{
    /** An input used by the Technician.  **/
    protected String input = null;
    //--------------------------------------------------------------------------
    /**
        The constructor stores the input for later use by the Technician.
        @param input the input to be stored.
    **/
    //--------------------------------------------------------------------------

    public MyNewValet(String input)
    {
        this.input = input;
    }

    //--------------------------------------------------------------------------
    /**
        This method causes the MyNewTechnician to "doSomething" with the input
            and returns the results.
        @param  techIn the technician that will do the work
        @return the results of "MyNewTechnician.doSomething"
    **/
    //--------------------------------------------------------------------------

    public Serializable execute(Object techIn) throws Exception
    {
        if (!(techIn instanceof MyNewTechnician))
        {
            throw new Exception("MyNewTechnician must passed into execute.");
        }
        MyNewTechnician tech = (MyNewTechnician)techIn;
        String result = tech.doSomething(input);
        return result;
    }
}
```

Technician

The following code provides an example of a minimal Technician class. A new Technician interface also needs to be created for this class.

Example 10–7 Sample Technician Class

```
package oracle.retail.stores.foundation.manager.mynew;

import oracle.retail.stores.foundation.manager.log.LogMessageConstants;
import oracle.retail.stores.foundation.tour.manager.Technician;
import oracle.retail.stores.foundation.tour.manager.ValetIfc;

public class MyNewTechnician extends Technician implements MyNewTechnicianIfc
{
    /** A value obtained from the config script.  **/
    protected String techField = null;
```

```
    public void setTechField(String value)
    {
        techField = value;
    }

    public void setConfigScript(String value)
    {
        // Complicated configuration could go here
    }

    //------------------------------------------------------------------------
    /**
        This method processes the input argument (via its Technician).
        @param  input a String to illustrate argument passing.
        @return  a transformed String
    **/
    //------------------------------------------------------------------------

    public String doSomething(String input)
    {
        String result = null;
        result = "MyNewTechnician processed " + input + " using " + techField;
        logger.debug(LogMessageConstants.SCOPE_SYSTEM,
                     "MyNewTechnician.doSomething, returns {0}", result);
        return result;
    }
}
```

Manager/Technician Reference

The following sections describe a Manager/Technician pair, important methods on the Manager, and an example of using the Manager in the application code.

Parameter Manager/Technician

The Parameter Manager is the interface that allows parameters to be used for customization and runtime configuration of applications. The following code from *<source_directory>*\applications\pos\deploy\client\config\conduit\ClientConduit.xml specifies the location and properties of the Parameter Manager and Technician. Note that the Parameter Manager is a Session Manager because it is defined with a PROPERTY element inside the APPLICATION tag. This means it can only be accessed using a tour bus.

Example 10–8 ClientConduit.xml: Code to Configure Parameter Manager

```
<APPLICATION name="APPLICATION"
                class="TierTechnician"
                package="oracle.retail.stores.foundation.manager.tier"

startservice="classpath://com/extendyourstore/pos/services/main/main.xml">
<PROPERTY propname="managerData"
propvalue="name=ParameterManager,managerpropname=className,managerpropvalue=oracle
.retail.stores.foundation.manager.parameter.ParameterManager"/>
<PROPERTY propname="managerData"

propvalue="name=ParameterManager,managerpropname=useDefaults,managerpropvalue=Y"/>
...
```

```
</APPLICATION>
```

Example 10–9 ClientConduit.xml: Code to Configure Parameter Technician

```
<TECHNICIAN name="ParameterTechnician" class = "ParameterTechnician"
               package = "oracle.retail.stores.foundation.manager.parameter"
               export = "Y" >
        <PROPERTY propname="paramScript"

propvalue="classpath://config/manager/PosParameterTechnician.xml"/>
</TECHNICIAN>
```

The Parameter Manager classes contain methods to retrieve parameter values. The Customization chapter describes details about where and how parameters are defined. A list of parameters can be found in the Parameter Names and Values Addendum.

The following code sample from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\inquiry\iteminquiry\AdvanceSearchSite.java illustrates the use of the Parameter Manager to retrieve parameter values.

Example 10–10 BrowserControlSite.java: Tour Code Using ParameterManagerIfc

```
ParameterManagerIfc pm =
(ParameterManagerIfc)bus.getManager(ParameterManagerIfc.TYPE);
Serializable[] values = pm.getParameterValues(ITEM_SEARCH_FIELDS);
```

UI Manager/Technician

The UI Manager/Technician is used to abstract the UI implementation. User events captured by the screen are sent to the UI Manager. The UI Manager communicates with a UI Technician, which updates the screen for a client running the UI. The UI Technician provides access to the application UI Subsystem. There is one UITechnician per application.

The model is an object that is used to transport information between the screen and the UI Manager using the UI Technician. Models and screens have a one-to-one relationship. The UI Manager allows you to set the model for a screen and retrieve a model for a screen; it knows which screen to show and which model is associated with the screen. The model has data members that map to the entry fields on the given screen. It can also contain data that dictates screen behavior, such as the field that has the starting focus or the state of a specific field.

The following code samples from *<source_directory>*\applications\pos\deploy\client\config\conduit\ClientConduit.xml specify the UI Manager and Technician properties. Like the Parameter Manager, the UI Manager can only be accessed using a tour bus.

Example 10–11 ClientConduit.xml: Code to Configure UI Manager

```
<APPLICATION name="APPLICATION"
                class="TierTechnician"
                package="oracle.retail.stores.foundation.manager.tier"

startservice="classpath://com/extendyourstore/pos/services/main/main.xml">
<PROPERTY propname="managerData"
propvalue="name=UIManager,managerpropname=className,managerpropvalue=oracle.retail
.stores.pos.ui.POSUIManager"/>
...configuration of other Managers...
</APPLICATION>
```

Example 10–12 ClientConduit.xml: Code to Configure UI Technician

```
<TECHNICIAN
        name="UITechnician"
        class="UITechnician"
        package="oracle.retail.stores.foundation.manager.gui" export="Y">

        <CLASS
            name="UISubsystem"
            package="oracle.retail.stores.pos.ui"
            class="POSJFCUISubsystem">

            <CLASSPROPERTY
                    propname="configFilename"

propvalue="classpath://com/extendyourstore/pos/config/defaults/defaultuicfg.xml"
                    proptype="STRING"/>
...
</TECHNICIAN>
```

The UI is configured in XML scripts. Each tour has its own uicfg file in which screen specifications are defined. The screen constants that bind to screen specification names are defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\POSUIManagerIfc.java. The UI Framework chapter discusses screen configuration in more detail.

POSUIManager is the UI Manager for the Point-of-Service application. One is started for each tour that is created.

Table 10–6 lists important POSUIManagerIfc methods, implemented in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\POSUIManager.java.

Table 10–6 Important POSUIManagerIfc Methods

Method	Description
void showScreen(String screenId, UIModelIfc beanModel)	Displays the specified screen using the specified model
UIModelIfc getModel(String screenId)	Gets the model from the specified screen
String getInput()	Gets the contents of the most recent Response area as a string
void setModel(String screenId, UIModelIfc beanModel)	Sets the model for the specified screen

These methods are used in tour code to display a screen, as in the following code from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\tender\check\GetCheckIDTypeSite.java.

Example 10–13 GetCheckInfoSite.java: Tour Code Using POSUIManagerIfc

```
POSUIManagerIfc ui = (POSUIManagerIfc) bus.getManager(UIManagerIfc.TYPE);
CheckEntryBeanModel model = new CheckEntryBeanModel();
 Locale lcl = LocaleMap.getLocale(LocaleConstantsIfc.USER_INTERFACE);
        if (personalIDTypes != null)
        {
        model.setIDTypes(personalIDTypes.getTextEntries(lcl));
        }
...set additional attributes...
```

```
ui.showScreen(POSUIManagerIfc.ENTER_ID, model);
```

Journal Manager/Technician

The Journal Manager provides location abstraction for journal facilities by implementing the JournalManagerIfc interface. By communicating with a JournalTechnicianIfc, the Journal Manager removes your need to know the location of resources. The Journal Technician is responsible for providing journal facilities to other tiers. The Journal Manager must be started on each tier that uses it. There must be a LocalJournalTechnician running on the local tier or an exported JournalTechnician running on a remote tier, or both. Transactions should be written to EJournal only when completed.

The following code samples from *<source_directory>*\applications\pos\deploy\client\config\conduit\ClientConduit.xml specify the Journal Manager and Technician properties. Note that this Manager is a Session Manager; it is defined outside of the APPLICATION element in which the UI Manager and Parameter Manager were defined. This allows the Journal Manager to be accessed outside of the bus, meaning it is more accessible and flexible.

Example 10–14 ClientConduit.xml: Code to Configure Journal Manager

```
<MANAGER name="JournalManager"
             class="JournalManager"
             package="oracle.retail.stores.foundation.manager.journal"
             export="N">
</MANAGER>
```

Example 10–15 ClientConduit.xml: Code to Configure Journal Technician

```
    <TECHNICIAN name="LocalJournalTechnician"
                class="JournalTechnician"
                package="oracle.retail.stores.foundation.manager.journal"
                export="Y">
    </TECHNICIAN>
```

The Journal Manager must be started on each tier that uses it. The Journal Manager sends journal entries in the following order: (1) Console if consolePrintable is set, (2) LocalJournalTechnician if defined, (3) JournalTechnician if defined.

Internationalizing EJournal Messages

To internationalize static texts, the hardcoded messages in Point-of-Service must be externalized to locale-specific resource bundle files and applications to use the resource bundle based on the configured Journal locale for EJournal. These modifications can be made either in-house or with the assistance of third-party system integrators.

If the modification efforts are not done correctly, the deployed product may not operate correctly. This situation causes serious issues for the retailer, Oracle Retail, and any system integrators involved. This section aims to mitigate that risk by providing guidance on how to safely and effectively make modifications to internationalize EJournal.

Internationalizing Static Texts

To internationalize static texts, the hardcoded messages in Point-of-Service will be externalized to locale-specific resource bundle files and applications to use the resource bundle based on the configured Journal locale for EJournal. Extend the

existing resource bundle ejournalText_<*language code*>.properties for EJournal static texts.

Internationalizing Transaction Data

Transaction EJ contains following type of information:

Database Data Transaction information such as Item Description, Reason code, Role Name, and so on, is retrieved from database for Journaling. This information must be in the Journal locale.

In the classes where Journal strings are hard-coded, use APIs provided by domain objects to get the data in Journal locale and use it for Journaling. Use Journal Locale configured in the Point-of-Service Client as argument to get the journal locale-specific data from domain objects.

Deprecate the existing toJournalString() method of domain classes and create new method toJournalString(locale) for preparing locale specific Journal String. Pass the Journal locale configured in the Point-of-Service Client application.properties file to toJournalString(locale) method of domain objects for Journal message.

Data Retrieved from Java Constants The information such as Tender type is defined as constant in Java file. Use resource bundle equivalent data for journaling instead of constant values.

For example, TenderTypeEnum has all the tender type constants defined. Get the journal locale equivalent data from EJournalText ResourceBundle for the value obtained from TenderTypeEnum. For Cash tender, define JournalEntry. Cash as key in EJournalText Resource bundle and get the value associated with it for journaling in formatter/domain classes.

Concatenated Strings Avoid journaling the messages, which need to be formed by concatenating two or more messages. However, if concatenated strings need to be logged, use argument-based messages in resource bundle and MessageFormat class to replace the arguments with the actual data before journaling.

For example, for the message **Buy 2 items and get a 25% discount**, we can prepare message such as **Buy {0} items and get {1} discount**. The arguments **0** and **1** can be replaced with the actual data.

DateTime and Currency Data The DateTime and Currency Data to be Journaled must be in the Journal locale format. Use the following:

- DateTimeServiceIfc.formatDate() to get the Journal locale format date
- DateTimeServiceIfc.formatTime() to get the Journal locale format time
- CurrencyServiceIfc.formatCurrency() to get the Journal locale format currency

Internationalization of Data Modification Event Messages

For Journaling the events such as Parameter value addition/modification, the EJournal message is prepared by concatenating one or more messages with actual data.

For example, the message **<<Parameter Name>> is modified** is formed by concatenating the Parameter Name and the constant string **is modified**. The approach to internationalize this data is same as internationalizing transaction data for concatenated Strings.

Persisting EJournal in UTF8 format

Do the following to persist the EJournal in the UTF8 format:

1. Modify the JL_TPE column type of the JL_ENR table from BLOB to CLOB.
2. Modify the existing EJournal text:
    a. Retrieve the data from the eJournalText bundle for the key in the class where it needs to be saved in EJournal.
    b. Convert the date, time, and currency data to the store locale format.
    c. Prepare the EJournal text using locale-specific journal static text, date, time and currency format.
3. Write the modified EJournal to the database and JMS using the existing framework.
4. Do the following to write the EJournal to a file:
    a. Modify the startUp method of the JournalTechnician to generate the EJournal text file name and create a JournalDisk object with the generated EJournal file name. The EJournal text file is generated by concatenating the JournalFileName and SequenceNumber values retrieved from JournalConfig.properties.
    b. Modify the store method of a JournalDisk class to write the EJournal string in UTF format instead of writing the EjournalEntry object in binary format.
    c. Modify the indexEntry method of the JournalTechnician to get the current EJournal file name.
    d. Modify the indexEntry method to append the journal file name to the index entry. The modified index entry looks like the following:

    ```
    0000000 0008 042411290119 09/02/2008 11:59 pos pos 129 journal_1.txt
    ```

5. If the current journal file size exceeds the configured limit:
    a. Generate the next sequence number and update the SequenceNumber property of the JournalConfig.properties file.
    b. Modify the indexEntry method of the JournalTechnician to generate the journal file name from the JournalFileName (containing the initial journal file name, such as journal.txt) and the SequenceNumber configured in JournalConfig.properties. The generated journal file name is of the format *JournalFileName_SequenceNumber* (for example, journal_1.txt).
    c. Close the currently open journal file and create a new journal file with the generated journal file name.
6. Modify the JdbcSaveJournalEntry class to contain updateClob method.
7. Create new classes DatabaseClobHelper and OracleDatabaseClobHelper and to update data to CLOB field.

Retrieve EJournal from Point-of-Service

Do the following to retrieve EJournal from Point-of-Service:

1. Modify classes DatabaseClobHelper and OracleDatabaseClobHelper to read EJournal text from CLOB field.
2. Do the following to read EJournal from a file:

a. Modify the searchJournal method of JournalTechnician to read the journal file name from index entry.

b. Modify the getEntry method of the JournalDisk to get the journal file name as an argument.

c. Modify the getEntry method of the JournalDisk class method to read the EJournal string from the given journal file in UTF format.

3. Modify the readJournalEntry method of JdbcReadJournalEntry class to read CLOB data instead of BLOB using OracleDatabaseClobHelper.

Display EJournal from Central Office

Do the following to display EJournal from Central Office:

1. Modify the JL_TPE column type of the JL_ENR table from BLOB to CLOB.
2. Add a new method readTapeClob to the class EJournalBean to read CLOB data and call this method from ejbLoad for setting tape data.
3. Add a new method updateTapeClob to the class EJournalBean to update CLOB data to the database and call this method from ejbCreate for setting tape data.

11

User Interface Framework

This chapter describes the User Interface (UI) Framework that is part of the Oracle Retail POS Suite architecture. The UI Framework encompasses all classes and interfaces included in Oracle Retail POS Suite to support rapid development of UI screens. It enables the building of custom screens using existing components.

For ease of development, the UI Framework hides many of the implementation details of Java UI classes and containment hierarchies by moving some of the UI specification from Java code into XML. This eases screen manipulation and layout changes affecting the look and feel of the entire screen, subsets of screens, and portions of a screen.

Table 11–1 provides a general description of features of the UI Framework.

Table 11–1 UI Framework Features

Feature	Description
Common Design	All UI implementations share code and extend or implement base UI classes that are provided as part of Oracle Retail POS Suite. The UI Framework provides basic functionality that does not need to be duplicated within each application.
Reuse	The UI Framework allows bean classes to be independent, thereby supporting their reuse. A UI Technician can be used with multiple applications and UI Framework components can be used across multiple features in an application.
Externally Configurable Screens	The UI Framework enables you to configure screens outside the code to accommodate applications that change frequently. The external screen configurations can be updated to use new Oracle Retail POS Suite or application-specific components as they are developed.
Support for Internationalization	The UI Framework provides hooks for implementing internationalization, including language and locale independence.
Extensibility and Flexibility	Additional formats for specifications can be defined without affecting the internal UI Framework classes. Portability is achieved through the use of the Java language and flexible layout managers.

The UI Framework is the set of classes and interfaces that define the elements and behavior of a window-based UI Subsystem. It defines a structure for defining user interfaces.

Table 11–2 briefly describes the components of the framework. This chapter discusses these components in more detail.

Table 11–2 UI Framework Components

Name	Description
Display	A display is the root container for the UI application window. Displays are any subclass of java.awt.Container that implement EYSRootPaneContainer.
Screen	A screen is a user-level snapshot of a UI window as it relates to an application. The screen is composed of displays, template areas, assignment beans, and listeners. Each of these parts can be individually configured and reassembled to compose the screen.
Template	A template divides the display into areas that contain the layout information used to place the information on the display. Templates can be interchanged to define screen layouts within an application. Each screen specifies the template that is associated with the screen.
Area	An area is a layout placeholder for UI components that operate together to perform a function. Each area contains a layout constraint that dictates how the area is placed on the display.
Bean	A bean is a user interface component or group of components that operate together to provide some useful functionality. For example, a bean could be an input form or group of navigation buttons.
Connection	A connection captures relationships between beans, or between devices and beans. When a bean or device generates an event, another bean responds with a change in behavior or visual display.
Listener	A listener provides a mechanism for reacting to user interface events.

Screens

Generally, for each package in an application, one UI script in the form of an XML file is created to define the screens for the given package. However, because many screens share basic components, certain components are defined in a default UI script. These basic screen components, including displays, templates, and default screens, are defined in `<source_directory>\applications\pos\src\oracle\retail\stores\pos\config\defaults\defaultuicfg.xml`. Overlay screens are then defined in the UI script for the given package. This section describes the components that are used to build Point-of-Service screens, except for beans which are described in the next section.

Displays define window properties. They are basic containers with dimensions and a title defined. In Point-of-Service, only two types of windows can be displayed at the same time—the main application window and a window displaying the Help browser.

Table 11–3 describes the two types of displays.

Table 11–3 Display Types

Name	Description
EYSPOSDisplaySpec	A 600x800 container for all application screens
HelpDialogDisplaySpec	A 600x800 container for Point-of-Service Help screens

Templates divide displays into geographical areas. The GridBagLayout is used to define the attributes of each area.

Table 11–4 describes the typical use of each template.

Table 11–4 Template Types

Name	Typical Use
BrowserTemplateSpec	Back Office screens within Point-of-Service application
EYSPOSTemplateSpec	Point-of-Service screens without required fields
HelpBrowserTemplateSpec	Point-of-Service help screens
ValidatingTemplateSpec	Point-of-Service screens with required fields that display an information panel below the work area

Default screens are partially-defined screens that represent elements common to multiple screens. Default screens are based on one display and one template. Default screens map beans to the commonly used areas of the template and define listeners for the bean. These screens are used by overlay specifications that define more specific screen components. For example, almost all screens in the Point-of-Service application display a status area region. The text displayed in the status region changes, but the StatusPanelSpec bean is the same from screen to screen, so a default screen would assign this bean to the StatusPanel area defined by a template.

Table 11–5 lists the areas of the template to which beans are assigned, and the display and template used by each of the six types of default screens.

Table 11–5 Default Screen Types

Name	Typical Use	Display	Template
BrowserDefaultSpec	Back Office screens within Point-of-Service application	EYSPOSDisplaySpec	BrowserTemplateSpec
DefaultHelpSpec	Point-of-Service help screens	HelpDialogDisplaySpec	HelpBrowserTemplateSpec
DefaultValidatingSpec	Point-of-Service screens with required fields that display an information panel below the work area	EYSPOSDisplaySpec	ValidatingTemplateSpec
EYSPOSDefaultSpec	Point-of-Service screens without required fields	EYSPOSDisplaySpec	EYSPOSTemplateSpec
ResponseEntryScreenSpec	Point-of-Service screens with information captured in the response area at the top of the screen	EYSPOSDisplaySpec	EYSPOSTemplateSpec

Each screen in Point-of-Service has an overlay screen defined in a UI script in the package to which it belongs or in a package higher in the hierarchy. For example, the Authorization tour script is found in `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\tender\authorization` but the UI script is located in `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\tender`. This overlay screen is based on a default screen and defines additional properties for the beans on the areas of the screen. The overlay screen may also specify connections, which are described in "Connections". The following code sample shows the definition of the ALTERATION_TYPE screen defined in `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\alterations\alterationsuicfg.xml`.

Example 11–1 alterationsuicfg.xml: Overlay Screen Definition

```
<OVERLAYSCREEN
 defaultScreenSpecName="EYSPOSDefaultSpec"
 resourceBundleFilename="alterationsText"
 specName="ALTERATION_TYPE">

        <ASSIGNMENT
            areaName="StatusPanel"
            beanSpecName="StatusPanelSpec">
            <BEANPROPERTY
                propName="screenNameTag" propValue="AlterationTypeScreenName"/>
        </ASSIGNMENT>

        <ASSIGNMENT
            areaName="PromptAndResponsePanel"
            beanSpecName="PromptAndResponsePanelSpec">
            <BEANPROPERTY
                propName="promptTextTag" propValue="AlterationTypePrompt"/>
        </ASSIGNMENT>

        <ASSIGNMENT
            areaName="LocalNavigationPanel"
            beanSpecName="AlterationsOptionsButtonSpec">
        </ASSIGNMENT>

</OVERLAYSCREEN>
```

Beans

A screen is composed of beans mapped to specific areas on the screen. All beans are defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\beans. The beans described in this section are commonly used in screen definitions. Bean properties that can be defined in assignments of beans to areas. Through Java reflection, properties defined in XML files invoke set() or create() methods in the bean class that accept a single string parameter or multiple string parameters.

The following section covers the PromptAndResponseBean, DataInputBean, NavigationButtonBean, and DialogBean.

PromptAndResponseBean

The PromptAndResponseBean configures and displays the text in the top areas of a Point-of-Service screen called the prompt region and the response region. This bean is implemented by *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\beans\PromptAndResponseBean.java and its corresponding model PromptAndResponseModel.java.

Bean Properties and Text Bundle

PromptAndResponsePanelSpec is the name of a bean specification that defines the implementation of the PromptAndResponseBean class. The following code sample shows the bean specification available to all screens, defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\config\defaults\defaultuicfg.xml.

Example 11–2 defaultuicfg.xml: Bean Specification Using PromptAndResponseBean

```
<BEAN
specName="PromptAndResponsePanelSpec"
beanClassName="PromptAndResponseBean"
beanPackage="oracle.retail.stores.pos.ui.beans"
configuratorPackage="oracle.retail.stores.pos.ui"
configuratorClassName="POSBeanConfigurator"
cachingScheme="ONE">
</BEAN>
```

Table 11–6 lists property names and values that can be defined in overlay specifications when specifying attributes of a PromptAndResponseBean.

Table 11–6 PromptAndResponseBean Property Names and Values

Item	Description	Example
enterData	Indicates whether data can be entered in the response area	true
promptTextTag	The label tag that corresponds to the text bundle	GiftCardPrompt
responseField	The type of field expected in the response area (see Field Type section for available types)	oracle.retail.stores.pos.ui.beans.AlphaNumericTextField
maxLength	Maximum length of response area input	15
minLength	Minimum length of response area input	2
zeroAllowed	Indicates whether a zero value is allowed in the response area	true
negativeAllowed	Indicates whether a negative value is allowed in the response area	false
grabFocus	Indicates whether focus should be grabbed when the screen is first displayed	true

These properties can be defined in UI scripts. The following code sample defines an overlay specification that assigns the PromptAndResponsePanelSpec defined above to the PromptAndResponsePanel. This example from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\tender\tenderuicfg.xml defines the COUPON_AMOUNT overlay screen for the Tender service. The property that retrieves text from a text bundle is highlighted.

Example 11–3 tenderuicfg.xml: PromptAndResponseBean Property Definition

```
<OVERLAYSCREEN>
        defaultScreenSpecName="ResponseEntryScreenSpec"
        resourceBundleFilename="tenderText"
        specName="COUPON_AMOUNT">
<ASSIGNMENT
            areaName="PromptAndResponsePanel"
            beanSpecName="PromptAndResponsePanelSpec">
            <BEANPROPERTY
                propName="promptTextTag" propValue="CouponAmountPrompt"/>
            <BEANPROPERTY
 propName="responseField"
 propValue="oracle.retail.stores.pos.ui.beans.CurrencyTextField"/>
            <BEANPROPERTY
 propName="maxLength" propValue="9"/>
</ASSIGNMENT>
```

```
...
</OVERLAYSCREEN>
```

The string that should be displayed as the prompt text is defined in a resource bundle. In the resource bundle for the Tender service, which for en locale is defined in *<source_directory>*\applications\pos\locales\en\config\ui\bundles\tenderText_en.properties, the following includes a line that defines the CouponAmountPrompt.

Example 11–4 tenderText_en.properties: PromptAndResponseBean Text Bundle Example

```
PromptAndResponsePanelSpec.CouponAmountPrompt=Enter coupon amount and press Next.
```

Tour Code

In the Tour code, bean models are created to hold the data on the bean components.

Table 11–7 lists some of the important methods in the PromptAndResponseModel class.

Table 11–7 PromptAndResponseModel Important Methods

Method	Description
boolean isSwiped()	Returns the flag indicating whether a card is swiped
void setsScanned(boolean)	Sets the flag indicating whether a code is scanned
boolean isResponseEditable()	Returns the flag indicating whether the response area is editable
void setGrabFocus(boolean)	Sets the flag indicating whether focus should stay on the response field

Example 11–5 is a sample from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\modifyitem\ModifyItemQuantitySite.java that shows creation of a PromptAndResponseModel, prefilling of data in the model, and display of the model on which the PromptAndResponseModel is set.

Example 11–5 ModifyItemQuantitySite.java: Creating and Displaying PromptAndResponseModel

```
 POSBaseBeanModel baseModel = new POSBaseBeanModel();
        PromptAndResponseModel beanModel = new PromptAndResponseModel();
        UtilityManagerIfc utility =
          (UtilityManagerIfc) bus.getManager(UtilityManagerIfc.TYPE);
        Locale locale = LocaleMap.getLocale(LocaleConstantsIfc.USER_INTERFACE);
        String unitId = UNITID_TEXT;
        if ((uom == null) || (unitId.equals(uom.getUnitID())))
        {

beanModel.setResponseText(Integer.toString(lineItem.getItemQuantityDecimal().intVa
lue()));
        }
        else
        {
            beanModel.setArguments(uom.getName(locale));
            String responseText =
LocaleUtilities.formatNumber(lineItem.getItemQuantityDecimal(),LocaleMap.getLocale
(LocaleConstantsIfc.USER_INTERFACE));
            beanModel.setResponseText(responseText);
            screenID = POSUIManagerIfc.ITEM_QUANTITY_UOM;
```

```
        }

        baseModel.setPromptAndResponseModel(beanModel);
        ui.showScreen(screenID, baseModel);
```

The screen constant, ITEM_QUANTITY_UOM, is mapped to an overlay screen name found in the UI script for the package. The screen constants are defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\POSUIManagerIfc.java.

Example 11–6 is a sample from ItemQuantityModifiedAisle.java in the same directory that shows how to retrieve data from the PromptAndResponseModel in a previous screen. To arrive at this code, a new quantity for an item is entered and the user presses **Next**. This line of code gets the new quantity from the previous screen.

Example 11–6 ItemQuantityModifiedAisle.java: Retrieving Data From PromptAndResponseModel

```
POSUIManagerIfc ui=(POSUIManagerIfc)bus.getManager(UIManagerIfc.TYPE);
        String quantity = ui.getInput();
```

DataInputBean

The DataInputBean is a standard bean that displays a form layout containing data input components and labels. This bean is implemented by *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\beans\DataInputBean.java and its corresponding model DataInputBeanModel.java. Field components are commonly defined with the FIELD element when defining a bean with the DataInputBean, as shown in the code sample.

Bean Properties and Text Bundle

The DataInputBean has two properties that can be defined in UI scripts, which override the settings in the field specifications.

Table 11–8 lists DataInputBean property names and values.

Table 11–8 DataInputBean Property Names and Values

Item	Description	Example
labelTags	Sets the property bundle tags for the component labels	NameLabel,AddressLabel,StateLabel
labelTexts	Sets the text on the component labels	Name,Address,State

The label tag is used for internationalization purposes, so the application can look for the correct text bundle in each language. The label tag overrides the value of the labelText field. Example 11–7 is code from manageruicfg.xml that shows a field specification defined in a DataInputBean bean specification.

Example 11–7 manageruicfg.xml: Bean Specification Using DataInputBean

```
<BEAN
        specName="RegisterStatusPanelSpec"
        configuratorPackage="oracle.retail.stores.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="oracle.retail.stores.pos.ui.beans"
        beanClassName="DataInputBean">
```

```
        <FIELD fieldName="storeID"
               fieldType="displayField"
               labelText="Store ID:"
               labelTag="StoreIDLabel"
               paramList="storeNumberField"/>
 ...
</BEAN>
```

The strings that should be displayed as labels on the screen are defined in a resource bundle. In the resource bundle for the Manager service, which for the en locale is defined in *<source_directory>*\applications\pos\locales\en\config\ui\bundles\managerText_en.properties, Example 11–8 is a line of code that defines the StoreIDLabel.

Example 11–8 managerText_en.properties: DataInputBean Text Bundle Example

```
RegisterStatusPanelSpec.StoreIDLabel=Store ID:
```

Fields do not have to be defined in the UI script. They can be defined in the beans and models instead. In the overlay screen specification, two bean properties that can be set are OptionalValidatingFields and RequiredValidatingFields. If the fields are optional and the user enters information in them, then they are validated. If the user does not enter any information, the fields are not validated. On the other hand, required fields are always validated.

Tour Code

Bean models are created to hold the data managed by the bean. This protects the bean from being changed. A bean can only be accessed by a model in the Tour code.

Table 11–9 lists some of the important methods in the DataInputBeanModel class.

Table 11–9 DataInputBeanModel Important Methods

Method	Description
String getValueAsString(String)	Returns the value of the specified field as a string
int getValueAsInt(String)	Returns the value of the specified field as an integer
void setSelectionValue(String, Object)	Sets the value of the specified field in a vector to the specified value
void setSelectionChoices(String, Vector)	Sets the value of the specified field to the specified vector of choices
void clearAllValues()	Clears the values of all the fields

Example 11–9 is a sample from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\admin\parametermanager\SelectParamStoreSite.java that shows creation of a DataInputBeanModel and prefilling of data in the model.

Example 11–9 SelectParamStoreSite.java: Creating and Displaying DataInputBeanModel

```
DataInputBeanModel beanModel = new DataInputBeanModel();
beanModel.setSelectionChoices("choiceList", vChoices);
beanModel.setSelectionValue("choiceList", (String)vChoices.firstElement());
```

Example 11–10 is a sample from Tour code that shows how to retrieve data from the DataInputBeanModel. In this example from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\admin\pa

rametermanager\StoreParamGroupAisle.java, after the model is created and displayed by the code from the previous code sample, the model is retrieved from the UI Manager, and data is retrieved from the model.

Example 11–10 StoreParamGroupAisle.java: Retrieving Data from DataInputBeanModel

```
DataInputBeanModel model =
(DataInputBeanModel)ui.getModel(POSUIManagerIfc.PARAM_SELECT_GROUP);
ParameterCargo cargo = (ParameterCargo)bus.getCargo();
String val = (String)model.getSelectionValue("choiceList");
cargo.setParameterGroup(val);
```

NavigationButtonBean

The NavigationButtonBean represents a collection of push buttons and associated key stroke equivalents. This bean is implemented by *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\beans\NavigationButtonBean.java and its corresponding model NavigationButtonBeanModel.java. The global navigation area and the local navigation area both use the NavigationButtonBean.

Bean Properties and Text Bundle

The LocalNavigationPanel and GlobalNavigationPanel bean specifications both use the NavigationButtonBean. Bean properties are described only for the GlobalNavigationPanelSpec because the LocalNavigationPanelSpec typically sets its properties in the bean specification and not the overlay specification.

LocalNavigationPanel The only property available to the NavigationButtonBean in XML is used to enable and disable buttons. When setting the states of buttons on a LocalNavigationPanel, the buttons are usually defined with the BUTTON element in the bean specification as in the following code sample. In fact, any bean that extends NavigationButtonBean, such as ValidateNavigationButtonBean, can define its buttons in the bean specification.

Example 11–11 from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\customer\customeruicfg.xml, defining the CustomerOptionsButtonSpec bean specification for the Customer service, shows how button text on a NavigationButtonBean is defined in a UI script.

Example 11–11 customeruicfg.xml: Bean Specification Using NavigationButtonBean

```
<BEAN
        specName="CustomerOptionsButtonSpec"
        configuratorPackage="oracle.retail.stores.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="oracle.retail.stores.pos.ui.beans"
        beanClassName="NavigationButtonBean">

<BUTTON
            actionName="AddBusiness"
            enabled="true"
            keyName="F4"
            labelTag="AddBusiness"/>
...
</BEAN>
```

The string that should be displayed on the buttons on the GlobalNavigationPanel is defined in a resource bundle. In the resource bundle customerText_en.properties, the following entry defines the label for the AddBusiness button.

Example 11–12 customerText_en.properties: NavigationButtonBean Text Bundle Example

```
CustomerOptionsButtonSpec.AddBusiness= Add Business
```

GlobalNavigationPanel The GlobalNavigationButtonBean extends the NavigationButtonBean. The following code sample shows the GlobalNavigationPanel bean specification defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\config\defaults\defaultuicfg.xml. The bean class is a subclass of NavigationButtonBean.

Example 11–13 defaultuicfg.xml: Bean Specification Using GlobalNavigationButtonBean

```
<BEAN
 specName="GlobalNavigationPanelSpec"
 configuratorPackage="oracle.retail.stores.pos.ui"
 configuratorClassName="POSBeanConfigurator"
 beanPackage="oracle.retail.stores.pos.ui.beans"
 beanClassName="GlobalNavigationButtonBean"
 cachingScheme="ONE">
...
</BEAN>
```

Table 11–10 lists property names and values that can be defined in UI scripts when specifying attributes of a GlobalNavigationButtonBean.

Table 11–10 GlobalNavigationButtonBean Property Names and Values

Item	Description	Example
manageNextButton	Indicates whether the bean should manage the enable property of the Next button	true
buttonStates	Sets the buttons with the action names listed to the specified state	Help[true],Clear[false],Cancel[false], Undo[true],Next[false]

These properties can be defined in overlay specifications, as in the following code sample from tenderuicfg.xml.

Example 11–14 tenderuicfg.xml: GlobalNavigationButtonBean Property Definitions

```
<OVERLAYSCREEN>

defaultScreenSpecName="EYSPOSDefaultSpec"
        resourceBundleFilename="tenderText"
specName="TENDER_OPTIONS">
 <ASSIGNMENT
            areaName="GlobalNavigationPanel"
            beanSpecName="GlobalNavigationPanelSpec">
            <BEANPROPERTY
                propName="manageNextButton"
                propValue="false"/>
            <BEANPROPERTY
                propName="buttonStates"

propValue="Help[true],Clear[false],Cancel[false],Undo[true],Next[false]"/>
```

```
        </ASSIGNMENT>
...
</OVERLAYSCREEN>
```

Tour Code

In the Tour code, bean models are created to hold the data on the bean components.

Table 11–11 lists some of the important methods in the NavigationButtonBeanModel class.

Table 11–11 NavigationButtonBeanModel Important Methods

Method	Description
ButtonSpec[] getNewButtons()	Returns an array of new buttons
void setButtonEnabled(String, boolean)	Sets the state of the specified action name of the button (the name of the letter the button mails)
void setButtonLabel(String, String)	Sets the label of the button using the specified action name of the button (the name of the letter the button mails)

The following sample from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\customer\lookup\CustomerSearchOptionsSite.java shows creation of a NavigationButtonBeanModel, prefilling of data in the model, and display of the model on which the NavigationButtonBeanModel is set.

Example 11–15 CustomerSearchOptionsSite.java: Creating and Displaying NavigationButtonBeanModel

```
NavigationButtonBeanModel nModel = new NavigationButtonBeanModel();

        if (cargo.isAddCustomerEnabled())
        {
            nModel.setButtonEnabled(CustomerCargo.EMPID, true);
            nModel.setButtonEnabled(CustomerCargo.CUSTINFO, true);
        }
        else
        {
            nModel.setButtonEnabled(CustomerCargo.EMPID, false);
            nModel.setButtonEnabled(CustomerCargo.CUSTINFO, false);
        }

        if (cargo.isAddBusinessEnabled())
        {
            nModel.setButtonEnabled(CustomerCargo.BUSINFO, true);
        }
        else
        {
            nModel.setButtonEnabled(CustomerCargo.BUSINFO, false);
        }

        model.setLocalButtonBeanModel(nModel);

        ui.showScreen(POSUIManagerIfc.CUSTOMER_SEARCH_OPTIONS, model);
```

The screen constant, CUSTOMER_SEARCH_OPTIONS, is mapped to an overlay screen name found in the UI script for the package. The screen constants are defined in *<source_*

directory>\applications\pos\src\oracle\retail\stores\pos\ui\POSUIManagerIfc.java.

DialogBean

The DialogBean provides dynamic creation of dialog screens. This bean is implemented by *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\beans\DialogBean.java and its corresponding model DialogBeanModel.java.

Bean Properties and Text Bundle

DialogSpec is the name of a bean specification that defines an implementation of the DialogBean class. The following code sample shows the bean specification defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\common\commonuicfg.xml.

Example 11–16 commonuicfg.xml: Bean Specification Using DialogBean

```
<BEAN
specName="DialogSpec"
configuratorPackage="oracle.retail.stores.pos.ui"
configuratorClassName="POSBeanConfigurator"
beanPackage="oracle.retail.stores.pos.ui.beans"
beanClassName="DialogBean">
<BEANPROPERTY propName="cachingScheme" propValue="none"/>
</BEAN>
```

The DialogBean does not have any properties that can be defined in UI scripts. Therefore, all its properties are defined in Tour code discussed in the next section. The following code sample defines the message displayed in the dialog. This example from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\inquiry\giftcardinquiry\InquirySlipPrintAisle.java shows how text on a DialogBean is defined in Java code.

Example 11–17 InquirySlipPrintAisle.java: DialogBean Label Definition

```
 DialogBeanModel model = new DialogBeanModel();
            model.setResourceID(RETRY_CONTINUE_TAG);
            model.setType(DialogScreensIfc.RETRY_CONTINUE);
            model.setButtonLetter(DialogScreensIfc.BUTTON_CONTINUE, "ExitPrint");
            model.setArgs(msg);
```

The resourceID corresponds to the name of the text bundle. For all dialog screens in the en locale, dialogText_en.properties contains the bundles that define the text on the screen, as shown in the following code.

Example 11–18 dialogText_en.properties: DialogBean Text Bundle Example

```
DialogSpec.Retry.title=Device is offline
DialogSpec.Retry.description=Device offline
DialogSpec.Retry.line2=<ARG>
DialogSpec.Retry.line5=Press the Retry button to attempt to use the device again.
```

Tour Code

In the Tour code, bean models are created to hold the data on the bean components.

Table 11–12 lists some of the important methods in the DialogBeanModel class.

Table 11–12 DialogBeanModel Important Methods

Method	Description
setResourceID(String)	Used to locate screen text in the text bundle
setArgs(String [])	Sets a string of arguments to replace <ARG> tags in the text bundle
setButtonLetter(int, String)	Sets the specified letter to be sent when the specified button is pressed
setType(int)	Sets the flag indicating whether focus should stay on the response field

The following sample from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\dailyoperations\register\registeropen\AcknowledgeRegisterOpenSite.java shows creation of a DialogBeanModel, prefilling of data in the model, and display of the model on which the DialogBeanModel is set.

Example 11–19 LookupStoreCreditSite.java: Creating and Displaying DialogBeanModel

```
DialogBeanModel dialogModel = new DialogBeanModel();
DialogModel.setResourceID("RegisterOpenPromptConfirmation");
String args[] = new String[1];
        args[0] =
String.valueOf(cargo.getRegister().getWorkstation().getWorkstationID());
    model.setArgs(args);
dialogModel.setType(DialogScreensIfc.ACKNOWLEDGEMENT);
ui.showScreen(POSUIManagerIfc.DIALOG_TEMPLATE, dialogModel);
```

The screen constant, DIALOG_TEMPLATE, is mapped to an overlay screen name found in the UI script for the package. The screen constants are defined in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\POSUIManagerIfc.java.

When setting the dialog type, refer to the following table. For each dialog type, the buttons on the dialog are specified. In most cases, the letter sent by the button has the same name as the button, except for the two types noted.

Table 11–13 lists some of the available dialog types as defined by constants in *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ui\DialogScreensIfc.java.

Table 11–13 Dialog Types

Dialog Type	Buttons	Details
ACKNOWLEDGEMENT	Enter	Button sends OK letter
CONFIRMATION	Yes, No	NA
CONTINUE_CANCEL	Continue, Cancel	NA
ERROR	Enter	Button sends OK letter, Screen displays red in the title bar
RETRY	Retry	NA
RETRY_CANCEL	Retry, Cancel	NA
RETRY_CONTINUE	Retry, Continue	NA
SIGNATURE	NA	Places a signature panel to capture the customer's signature

When setting a letter to a button, refer to the following table that lists the available button types also defined in DialogScreensIfc.java. These constants are used as arguments to DialogBean methods that modify button behavior.

Table 11–14 lists some of the available button types also defined in DialogScreensIfc.java.

Table 11–14 Button Types

Button	ButtonID
Enter, OK	BUTTON_OK
Yes	BUTTON_YES
No	BUTTON_NO
Continue	BUTTON_CONTINUE
Retry	BUTTON_RETRY
Cancel	BUTTON_CANCEL

Field Types

This section defines field types available to all beans. The following field types may be used by all the beans, but DataInputBean is the only bean that understands the FIELD element. In other words, DataInputBean is the only bean that defines fields in bean specifications.

These field types correspond to create() methods in **UIFactory.java**, such as createCurrencyField() and createDisplayField(). The application framework uses reflection to create the fields. Therefore, the field names in the following table can be set as the fieldType attribute in an XML bean specification using the DataInputBean class. The corresponding parameter list is a list of strings that can be set as the paramList attribute.

Table 11–15 lists field types and their descriptions.

Table 11–15 Field Types and Descriptions

Name	Description	Parameter List Strings (no spaces allowed)
alphaNumericField	Allows letters and/or numbers, no spaces or characters	name,minLength,maxLength
constrainedPasswordField	Text where the view indicates something was typed, but does not show the original characters	name,minLength,maxLength
constrainedTextAreaField	Multi-line area that allows plain text, with restrictions on length	name,minLength,maxLength,columns,wrapStyle,lineWrap
constrainedField	Allows letters, numbers, special characters, and punctuation, with restrictions on length	name,minLength,maxLength
currencyField	Displays Currency objects in a localized format	name,zeroAllowed,negativeAllowed,emptyAllowed
decimalField	Allows decimal numbers only	name,maxLength,negativeAllowed,emptyAllowed
displayField	Display area that allows a short text string or an image, or both	name

Table 11–15 (Cont.) Field Types and Descriptions

Name	Description	Parameter List Strings (no spaces allowed)
driversLicenseField	Allows alphanumeric text that can contain '*' or ' '	name
EYSDateField	Displays a date object in a localized format	name
EYSTimeField	Displays a time object in a localized format	name
nonZeroDecimalField	Allows non-zero decimal numbers only	name,maxLength
numericField	Allows integers only, no special characters or letters	name,maxLength,minLength
nonZeroNumericField	Allows non-zero integers only	name,maxLength,minLength
textField	Allows letters, numbers, special characters, and punctuation	name
validatingTextField	Line of text that can be validated by length requirements	name

Multi-byte Support For Input Fields

The current UI framework allows the display and entering of multi-byte characters and the proper display and handling of the UI elements.

Multi-byte characters occupy up to four times more space than single-byte characters. In order to support these bigger sizes, the database columns need to be updated, the number of digits entered in the UI can be updated and the size of the fields in the UI needs to be adjusted.

The following are the extensions added to the UI framework subsystem:

1. Enable the configuration of maxLength for all input elements.
2. Enable the configuration of size (the width of the display field) for all input elements.
3. Enable UI fields to accept multi-byte characters.
4. Enable the UI text fields to be populated with data that is greater, up to the value of maxLength.

UI Framework Architecture for Input Fields

The main component of the UI framework subsystem is UIFactory.java. This class contains methods to create and configure all the UI Elements. It is used by the DataInputBean.java file and by all the custom beans that define the UI elements of a specific screen.

The following classes represent the UI input fields for alphanumeric characters:

- ConstrainedTextField – It is a base class and it allows input fields to accept all characters.
- AlphaNumericTextField – Allows fields to accept only alphanumeric characters.

The DataInputBean.java class implements a wide variety of UI elements. It is configured by using a *uicfg.xml file. For further information see DataInputBean.

Example 11–20 DataInputBean.java Class

```
<BEAN
    specName="PurchaseOrderAgencyNameSpec"
    configuratorPackage="oracle.retail.stores.pos.ui"
    configuratorClassName="POSBeanConfigurator"
    beanPackage="oracle.retail.stores.pos.ui.beans"
    beanClassName="DataInputBean">

    <FIELD fieldName="businessNameField"
           fieldType="ConstrainedField"
           labelText="Agency/Business Name:"
           labelTag="AgencyBusinessName"
           paramList="businessNameField,2,30 />
</BEAN>
```

The following are definitions for the <FIELD> excerpt:

FieldName
The logical name of the field.

FieldType
The type of the field. When reading the configuration file, by using reflexion, the UIFactory.createConstrainedField is invoked.

LabelText
The default label in case there is no resource bundle defined for this label.

LabelTag
The resource bundle property name.

ParamList
Name of the field, minimum length and maximum length. When reading the configuration file, by using reflexion, the UIFactory.createConstrainedField (String fieldName, String minLength, String maxLength)is invoked.

Custom Beans are java classes in which all UI elements can be defined.They ares not as configurable as the DataInputBean.java class, but they are widely used in Oracle Retail Point-of-Service.

These beans are referenced in the *uicfg.xml files but the definition of the UI elements are done in java.

Example 11–21 is an example of a custom bean java class:

Example 11–21 customeruicfg.xml

```
<BEAN
        specName="CustomerAddSpec"
        configuratorPackage="oracle.retail.stores.pos.ui"
        configuratorClassName="POSBeanConfigurator"
        beanPackage="oracle.retail.stores.pos.ui.beans"
        beanClassName="CustomerAddBean">
</BEAN>
```

Updating MaxLength and Size of Multi-byte Fields

Do the following to update the maxLength and size of those fields that can support multi-byte characters:

1. Identify the field to be updated in the UI.

If the field is created inside a Custom Bean, open the class and look for the `initializeField` method. This method creates and initializes all screen UI elements.

Use the overloaded constructors from the UIFactory to create the field with the new maxLength:

```
UIFactory.createConstrainedField(String name, String minLength, String
maxLength, String columns)
UIFactory.AlphaNumericTextField createAlphaConstrainedField(String name, String
minLength, String maxLength, String columns)
UIFactory.ValidatingComboBox createValidatingComboBox (String name, String
emptyAllowed, String columns)
```

2. Identify the size of the equivalent database column. The new field maxLength is the database column size divided by four.

   The current maxLength will be the new size/columns. The new field maxLength will be the DB Column size divided by 4.

   > **Note:** If the size isn't passed as an input parameter, then internally, **size = maxLength**.

   For dropdown boxes, no maxLength or size is specified.

   To make it a standard when defining the new size of the field, use the size of the longest element displayed in the input box. For example, suppose the database column `FirstName` in the database was updated from **16** to **80**.

   Update from:

   ```
   firstNameField = uiFactory.createConstrainedField("firstNameField", "2", "16");
   ```

   to:

   ```
   firstNameField = uiFactory.createConstrainedField("firstNameField", "2", "80",
   "16");
   ```

3. Identify where this field is being created. Look into the *uicfg.xml which represents the current UI screen.

   If the field is created inside a DataInputBean, the entire configuration is done in the *uicfg.xml.

   From:

   ```
   <FIELD fieldName="firstNameField"
          fieldType="ConstrainedField"
          labelText="First Name:"
          labelTag="firstName"
          paramList="firstNameField,2,16" />
   ```

   to:

   ```
   <FIELD fieldName="firstNameField"
          fieldType="ConstrainedField"
          labelText="First Name:"
          labelTag="firstName"
          paramList="firstNameField,2,80,16" />
   ```

   `UIFactory.createConstrainedField(String name, String minLength, String maxLength, String columns)` is called.

4. Determine if the field to be updated is being created in a Custom Bean or through the DataInputBean. In the *uicfg.xml file, look for the <BEAN> that contains the screenSpec. Look into the beanClassName attribute. It has the name of the bean class that creates the UI elements for the screen.

Allowing or Disallowing UI Fields to Accept UTF8 Characters

Do the following to allow or disallow UI fields to accept UTF8 characters:

1. Identify the field to be updated in the UI.

   If the field is created inside a Custom Bean, open the class and look for the initializeField method. This method creates and initializes all screen UI elements.

   Use the overloaded constructors from the UIFactory to set the doubleByteCharsAllowed flag to **false**. By default the flag is set to **true**.

   ```
   UIFactory.createConstrainedField(String name, String minLength, String
   maxLength, String columns, false)
   ```

   Use this constructor if you also want to change the maxLength and size.

   ```
   UIFactory.createConstrainedField(String name, String minLength, String
   maxLength, false)
   ```

   Use this constructor if you want to keep the same maxLength and size.

   ```
   UIFactory.AlphaNumericTextField createAlphaConstrainedField(String name, String
   minLength, String maxLength, String columns, false )
   ```

   Use this constructor if you also want to change the maxLength and size.

   ```
   UIFactory.AlphaNumericTextField createAlphaConstrainedField(String name, String
   minLength, String maxLength, false )
   ```

   Use this constructor if you do not want to change the maxLength and size.

   ```
   UIFactory.DriversLicenseTextField createDriversLicenseField(String name, String
   minLength, String maxLength, String columns, false )
   ```

   Use this constructor if you also want to change the maxLength and size.

   ```
   UIFactory.DriversLicenseTextField createDriversLicenseField(String name, String
   minLength, String maxLength, false )
   ```

   Use this constructor if you do not want to change the maxLength and size.

2. Identify where this field is being created. Look into the *uicfg.xml which represents the current UI screen.

   If the field is created inside a DataInputBean, the entire configuration is done in the *uicfg.xml.

   From:

   ```
   <FIELD fieldName="firstNameField"
          fieldType="ConstrainedField"
          labelText="First Name:"
          labelTag="firstName"
          paramList="firstNameField,2,16" />
   ```

   to:

   ```
   <FIELD fieldName="firstNameField"
          fieldType="ConstrainedField"
   ```

```
        labelText="First Name:"
        labelTag="firstName"
        paramList="firstNameField,2,80,16, false /> 
```

when you want to change the maxLength and size.

Or:

```
<FIELD fieldName="firstNameField"
       fieldType="ConstrainedField"
       labelText="First Name:"
       labelTag="firstName"
       paramList="firstNameField,2,16, false />
```

when you do not want to change the maxLength and size.

One of the following is called:

- `UIFactory.createConstrainedField(String name, String minLength, String maxLength, String columns, false )`
- `UIFactory.createConstrainedField(String name, String minLength, String maxLength,  false )`

3. Determine if the field to be updated is being created in a Custom Bean or through the DataInputBean. In the *uicfg.xml file, look for the `<BEAN>` that contains the `screenSpec`. Look into the beanClassName attribute. It has the name of the bean class that creates the UI elements for the screen.

Connections

Connections configure the handling of an event in the UI Framework. They are used to define inter-bean dependencies and behavior and to tie the bean event responses back to the business logic. When one bean generates an event, another bean can be notified of the event. Connections have a source bean, a Listener Type for the target, and a target bean.

Connections attach a source bean to a target bean, which receives event notifications from the source bean. The Listener Type specifies which type of events can be received. The XML in the following sections is found in *<source_directory>*`\applications\pos\src\oracle\retail\stores\pos\services\tender\tenderuicfg.xml`. Other listeners used in Point-of-Service include ConfirmCancelAction, HelpAction, and CloseDialogAction.

ClearActionListener

ClearActionListener is an interface that extends ActionListener in Swing to make it unique for its use in Point-of-Service. The following code shows how this listener is used in an overlay specification. Adding the ClearActionListener allows **Clear** to erase the text in the selected field in the work area when **Clear** on the GlobalNavigationPanelSpec is clicked.

Example 11–22 tender.xml: ClearActionListener XML tag

```
<CONNECTION
       listenerInterfaceName="ClearActionListener"
       listenerPackage="oracle.retail.stores.pos.ui.behavior"
       sourceBeanSpecName="GlobalNavigationPanelSpec"
       targetBeanSpecName="CreditCardSpec"/>
```

DocumentListener

DocumentListener is an interface defined in Swing. The following code shows how this listener is used in an overlay specification. Adding the DocumentListener allows the Clear button on the GlobalNavigationPanelSpec to be disabled until input is entered in the selected field on the work area.

Example 11–23 tender.xml: DocumentListener XML tag

```
<CONNECTION
            sourceBeanSpecName="CreditCardSpec"
            targetBeanSpecName="GlobalNavigationPanelSpec"
            listenerPackage="javax.swing.event"
            listenerInterfaceName="DocumentListener"/>
```

ValidateActionListener

ValidateActionListener is an interface that extends ActionListener in Swing to make it unique for its use in Point-of-Service. The following code shows how this listener is defined in an overlay specification. Adding the ValidateActionListener allows the CreditCardSpec to recognize when the Next button on the GlobalNavigationPanelSpec is clicked, resulting in the validation of the required fields on the work area. If the required fields are empty, an error dialog appears stating that the required field(s) must have data.

Example 11–24 tender.xml: ValidateActionListener XML tag

```
<CONNECTION
listenerInterfaceName="ValidateActionListener"
listenerPackage="oracle.retail.stores.pos.ui.behavior"
sourceBeanSpecName="GlobalNavigationPanelSpec"
targetBeanSpecName="CreditCardSpec"/>
```

The fields that are required must be specified for this listener in the overlay specification for the target bean, as in the following XML from tenderuicfg.xml.

Example 11–25 tenderuicfg.xml: ValidateActionListener Required Fields

```
<ASSIGNMENT
   areaName="WorkPanel"
   beanSpecName="CreditCardSpec">
   <BEANPROPERTY
      propName="RequiredValidatingFields"
propValue="CreditCardField,ExpirationDateField"/>
   </ASSIGNMENT>
```

Text Bundles

Currently, over forty text bundles exist for the Point-of-Service application. Many of these bundles are service-specific. A properties file with the same name exists for every language, located in *<source_directory>*\applications\pos\locales\<locale name>\config\ui\bundles with the language name appended to the filename. For example, the Customer service would have its text defined in the customerText_en.properties file in English.

A similarly named properties file would exist for each locale. Because they are discussed earlier in the chapter, service-specific bundles and the dialogText bundle are not described in this section.

parameterText

In overlay specifications, the parameterText bundle is specified to define the text for particular screens. For example, the following code from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\admin\parametermanager\parameteruicfg.xml defines text for the PARAM_SELECT_PARAMETER overlay screen. On this screen, the names of the parameters found in the parameterText properties file are displayed.

Example 11–26 parameteruicfg.xml: Overlay Specification Using parameterText

```
<OVERLAYSCREEN
        defaultScreenSpecName="EYSPOSDefaultSpec"
        resourceBundleFilename="parameterText"
        specName="PARAM_SELECT_PARAMETER">
```

In the utility package, the ParameterManager is used to retrieve parameter values. The following code from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\utility\GiftCardUtility.java shows how a parameter is retrieved from the ParameterManager. The handle to the ParameterManager, pm, is passed into the method but originally retrieved by the code ParameterManagerIfc pm = (ParameterManagerIfc)bus.getManager(ParameterManagerIfc.TYPE);

Example 11–27 GiftCardUtility.java: Tour Code to Retrieve Parameter

```
public static final String DAYS_TO_EXPIRATION_PARAMETER =
"GiftCardDaysToExpiration";
daysToExpiration = pm.getIntegerValue(DAYS_TO_EXPIRATION_PARAMETER);
```

In the parameterText_<language>.properties file, the corresponding text is defined. This text is displayed on the Parameter List screen when viewing Security options and choosing the Tender parameter group.

Example 11–28 parameterText_en.properties: Text Bundle

```
Common.GiftCardDaysToExpiration=Days To Giftcard Expiration
```

The value of the parameter is defined in *<source_directory>*\applications\pos\deploy\shared\config\parameter\application\application.xml by the code sample below. Each parameters belongs to a group, a collection of related parameters.

Example 11–29 application.xml: Definition of Parameter

```
<PARAMETER name="GiftCardDaysToExpiration"
         type="INTEGER"
         final="N"
         hidden="N">
         <VALIDATOR class="IntegerRangeValidator"
            package="oracle.retail.stores.foundation.manager.parameter">
            <PROPERTY propname="minimum" propvalue="1" />
            <PROPERTY propname="maximum" propvalue="9999" />
         </VALIDATOR>
         <VALUE value="365"/>
</PARAMETER>
```

12

Oracle Retail Tour Framework

The Tour framework is a component of the Oracle Retail Platform layer of the Point-of-Service architecture. The Tour framework implements a state engine that controls the workflow of the application. Tour scripts are a part of this framework; they define the states and transitions that provide instructions for the state engine that controls the workflow. Java classes are also part of this framework; they implement the behavior that is accessed by the tour engine, based on instructions in the tour scripts.

Tour Components

The tour metaphor helps the user visualize how the Oracle Retail Platform engine interacts with application code. In the following description of the metaphor, the words in **italics** are part of a simple tour script language that Oracle Retail Platform uses to represent the application elements.

Tour Metaphor

For a moment, imagine that you are a traveler about to embark on a journey. You have the itinerary of a business traveler (changeable at any time), your luggage, and transportation. In addition, you have a video camera (TourCam) to record your tour so you can remember it later.

You leave on your journey with a specific goal to achieve. Your itinerary shows a list of tours that you can choose from to help you accomplish your task. Each tour provides a tour bus with a cargo compartment and a driver. Each driver has a map that shows the various service regions that you can visit. These regions are made up of sites (like cities) and transfer stations (bus stations, airports, and so forth). The maps show a finite number of lanes, which are either roads joining one site to another or aisles within one site. To notify the driver to start the bus and drive, you must send a letter to the driver. The driver reads the name on the letter and looks for a lane that matches the letter.

When a matching letter is found, the driver looks for a traffic signal on the road. If there is no signal, the driver can traverse the road. If there is a signal, the driver can traverse the road only if the signal is green. If the signal is red, the driver attempts to traverse the next alternative road that matches the letter. If the driver cannot find any passable road, he or she returns to the garage. When you arrive at a site or traverse a lane, you may perform an action to achieve your goal, like take a picture of the countryside.

Upon arriving at a transfer station, you immediately transfer to another service, and you load a portion of your cargo onto a shuttle and board the shuttle. The shuttle takes you and your cargo to the bus that runs in the map of the other tour. Upon arrival at the new bus, you unload the shuttle and load the new bus. Then the new driver starts

the bus and your journey begins in the new tour. When the transfer tour itinerary is complete, you load whatever cargo you want to keep onto a shuttle and return to the original tour bus. At that time, you unload the shuttle and continue your tour.

These tour script components map to terms in the metaphor. The tour metaphor provides labels and descriptions of these components that improve understanding of the system as a whole.

Table 12–1 includes a metaphor description and a technical description for the basic metaphor components.

Table 12–1 Metaphor Components

Name	Metaphor Description	Technical Description
Service	A group of related cities, for example "A Mediterranean Tour"	An implementation of workflow and behavior for a set of functionality
Bus	The vehicle that provides transportation from city to city	The entity that follows the workflow between the sites
Cargo	The baggage that the traveler takes with him/her from city to city	The data that follows the workflow, modified as necessary
Site	A city	A function point in the workflow
Road	A path the bus takes to get from one city to another	A transition that takes place based on an event that changes the state
Aisle	A path the traveler takes while staying on the same bus in the same city	An action that takes place based on an event, without leaving the current state
Letter	A message the bus driver receives instructing him/her to perform an action	A message that causes a road or aisle to be taken

When given a use case, create a tour script by identifying components for the tour metaphor. Strategies for identifying components are listed in the table below. The following sections describe each component in more detail.

Table 12–2 includes strategies for identifying components.

Table 12–2 Component Identification Strategies

Component	How to Identify
Service	A service generally corresponds to a set of related functionality.
Site	Sites generally correspond to points in the workflow that need input from outside the tour. Outside input sources include the user interface, the database, and devices among others.
Road	At a site, look at the ways control can be moved to another site. There is one road for each of these cases.
Aisle	At a site, there might be a task that you want to handle in a separate module and then return to the site when the task is complete. There is one aisle for each of these cases.
Letter	Letters generally correspond to buttons on a UI screen and responses from the database and devices. Look for the events that move control from one site to another or prompt additional behavior within a site to help identify letters.

Follow the naming conventions in the Development Standards when deciding the names for the components. It is important to understand that the tour metaphor is not only used to describe the interaction of the components, but the component's names

are used in the code. By convention, a site named GetTender has a Java class in the package named GetTenderSite.java that performs the work done at the site.

Service and Service Region

Tours provide a way of grouping related functionality to minimize maintenance and increase reusability. All tours provide a bus to maintain state and cargo for data storage. All sites, lanes, and stations contained within a tour have access to these resources. Furthermore, in the Point-of-Service source code, the tours are found in the `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services` directory. Generally, this chapter uses the word tour to refer to a tour. The word **service** and phrase **service region** are used in this section because they are elements in the XML code.

The service region contains all functionality related to running the application when no exceptions are encountered. The following code sample from `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\tender\tender.xml` shows the definition of a service and service region in a tour script.

Example 12–1 tender.xml: Definition of Service and Service Region

```
<SERVICE name="Tender" package="oracle.retail.stores.pos.services.tender"
tourcam="ON">
<SERVICECODE>
...definition of letters, siteaction classes, and laneaction classes...
</SERVICECODE>
<MAP>
<REGION region="SERVICE" startsite="GetTender">
...definition of sites, stations, and lanes...
</REGION>
</MAP>
</SERVICE>
```

As shown in the code sample, there are two main sections of a tour script. The SERVICECODE element defines the Java classes in the tour and the letters that may be sent in the tour code or by the user. The MAP element links the classes and letters to the sites and lanes. In the following sections, code samples are shown from both sections of the tour script.

Bus

The bus object is passed as a parameter to all tour methods called by the tour engine. Methods can be called on the bus to get access to the cargo, managers and other state information.

Cargo

Cargo is data that exists for the length of the tour in which it is used. Any data that needs to be used at different tour components such as sites and aisles needs to be stored on the cargo. Cargo always has a Java class. The following code sample from tender.xml defines the Tender cargo.

Example 12–2 tender.xml: Definition of Cargo

```
<CARGO class="TenderCargo">
</CARGO>
```

With the concept of a tourmap, a cargo class can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class.

Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Example 12–3 tourmap.xml: Example of Overriding Cargo Class

```
<CARGO class="TenderCargo"
replacewith="oracle.retail.stores.cargo.SomeCargo"/>
```

Sites

Sites correspond to nodes in a finite state machine and cities in the tour metaphor. Sites are usually used as stopping places within the workflow. Arrival at a site usually triggers access to an external interface, such as a graphical user interface, a database, or a device. Sites always have a corresponding siteaction class.

The tender.xml code sample below contains the site information from the two main parts of a tour script: the XML elements SERVICECODE and MAP, respectively.

Example 12–4 tender.xml: Definition of Site Class

```
<SITEACTION  class="GetTenderSite"/>
```

Example 12–5 tender.xml: Mapping of Site to SiteAction

```
<SITE  name="GetTender" siteaction="GetTenderSite">
... definition of lanes ...
</SITE>
```

With the concept of Tourmap, a site's siteaction can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Example 12–6 tourmap.xml: Overriding Siteaction With Tourmap

```
<SITEACTION
class="GetTenderSite"replacewith="oracle.retail.stores.actions.SomeOtherSiteAction
"/>
```

System Sites

System sites are defined by the Oracle Retail Platform engine but can be referenced within a tour script. For example, a road defined by a tour script can have a system site as its destination. Each system site must have a unique name in the tour script file. The following code from tender.xml shows the definition of two system sites. The Final system site stops a bus and returns it to the parent bus, and LastIndexed resumes the normal bus operation after an exception.

Example 12–7 tender.xml: Definition of System Sites

```
<REGION>
<MAP>
```

```
...definition of sites, lanes, and stations...
<SYSTEMSITE name="Final" action="RETURN" />
<SYSTEMSITE name="LastIndexed" action="BACKUP" />
</MAP>
</REGION>
```

Letters

Letters are messages that get sent from the application code or the user interface to the tour engine. Letters indicate that some event has occurred. Typically, letters are sent by the external interfaces, such as the graphical user interface, database, or device to indicate completion of a task.

Lanes are defined as roads and aisles. When the system receives a letter, it checks all lanes defined within the current site or station to see if the letter matches the letter for a lane. If no matching lane is found, the letter is ignored. Letters do not have a Java class associated with them.

Standard letter names are used in the application, such as Success, Failure, Undo, and Cancel. The following code sample shows tender.xml code that defines letters. The definition is added to the SERVICECODE XML element.

Example 12–8 tender.xml: Definition of Letter

```
 <LETTER  name="Credit"/>
```

Roads

Roads provide a way for the bus to move between sites and stations. Each road has a name, destination, and letter that activates the road. A road may have a laneaction class, depending on whether the road has behavior; only roads that have behavior require a class. Roads are defined within site definitions because they handle letters received at the site.

Following is tender.xml code that shows the definition of a road. The definition is added to the SERVICECODE XML element. After the first code sample is another sample that maps the road to a site and letter, which is contained in the MAP section of the tour script.

Example 12–9 tender.xml: Definition of Road Class

```
<LANEACTION  class="ValidCreditInfoEnteredRoad"/>
tender.xml: Mapping of Road to Site
<SITE  name="GetCreditInfo" siteaction="GetCreditInfoSite">
  <ROAD
    name="ValidCreditInfoEntered"
    letter="Valid"
    laneaction="ValidCreditInfoEnteredRoad"
    destination="GetTender"
    tape="ADVANCE"
    record="OFF"
    index="OFF">
  </ROAD>
...other lanes defined...
</SITE>
```

With the concept of Tourmap, a road's laneaction can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following

tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Example 12–10 tourmap.xml: Example of Overriding Site Laneaction

```
<LANEACTION class="ValidCreditInfoEnteredRoad"
replacewith="oracle.retail.stores.actions.SomeOtherLaneAction"/>
```

Common Roads

The COMMON element is defined in the REGION element of the tour script. The COMMON element can contain roads that are available to all sites and stations in a tour. Common roads have the same attributes as roads defined within a site, but they are defined outside of a site so they can be accessed by all sites. If a common road and a tour road are both activated by the same letter, the site road is taken. The following is an example that differentiates common roads from tour roads.

Example 12–11 Example of Common Road

```
<MAP>
     <REGION region="SERVICE" startsite="Example">
          <COMMON>
               <ROAD name="QuitSelected" letter="exit"
                    destination="NamedIndex"
                    tape="REWIND"/>
                    <COMMENT>
                    </COMMENT>
               </ROAD>
          </COMMON>
          <SITE name="RequestExample" siteaction="RequestExampleSite">
               <ROAD name="ExampleSelected" letter="next"
                     laneaction="ExampleSelectedRoad"
                     destination="ShowExample"
                     tape="ADVANCE"
                     record="OFF"
                     index="ON"/>
                    <COMMENT>
                    </COMMENT>
               </ROAD>
     </REGION>
</MAP>
```

Aisles

Aisles provide a means for moving within a site and executing code. Aisles are used when a change is required but there is no reason to leave the current site or station. Each aisle contains a name, a letter, and a laneaction. Aisles always require a Java class because they must have behavior since they do not lead to a different site or station like roads.

Following is the tender.xml code that shows the definition of an aisle. The definition is added to the SERVICECODE XML element. The second code sample from the same tour script maps an aisle to the site and letter, which is contained in the MAP section.

Example 12–12 tender.xml: Definition of Aisle Class

```
<LANEACTION  class="CardInfoEnteredAisle"/>
```

Example 12–13 tender.xml: Mapping of Aisle to Site

```
<SITE  name="GetCreditInfo" siteaction="GetCreditInfoSite">
  <AISLE
    name="CardInfoEntered"
    letter="Next"
    laneaction="CardInfoEnteredAisle">
  </AISLE>
...other lanes defined...
</SITE>
```

With the concept of Tourmap, an aisle's laneaction can be overridden with another class. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The following tourmap definition specifies the class to override and the new class to use in place of the original class. Note that replacewith is a fully qualified classname, with both package and classname specified, unlike the class attribute.

Example 12–14 tourmap.xml: Example of Overriding Aisle Laneaction

```
<LANEACTION class="CardInfoEnteredAisle"
replacewith="oracle.retail.stores.actions.SomeOtherLaneAction"/>
```

Stations and Shuttles

Transfer stations are used to transfer workflow to another tour and return once the tour workflow has completed. A transfer station describes a location where another tour is started and the passenger exits one bus and enters the bus for another tour.

Transfer stations specify the name of the nested tour and define data transport mechanisms called shuttles. Shuttles are used to transfer cargo to and from the nested tour. These shuttles are either launch shuttles or return shuttles. Launch shuttles transfer cargo to the nested tour and the return shuttles transfer newly acquired cargo from the nested tour to the calling tour. Shuttles have Java classes associated with them, but stations do not.

The following code samples from `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\tender\tender.xml` contain the station and shuttle information from the SERVICECODE and MAP elements in the tour script, respectively.

Example 12–15 tender.xml: Definition of Shuttle Class

```
  <SHUTTLE  class="TenderAuthorizationLaunchShuttle"/>
```

Example 12–16 tender.xml: Mapping of Station to Service and Shuttle Classes

```
  <STATION
    name="AuthorizationStation"

servicename="classpath://com/extendyourstore/pos/services/tender/authorization/Aut
horization.xml"
    targettier="APPLICATIONTIER"
    launchshuttle="TenderAuthorizationLaunchShuttle"
    returnshuttle="TenderAuthorizationReturnShuttle">
    ...lane definitions to handle exit letter from nested service...
</STATION>
```

The servicename can be defined as a logical name like "authorizationService" and mapped to a filename is the tourmap file. The shuttle names can also be overridden in

the tourmap file. This allows you to override the class name for a customer implementation yet still keep the same workflow for the customer as in the product. The code samples below illustrate this.

Example 12–17 tourmap.xml: Example of Mapping Servicename

```
<tour name="authorizationService">
<file>classpath://com/extendyourstore/pos/services/tender/authorization/Authorizat
ion.xml</file>
</tour>
```

Example 12–18 tourmap.xml: Example of Overriding Shuttle Name

```
<SHUTTLE class="TenderAuthorizationLaunchShuttle"
replacewith="oracle.retail.stores.shuttles.NewShuttle"/>
```

Nested tours operate independently, with their own XML script and Java classes. Stations and shuttles simply provide the functionality to transfer control and data between two independent tours.

Signals

Signals direct the tour to the correct lane when two or more lanes from the same site or station are activated by the same letter. The lane that has a signal that evaluates to true is the one that is traversed. Each signal has an associated Java class. Signal classes evaluate the contents of the cargo and do not modify data.

The following code sample lists the tender.xml code that relates to the definition of two roads with Light signals defined. The definition is added to the SERVICECODE XML element, whereas the road description is added to the MAP XML element. The negate tag negates the Boolean value returned by the specified signal class.

Example 12–19 tender.xml: Definition of Traffic Signal

```
  <SIGNAL  class="IsAuthRequiredSignal"/>
```

Example 12–20 tender.xml: Signal Processing With Negate Tag

```
<STATION>
  name="AuthorizationStation"
  <ROAD
    name="AuthorizationRequested"
    letter="Next"
    destination="AuthorizationStation"
    tape="ADVANCE"
    record="OFF"
    index="OFF">
    <LIGHT  signal="IsAuthRequiredSignal"/>
  </ROAD>
  <ROAD
    name="BalancePaid"
    letter="Next"
    destination="CompleteTender"
    tape="ADVANCE"
    record="OFF"
    index="OFF">
    <LIGHT  signal="IsAuthRequiredSignal" negate="Y"/>
  </ROAD>
  ...additional lane definitions...
</STATION>
```

Exception Region

Continuing the tour metaphor, the bus could break down at any time. If the bus driver detects that the bus has broken down, the bus driver takes the bus to the nearest Garage system site. Once the bus is in the garage, the mechanic assumes control of and diagnoses the breakdown.

- If the mechanic is able to restore the cargo to a valid state, the mechanic informs the bus driver by traversing to the Resume system site. The bus driver subsequently resumes driving by resetting the bus at the site where the breakdown occurred.
- If the mechanic is not successful in repairing the bus, the mechanic stops the bus, and mails the parent tour a letter informing it of the breakdown.
- If there is no mechanic within the tour, the bus driver stops the bus, and mails the parent tour a letter informing it of the breakdown. The bus completes its tour when it arrives at the final site.

The exception region includes the functionality for handling exceptions. It can contain sites, roads, and stations just like the service region. There are two ways to exit the exception region: at the Return system site or the Resume system site. Return shuts down the application, and Resume starts the application at the last visited site or station in the service region.

The mechanic operates within the exception region of the tour. Any exception that occurs within the tour region where the bus driver operates is converted to an Exception letter and is passed to the mechanic. When the exception is being processed, the mechanic assumes control of the bus and processes all incoming letters. If the application developer has created an exception region for the mechanic, the Exception letter is processed using application-specific actions and traffic lights. However, if the exception region does not exist, the mechanic stops the bus and informs the parent bus of the problem.

Depending on the application definition, recovery from exceptions can result in a rollback, resumption, or a restart of the bus.

Role of Java Classes

All the code samples in this chapter have been from tour scripts. Tour scripts exist in the form of one XML file per tour. The tour script refers to Java classes that implement specific behavior, such as the siteaction and laneaction attributes. A tour has the following Java classes, one for:

- The cargo
- Each site
- Each aisle
- Each road that implements behavior
- Each shuttle
- Each signal

Table 12–3 lists methods that the tour engine looks for when it arrives at a specified place in the tour.

Table 12–3 System-Called Methods

Class	Methods
Site	arrive(), depart()
Road (if behavior)	traverse()
Aisle	traverse()
Shuttle	load(), unload()
Signal	roadClear()
Cargo	<none>

Tour Cam

TourCam allows you to navigate backward through your application in a controlled manner while requiring minimal programming to accomplish the navigation. It provides the ability to back up from a tour or process by tracking the state of the cargo and the location of the tours. TourCam is turned on or off at the tour level. If there is no reason to back up, TourCam should not be turned on.

The ability to backup or restore data to a previous state is accomplished using TourCam. TourCam is used to record the bus path through the map, as well as the associated cargo changes. TourCam is described using the TourCam metaphor. The words in italics in the following paragraphs are the TourCam-specific terms.

A bus driver records the progress along the bus route using TourCam. The bus driver records snapshots of the passenger cargo immediately before traversing a road. Each snapshot is mounted in a frame within the current tape. The frame is stamped with the current road. Using this method, the bus driver can retrace steps through the map. If the frame is indexed, the driver stops at that index when retracing his steps.

The bus driver may adjust the TourCam tape while the bus traverses a road between sites.

- The bus driver can advance the current TourCam tape, and add the next road and snapshot of the cargo as a frame in the tape.
- The bus driver can discard the current TourCam tape, and replace it with a blank tape.
- The bus driver can rewind the current tape to restore the cargo to be consistent with a previously visited site.
- The bus driver can splice the current TourCam tape by removing all frames that were recorded since a previously visited site.

When the passenger wants to back up, they instruct the bus driver to traverse a road whose destination is the Backup system site. The backup road can inform the bus driver to rewind or splice the TourCam tape while retracing its path along the last recorded road. Similarly, the passenger can instruct the bus driver to traverse a road to a specific, previously visited site. That road effectively backs up the bus when it instructs the bus driver to rewind or splice the TourCam tape.

When the passenger wants to end the trip, they instruct the bus driver to travel down a road whose destination is the Return system site. The final road may advance or discard the TourCam tape. A passenger may return to the tour if they back into the parent transfer station. If the TourCam tape is advanced, a return visit retraces the path through the map in reverse order. If the TourCam tape is discarded, all return visits start at the start site, as if the passenger were visiting the tour for the first time.

Attributes

The TourCam processing model places all undo actions on roads and treats sites and stations as black boxes. The tour attribute that turns TourCam on or off is tourcam. The following code from tender.xml shows the location in the tour script where the tourcam is set. The default value is OFF.

Example 12–21 tender.xml: Definition of tourcam

```
<SERVICE
    name="Tender"
    package="oracle.retail.stores.pos.services.tender"
    tourcam="ON">
```

The rest of the TourCam attributes are set on the road element in the MAP section of the tour script. The following code from tender.xml shows a road definition with these attributes set.

Example 12–22 tender.xml: Definition of Road With TourCam Attributes

```
<SITE name="GetGiftCertificateInfo" siteaction="GetGiftCertificateInfoSite">
               <ROAD name="GiftCertificateInfoEntered"
                     letter="Next"
                     laneaction="GiftCertificateInfoEnteredRoad"
                     destination="GetTender"
                     tape="ADVANCE"
                     record="OFF"
                     index="OFF">
...definitions of lanes...
</SITE>
```

Table 12–4 lists TourCam attributes and their values.

Table 12–4 Road Tag Element Attributes

Tag	Description	Values	Default
tape	Indicates what tour action to take when traversing the road.	ADVANCE – Adds a frame representing this road to the tourcam tape DISCARD – Discards the entire tour cam tape REWIND – Back up to the site specified by the 'destination' while calling the backup method on all roads SPLICE – Back up to the site specified by the 'destination' without calling the backup method on any roads	ADVANCE
record	Indicates that a snapshot of the cargo should be recorded and saved on the tourcam tape	ON – Record a snapshot OFF – Do not record a snapshot	ON

Table 12–4 (Cont.) Road Tag Element Attributes

Tag	Description	Values	Default
index	Indicates that an index should be placed on the tourcam tape when this road is traversed	ON – Place an index on the tape OFF – Do not place an index on the tape	ON
namedIndex	Indicates that a named index should be placed on the tourcam tape when this road is traversed	Any string value is allowed	None
destination	Used when the tape has a value of REWIND or SPLICE to indicate where the tourcam should back the bus up to	<SITENAME> – The name of a site to back up to. The site must be in the current tour. LastIndexed – The backup should end at the site that is the origin of the first road found with an unnamed index. NamedIndex – The backup should end at the at the site that is the origin of the first road found with the named index specified by the named Index.	None

Each of the following combinations describes a combination of settings and how it is useful in different situations. The following tables describe the forward and backward TourCam settings:

Table 12–5 describes the forward TourCam settings.

Table 12–5 Forward TourCam Settings

Settings	Behavior
ADVANCE index=ON record=ON	This combination permits you to return to the site without specifying it as a destination and storing the state of the cargo. Use this combination if you are entering data and making decisions. The UI provides a method for backing up to the previous step.
tape=ADVANCE index=OFF record=ON	This combination allows you to track visited sites, and allows you to attach undo behavior. However, you cannot back up to this site. A common scenario for use would be for performing external lookups and the user must backup to the site that started the lookup. This combination is used, rather than the following combination, when changes made to the cargo that must be reversible.
tape=ADVANCE index=OFF record=OFF	This combination is useful for sites that require external setup from another site, but do not result in a significant change in cargo. You cannot back up to a site that uses these settings and you cannot restore cargo at this site. As with the previous combination, these settings are used for sites that perform external lookups.
tape=ADVANCE index=ON record=OFF	This combination is used when a site does not do anything of significance to cargo. You would use this setting if a site prompts to choose an option from a list and there is a default, or to respond to a yes/no dialog and you want to ensure the data collected at the site is reset.
tape=ADVANCE namedIndex=LOGIN	This combination is used when you want the application to be able to return to a specific index, even if the backup begins in a child tour.
tape=DISCARD	This combination is used when you want the application flow only to go forward from this site. For example, after a user tenders a credit card for a sale, the user cannot backup to enter, delete or modify items. This setting does not permit you to backup or restore cargo to a previously recorded site.

Table 12–6 describes the backup TourCam settings.

Table 12–6 Backup Tour Cam Settings

Settings	Behavior
destination=BACKUP tape=REWIND	This combination returns the application to the previously marked site and makes the snapshot available for undo. This is the preferred method of performing a full backup with restore.
destination=site tape=REWIND	This combination backs up the application until it reaches the specified site. It is only used if the site to which you want to backup does not directly precede the current site or you know that you always want to backup to the specified site. These settings could produce unpredictable results if new sites are later inserted in the map between the current site and the target backup site.
destination=LastIndexed tape=SPLICE	This combination returns the application to the previously marked site without restoring the cargo. These settings are used in scenarios when the cargo is inconsequential.
destination=site tape=SPLICE	This combination backs up the application to the specified site without restoring the cargo. It is used when the cargo is inconsequential, or when you want to loop back to a base site in a tour without permitting backup or undoing cargo after returning to the base site. For example, the application starts from a menu and permits the user to back up until a series of steps are complete, but not afterward. In this case, the final road from the last site returns to the menu. The need to use this combination might indicate a design flaw in the tour. The developer should question whether the series of sites that branch from the menu should be a separate tour. If the answer is no, this combination is the solution.
destination=NamedIndex namedIndex=LOGIN	This combination backs up the application to the origin of the road with the specified named index. This is used to back up to a specific index, even if it was set in a parent tour.

Letter Processing

In the absence of TourCam, processing of letters is straightforward. If the letter triggers a lane, the bus simply traverses the lane. With TourCam enabled, the processing of letters must consider the actions required to retrace the path of the bus. If the letter triggers an aisle, the bus traverses the aisle. There is no backup over an aisle. If the letter triggers a road, tape=advance or tape=discard indicate a forward direction, and tape=rewind or tape=splice indicate a backward direction. The destination of the road element is used to indicate the backup destination when tape=rewind or tape=splice. It can be one of the following values: “LastIndexed”, “NamedIndex”, or <sitename>.

Cargo Restoration

One of the primary strengths of TourCam is the ability to restore the bus’ cargo to a previous state. TourCamIfc provides a mechanism for the bus driver to make and subsequently restore a copy of the cargo when specified by road attributes. Classes that implement TourCamIfc must implement the makeSnapshot() and restoreSnapshot() methods. An example of this is `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\inquiry\giftreceipt\GiftReceiptCargo.java`.

Example 12–23 GiftReceiptCargo.java: TourCamIfc Implementation

```
public class GiftReceiptCargo implements CargoIfc, TourCamIfc
{
...body of GiftReceiptCargo class...
public SnapshotIfc makeSnapshot()
```

```
    {
        return new TourCamSnapshot(this);
    }
public void restoreSnapshot(SnapshotIfc snapshot) throws ObjectRestoreException
{
GiftReceiptCargo savedCargo = (GiftReceiptCargo) snapshot.restoreObject();
this.setPriceCode(savedCargo.getPriceCode());
this.setPrice(savedCargo.getPrice());
    }
}
```

SnapshotIfc provides a mechanism to create a copy of the cargo. The class that implements SnapshotIfc is responsible for storing information about the cargo and restoring it later, by calling restoreObject().

A shuttle allows the optional transfer of cargo from the calling tour to the nested tour during backups. If defined, this shuttle is used during rewind and splice backup procedures. The classname for the shuttle is specified in the tour script using the backupshuttle attribute of the station element.

Example 12–24 Sample Backupshuttle Definition

```
<STATION servicename="foo.xml"
         launchshuttle="MyLaunchShuttle"
         backupshuttle="MyBackupShuttle"\/>
```

Tender Tour Reference

The files in the Tender package can be found in `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\tender`.

Table 12–7 describes resources in the Tender package that are common to all tours.

Table 12–7 Tender Package Components

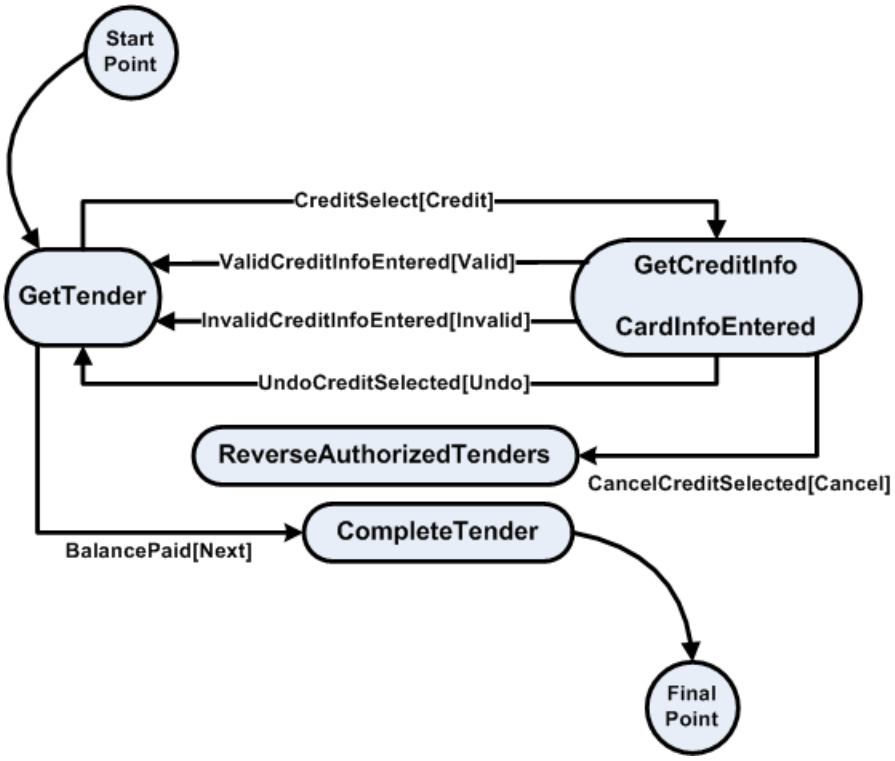
Resource	Filename	Description
Tour script	tender.xml	This file defines the components (sites, letters, roads, and so on) of the Tender tour and the map of the Tender tour.
Tour screens	tenderuicfg.xml	This configuration file contains bean specifications and overlay screen specifications for the Tender tour.
Starting site	GetTenderSite.java	Tender types are displayed from this site. If the selected tender requires input, it is entered using another site, which then returns control to this site. When the balance due is paid, control is returned to the calling service.
Cargo	TenderCargo.java	This class represents the cargo for the Tender tour.

Table 12–7 (Cont.) Tender Package Components

Resource	Filename	Description
Stations	Names (stations do not have classes): AuthorizationStation PINPadStation AddCustomer AddBusinessCustomer FindCustomer SecurityOverrideStation LinkCustomerStation	These stations provide access to other tours. Each of these stations define one or more shuttle classes which are part of the Tender package. The workflows are defined in other packages, but can be called from the Tender tour. For example, AuthFailedRoad is defined in the Tender tour because it handles the exit letter from the Authorization tour. However, Authorization.xml, the workflow for the Authorization tour, is located in `<source_directory>\applications\pos\src\oracle\retail\stores\pos\services\tender\authorization`.

The Tender package is unique in that the workflow is generally similar for all the tender type options available from the main site. For example, if the user chooses to pay by check or credit card, the workflow is similar. When the user cancels the form of payment, the Oracle Retail Platform engine is directed to the ReverseAuthorizedTenders site. When the user decides to undo the operation, the engine is directed back to the GetTender site. The workflow for the credit card tender option is shown in Figure 12–1.

Figure 12–1 Workflow Example: Tender with Credit Card Option



13

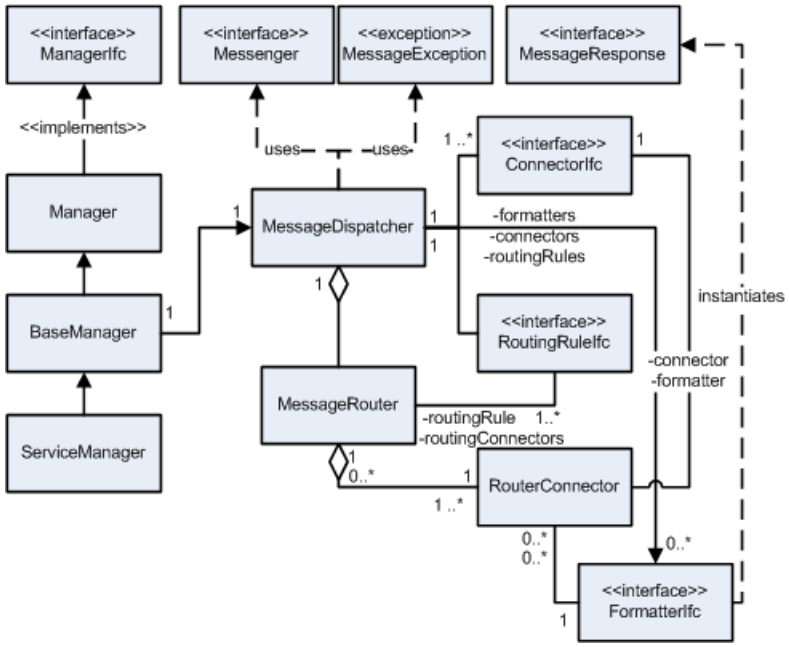
Point-of-Service COMMEXT Framework

The Communication Extension framework (COMMEXT) provides a useful structure for separation of concerns between connectivity to a protocol and the formatting of domain object to message content.

Point-of-Service Connector Framework

The component model shown in Figure 13–1 depicts the Point-of-Service Communication Extension Framework. Data structure manipulation or transformation, and handling connectivity to a service, are divided between the Formatter and the Connector. Each of the components shown in the following model are described in more detail in this section.

Figure 13–1 COMMEXT Overview



BaseManager/BaseTechnician

This is the integration point between the Point-of-Service tier and the communication framework. This abstract class creates and configures the MessageDispatcher based on the configuration script. Access to the MessageDispatcher is through the sendMessage method. A BaseTechnician class provides integration with the Point-of-Service tier for

utilizing the framework within a technician role. This class was omitted from Figure 13–1 for clarity.

ServiceManager/ServiceTechnician

Extension of the BaseManager class provides the application-level API for accessing the service. The ServiceManager prepares messages and performs post-response processing of responses from the external service. The ServiceTechnician is not shown in Figure 13–1, but provides the application level APIs in a similar manner to the ServiceManager class.

MessageDispatcher

MessageDispatcher is the core of the communication framework. Its primary function is to dispatch messages to mapped routers. Also, MessageDispatcher performs administrative and control operations on the associated connectors.

MessageRouter

MessageRouter coordinates the processing of a message using the associated routing rule and the RouterConnectors. The MessageRouter attempts to send the message to a RouterConnector. The results of the attempt are sent to the routing rule and a control action is returned to the MessageRouter. The MessageRouter responds to the control action and can exit as a completed request, throw an exception, retry the current RouterConnector, or try an alternate RouterConnector to process the request.

RouterConnector

RouterConnector provides an association between a message type, connector, and formatter. This decouples the formatting of the message from the chosen connector.

ConnectorIfc

ConnectorIfc handles the communication between the application and the external service. It is responsible for locating the service, establishing a connection, and interacting with the service using appropriate protocols.

FormatterIfc

FormatterIfc translates the raw data from the message into the format expected by the external service. It also translates the response from the remote service into the format expected by the application.

RoutingRuleIfc

RoutingRuleIfc determines the action to be taken by the MessageRouter after sending a message to a connector. The available actions are continue processing, retry the current formatter/connector, successfully completed and return to caller, or throw an exception.

MessageIfc

MessageIfc defines the interface for a container with the message type and the raw data for the message.

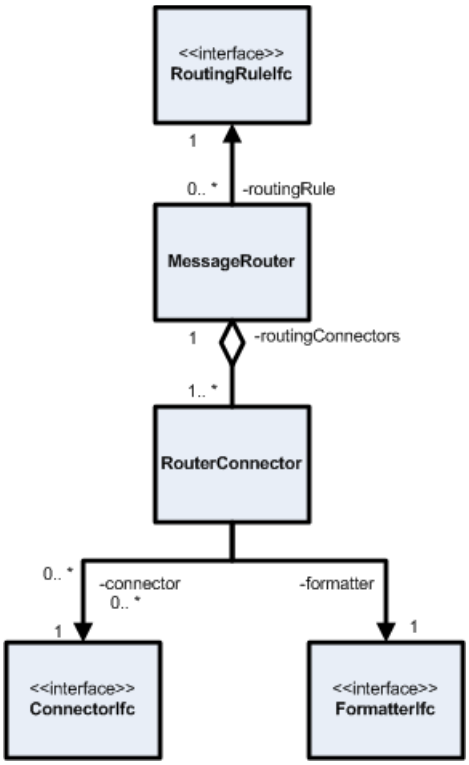
MessageResponseIfc

MessageResponseIfc defines an interface for a container with the translated response to the request.

Message Routing

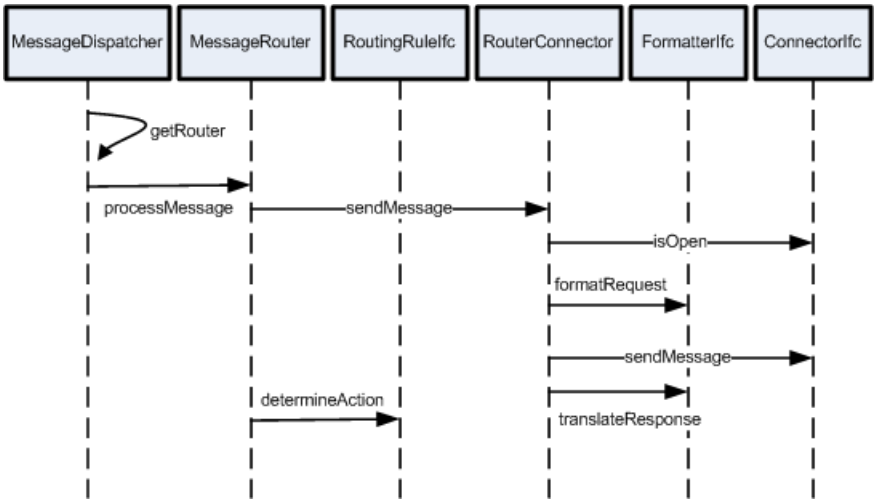
The core functionality of the communication framework is provided by the MessageDispatcher and MessageRouter. The BaseManager delegates the message call to the MessageDispatcher which in turn delegates the message handling to a specific MessageRouter. See Figure 13–2 for a view of the MessageRouter and related components.

Figure 13–2 Message Routing Details



The MessageRouter has a collection of potential routes to send the message. Having multiple routings available provides options for distributing the message to multiple connectors or alternative sources in case of a failure. Specification of the mapping between message types with routing rules, formatters, and connectors is done with a configuration file supported by the BaseManager class. Figure 13–3 shows the sequence diagram for how messages are processed within the MessageRouter.

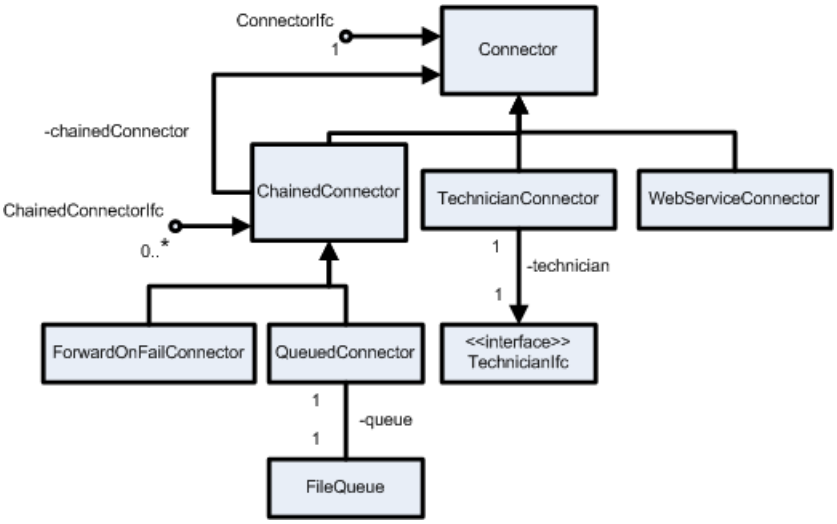
Figure 13–3 Message Routing Sequence



Connectors

The Connector is a base class implementation of the ConnectorIfc. It is recommended that new ConnectorIfc implementations extend Connector or a subclass of Connector. The Connector class provides basic JMX instrumentation for connector operations and coordinates the administrative operations. The extending class must provide an implementation of three abstract protected methods.

Figure 13–4 Connector Hierarchy Example



The Chained Connectors provide opportunities to link connectors head-to-tail. The framework uses this structure to provide store and forward operations implemented in QueuedConnector. Other possible uses might be for encryption/decryption or statistics collection.

The QueuedConnector class provides an implementation of ChainedConnectorIfc utilizing a local file to queue requests. The requests are forwarded to the associated ConnectorIfc chainedConnector and removed from the queue if the sendMessage() on the downstream connector is successful.

The ForwardOnFailConnector is a component to provide request/response operations if successful, with the option to forward the message to an alternate routing on failure.

The TechnicianConnector provides a connector implementation to communicate with Point-of-Service technicians. The message data must be translated by the formatter into an appropriate ValetIfc implementation for use with the associated technician.

The WebServiceConnector represents connector implementations for interacting with web services exposed by target systems.

COMMEXT Patterns to Support Interaction Behavior

The following are COMMEXT patterns supporting interaction behavior.

Store and Forward

Figure 13–5 shows the COMMEXT pattern for providing Store and Forward message processing.

Figure 13–5 Store and Forward Operations in COMMEXT

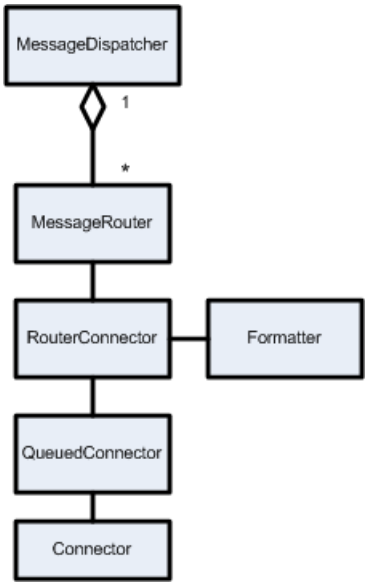
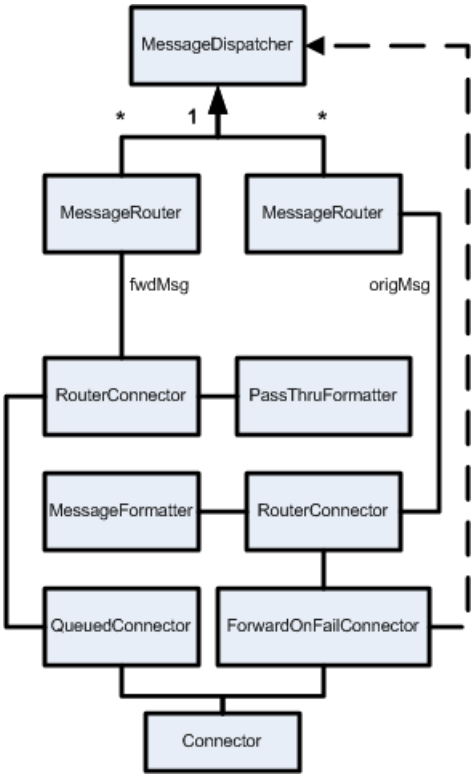


Figure 13–5 depicts how the framework is configured to provide Store and Forward message processing. Store and Forward message handling uses a QueuedConnector instance, which is a subtype of a ChainedConnector to provide the buffering for the message. The Client sends a message which is delegated to the MessageRouter by the MessageDispatcher. The MessageRouter uses the RouterConnector to link a formatter and connector. The Formatter class shown is an implementation of the FormatterIfc. After formatting the message, the message is sent to the QueuedConnector where it is persisted and the submittal of the message from the client ends. A background thread in the QueuedConnector checks to see if messages are present in the queue. If a message is available, the message is read from the head of the queue and passed to the chained connector. If the message is successfully handled by the associated connector, the message is removed from the head of the queue. If the message is not successfully processed by the downstream connector, the message is left on the queue for a later retry.

Attempt, Store and Forward on Failure

Figure 13–6 depicts how the framework is configured to provide a request/response operation on successful processing, with store and forward handling of the message when immediate process is not successful. The client will be aware that the message has failed on the immediate attempt.

Figure 13–6 Attempt, Store and Forward on Failure in COMMEXT



The client sends a message which the MessageDispatcher directs to the appropriate MessageRouter. The MessageRouter delegates the original message to the RouterConnector to apply the necessary formatting and connector, in this case the ForwardOnFailConnector. The ForwardOnFailConnector first sends the message to the Connector. If the message processing succeeds, the process is complete and control returns to the client. If the message processing in the Connector fails, the ForwardOnFailConnector catches the exception, repackages the payload into the a new message, fwdMsg, and uses the message forwarding capabilities of the COMMEXT framework to send the new message to the MessageDispatcher. Handling of the fwdMsg is the same as the Store and Forward pattern described in the previous section.

14

Oracle Retail Returns Management Extensibility Framework

The purpose of the extensibility framework is to simplify the creation and deployment of new rules and calculators. The framework uses Apache Ant to automate library extraction, code compilation, and packaging of the extensions.

The framework is packaged in the `returnsExtensibility.zip` file that is contained within the EPD application archive under `returnsmgmt/api`. To use the framework simply unzip the archive to a working directory. In most cases, you can simply update the input and output directory properties in the `extensibility.properties` file. If you want to compile and package the samples that are provided in the framework, all you need to do is run the `build` target.

Adding a New Rule

Rules get executed during the evaluateReturnRequest() method call. This section demonstrates how to create a rule that is based off of the item's technical check condition. This section also demonstrates how to make use of the MessageExtension elements.

First, make a new Evaluator class. For ease of implementation, subclass the DiscreteRuleEvaluator class. This enables you to reuse the getMatchingAction() method, which looks through the rule actions for the discrete value desired. This class figures out that discrete value.

There are only two methods to be implemented:

- evaluate()
- getDiscreteRuleValues()

The evaluate() method is the method that provides the meat of the Evaluator implementation. The getDiscreteRuleValues() method returns a list of string values. These strings constitute the list of valid values for this particular class. Start with a skeleton class:

```
package com.yourcompany.returns.rules;

import oracle.retail.stores.commerceservices.returns.ejb.ReturnServiceRemote;
import
oracle.retail.stores.commerceservices.returns.journal.ItemAuthorizationJournalEntr
y;
import oracle.retail.stores.commerceservices.returns.policy.PolicyRuleDTO;
import oracle.retail.stores.commerceservices.returns.rule.RuleActionIfc;
import oracle.retail.stores.commerceservices.returns.rule.RuleProcessingException;
import oracle.retail.stores.commerceservices.returns.xml.ReturnRequestType;
```

```
import java.util.List;

/**
 * A discrete rule evaluator that checks the value of the
 * techCheckCondition extensible attribute.
 */
public class TechCheckConditionEvaluator extends DiscreteRuleEvaluator
{

    public RuleActionIfc evaluate(ReturnServiceRemote returnService,
            PolicyRuleDTO rule, ReturnRequestType returnRequest,
            int itemIndex, Integer rmCustomerID,
            String evaluationBusinessDate,
            ItemAuthorizationJournalEntry journalEntry)
            throws RuleProcessingException
    {
        return null;
    }

    public List getDiscreteRuleValues()
    {
        return null;
    }
}
```

Determine what the technical check condition is. This is the discrete value used to find a rule action.

In this case, parse through the returnRequest object to find the extensible value associated with this item in the request. Here is the new class, with the changes in bold:

```
public class TechCheckConditionEvaluator extends DiscreteRuleEvaluator
{
    public RuleActionIfc evaluate(ReturnServiceRemote returnService,
            PolicyRuleDTO rule, ReturnRequestType returnRequest,
            int itemIndex, Integer rmCustomerID,
            String evaluationBusinessDate,
            ItemAuthorizationJournalEntry journalEntry)
            throws RuleProcessingException
    {
        String techCheckReason = null;
        // find the item in the request
        ItemReturnInfo item = (ItemReturnInfo)
            returnRequest.getItemReturnInfo().get(itemIndex);
        // get the extensible attributes for the item
        MessageExtension ext = item.getMessageExtension();
        // find the techCheckCondition attribute
        if (ext != null) {
            for (Iterator it = ext.getExtensionEntry().iterator();
                it.hasNext();)
            {
                ExtensionEntry entry = (ExtensionEntry) it.next();

                if (entry.getName().equals("techCheckCondition")) {
                    techCheckReason = entry.getValue();
                }
            }
        }
        // return the action for this value
```

```
        return super.getMatchingAction(rule.getActions(),
            techCheckReason);
    }

    public List getDiscreteRuleValues() {
        return null;
    }
}
```

Next, ensure that this class records its actions into the journal entry, like other Evaluators do. Add a call to the journal entry. Now the class looks like the following, again with the changed portions in bold:

```
public class TechCheckConditionEvaluator extends DiscreteRuleEvaluator
{
    public RuleActionIfc evaluate(ReturnServiceRemote returnService,
            PolicyRuleDTO rule, ReturnRequestType returnRequest,
            int itemIndex, Integer rmCustomerID,
            String evaluationBusinessDate,
            ItemAuthorizationJournalEntry journalEntry)
            throws RuleProcessingException
    {
        String techCheckCondition = null;
        // find the item in the request
        ItemReturnInfo item = (ItemReturnInfo)
            returnRequest.getItemReturnInfo().get(itemIndex);
        // get the extensible attributes for the item
        MessageExtension ext = item.getMessageExtension();
        // find the techCheckCondition attribute
        if (ext != null) {
            for (Iterator it = ext.getExtensionEntry().iterator();
                it.hasNext();)
            {
                ExtensionEntry entry = (ExtensionEntry) it.next();

                if (entry.getName().equals("techCheckCondition")) {
                    techCheckCondition = entry.getValue();
                }
            }
        }
        // log the rule and result value
        journalEntry.addRuleResult(rule.getName(),
                techCheckCondition == null ? "No condition found."
                        : techCheckCondition);
        // return the action for this value
        return super.getMatchingAction(rule.getActions(),
            techCheckCondition);
    }

    public List getDiscreteRuleValues() {
        return null;
    }
}
```

Finally, fill the list of allowed values. This list is used to fill in the menu box in the Returns Management UI, for configuring the allowed values for this rule.

```
public class TechCheckConditionEvaluator extends DiscreteRuleEvaluator
{
    public RuleActionIfc evaluate(ReturnServiceRemote returnService,
            PolicyRuleDTO rule, ReturnRequestType returnRequest,
```

```
            int itemIndex, Integer rmCustomerID,
            String evaluationBusinessDate,
            ItemAuthorizationJournalEntry journalEntry)
            throws RuleProcessingException
    {
        // … lines removed …
    }

    public List getDiscreteRuleValues()
    {
        String[] values = new String[] {
                "All Tests Passed",
                "Partial Test Failure",
                "Complete Test Failure"
        };
        return Arrays.asList(values);
    }
}
```

A static set of strings is not the ideal implementation. The rules that are shipped with Oracle Retail Returns Management rely on parameters to define their valid values and make use of the ParameterDiscreteRuleEvaluator. This solution allows for the values to be changed independently of the class at runtime. However, this is not the only possible implementation. Since the evaluator only needs to implement the getDiscreteRuleValues() method, the implementer has a great deal of latitude in deciding how the list of strings is generated.

Now that the class is written, you need to do two things:

1. Compile and deploy the new class in the application server. Using the extensibility framework simplifies this task.
2. Configure the database so that we can see the new evaluator in action.

For a rule, there is only one database table to configure, the rule table (RM_RU). Table 14–1 identifies the columns in the RM_RU table.

Table 14–1 RM_RU Columns

Column Name	Data Type	Description	Notes
ID_RU	Integer	Primary Key	Must be unique.
RU_TY_EVAL	VARCHAR	Type of Rule	One of: ■ BOOLEAN ■ DISCRETE ■ RANGE
NM_RU	VARCHAR	Display Name	NA
LU_RU_CLS	VARCHAR	Class Name	Fully qualified classname.
ID_KPI	Integer	KPI Reference	ID of associated KPI. Otherwise null.

Create a new discrete rule for the class. It does not have a KPI reference. The following SQL creates this new discrete rule:

```
INSERT INTO rm_ru
  (id_ru, ru_ty_eval, nm_ru, lu_ru_cls, id_kpi, fl_pnty_bx)
VALUES
  (100, 'DISCRETE', 'What is the item technical check condition?',
```

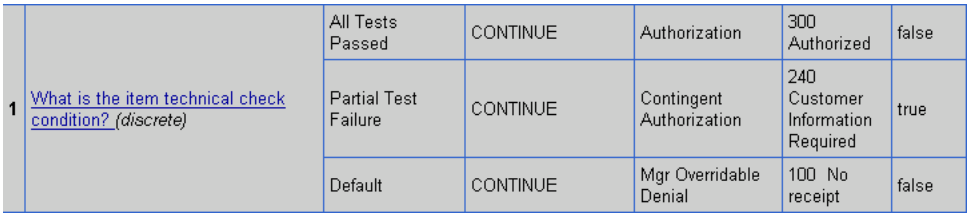
```
'com.yourcompany.returns.rules.TechCheckConditionEvaluator', null, '0');
```

Ensure that the rule ID is unique, and that the class name is both fully qualified and correct.

Once you have updated the database, navigate to the desired policy in the Returns Management UI. Then click the **change rules/order** button next to **Policy Rules**. You can see the new rule at the bottom of the list.

Once you click the **Done** button, you can click on the rule name and add actions and response codes for the various conditions you want to configure.

Figure 14–1 Example Rule Configuration Screen



1	What is the item technical check condition? *(discrete)*	All Tests Passed	CONTINUE	Authorization	300 Authorized	false
		Partial Test Failure	CONTINUE	Contingent Authorization	240 Customer Information Required	true
		Default	CONTINUE	Mgr Overridable Denial	100 No receipt	false

Be sure to save the policy after configuring the rule.

The following is the complete source for the TechCheckConditionEvaluator:

```
package com.yourcompany.returns.rules;

import oracle.retail.stores.commerceservices.returns.ejb.ReturnServiceRemote;
import
oracle.retail.stores.commerceservices.returns.journal.ItemAuthorizationJournalEntr
y;
import oracle.retail.stores.commerceservices.returns.policy.PolicyRuleDTO;
import oracle.retail.stores.commerceservices.returns.rule.RuleActionIfc;
import oracle.retail.stores.commerceservices.returns.rule.RuleProcessingException;
import
oracle.retail.stores.commerceservices.returns.rule.evaluator.DiscreteRuleEvaluator
;
import oracle.retail.stores.commerceservices.returns.xml.ReturnRequestType;
import oracle.retail.stores.commerceservices.returns.xml.ItemReturnInfo;
import oracle.retail.stores.commerceservices.returns.xml.MessageExtension;
import oracle.retail.stores.commerceservices.returns.xml.ExtensionEntry;

import java.util.List;
import java.util.Iterator;
import java.util.Arrays;
/**
 * A discrete rule evaluator that checks the value of the
 * techCheckCondition extensible attribute.
 */
public class TechCheckConditionEvaluator extends DiscreteRuleEvaluator
{

    public RuleActionIfc evaluate(ReturnServiceRemote returnService,
            PolicyRuleDTO rule, ReturnRequestType returnRequest,
            int itemIndex, Integer rmCustomerID,
            String evaluationBusinessDate,
            ItemAuthorizationJournalEntry journalEntry)
            throws RuleProcessingException
    {
        String techCheckCondition = null;
```

```
            // find the item in the request
            ItemReturnInfo item = (ItemReturnInfo)
                returnRequest.getItemReturnInfo().get(itemIndex);
            // get the extensible attributes for the item
            MessageExtension ext = item.getMessageExtension();
            // find the techCheckCondition attribute
            if (ext != null) {
                for (Iterator it = ext.getExtensionEntry().iterator();
                    it.hasNext();)
                {
                    ExtensionEntry entry = (ExtensionEntry) it.next();

                    if (entry.getName().equals("techCheckCondition")) {
                        techCheckCondition = entry.getValue();
                    }
                }
            }
            // log the rule and result value
            journalEntry.addRuleResult(rule.getName(),
                    techCheckCondition == null ? "No condition found."
                            : techCheckCondition);
            // return the action for this value
            return super.getMatchingAction(rule.getActions(),
                techCheckCondition);
        }

        public List getDiscreteRuleValues()
        {
            String[] values = new String[] {
                    "All Tests Passed",
                    "Partial Test Failure",
                    "Complete Test Failure"
            };
            return Arrays.asList(values);
        }
    }
```

Adding a New KPI Calculator

Now that you have added a new rule evaluator, explore a new KPI Calculator. One of the things that a KPI can do is add to the cumulative exception count. Exceptions are triggered during the processFinalResult() method. Consider a trivial exception that simply counts up the number of times a certain item has been returned. Each time this item is returned, a row is added into the exception count.

To create a new KPI, do the following:

1. Create the new class.
2. Configure the database.
3. Create some JSP fragments to manipulate the parameters for the new KPI.

The Calculator Class

First of all, you need to make a new instance of the KPICalculatorIfc interface, or more specifically a new subclass of the class BaseKPICalculator.

Because you are using the abstract class, there are three methods to implement:

- initialize() – Called on each KPI Calculator to set up an initial state. In practice, this usually means parsing KPI instance parameters or getting a reference to the KPIService from the EJB Handle passed in. The abstract base class has a method to achieve the latter called initializeServiceFromHandle().
- hasMatchingBehavior() – This method enables the KPIValueDTO to indicate, ex-post-facto, if the KPI was fired for a particular return ticket. Does this return match the behavior that I'm looking for? This method must not rely on any values computed by the calculate method because that method might not be called depending on the calculator type.
- calculate() – This is the workhorse method of the class. Calculate is sent in a set of facts, using the Map object, with which it can make a decision or use to perform some kind of count of historical data. This decision or count is then returned as a BigDecimal.

For this simple class, first make a skeletal stub class:

```
package com.yourcompany.returns.kpis;

import oracle.retail.stores.commerceservices.returns.kpi.CalculatorException;
import oracle.retail.stores.commerceservices.returns.kpi.KPIInstanceIfc;
import oracle.retail.stores.commerceservices.returns.kpi.impl.BaseKPICalculator;
import oracle.retail.stores.commerceservices.returns.ticket.ReturnTicketDTO;

import javax.ejb.Handle;
import java.math.BigDecimal;
import java.util.Map;

/**
 * A KPI Calculator used for exception tracking purposes.
 */
public class SpecificItemCalculator extends BaseKPICalculator
{

    public void initialize(Handle handle, KPIInstanceIfc kpiInstance)
            throws CalculatorException
    {
    }

    public BigDecimal calculate(Map params)
    {
        return null;
    }

    public boolean hasMatchingBehavior(ReturnTicketDTO returnTicket)
            throws CalculatorException
    {
        return false;
    }
}
```

Now we need to fill in some blanks. Tackle the calculate() method first. In most cases the calculator accesses the database to answer this question. For this example KPI that only counts exceptions, this method returns a default zero value.

The map passed in contains the return information. It is somewhat awkward to withdraw data from the non-type-safe map. The map is populated with default values depending on the context in which the KPI is called. Generally, two values are populated:

```
mapIDs.put(KPIParameterIfc.CUSTOMER_ID, rmCustomerID);
mapIDs.put(KPIParameterIfc.RETURN_TICKET, returnTicket);
```

However, when the KPI is called from the KPIRangeEvaluator or the KPICashierRangeEvaluator, then only the customer or cashier, respectively, is populated.

Each KPI also might have *instance* parameters that correspond to this particular KPI. In this case, the instance parameters can be withdrawn during the initialization phase. The parameters are part of the kpiInstance class and can be accessed by the method getInstanceParameters(). This method returns a TreeSet rather than a map, though order is not important here. The set contains a group of KPIInstanceParameterDTO classes. Using the getName() method, a KPI can determine which of these parameters satisfy which criteria.

In this case, determine if a certain item is in a return ticket. First, pull the item from the KPI instance parameters. To do this, define a new constant for the parameter name and get it out of the parameters passed in to the initialize function.

```
protected String itemID = null;
public static final String PARAM_ITEM_ID = "itemID";

public void initialize(Handle handle, KPIInstanceIfc kpiInstance)
        throws CalculatorException
{
    if (! isInitialized()) {
        TreeSet kpiParams = kpiInstance.getInstanceParameters();
        Iterator iter = kpiParams.iterator();
        while (iter.hasNext())
        {
            KPIInstanceParameterDTO param =
                    (KPIInstanceParameterDTO) iter.next();
            String paramName = param.getName();
            if (paramName.equals(PARAM_ITEM_ID))
            {
                itemID = param.getInstanceValue();
                break;
            }
        }

        setInitialized(true);
    } else {
        throw new IllegalStateException("Initializing a " +
            "calculator which is already initialized");
    }
}
```

Notice that two variables are added: a constant to indicate the parameter (used later to update the SQL, and in the JSP) and an instance variable to hold the value being sought.

Add in the simplified calculate() method. In this case, return a default value of zero.

```
public BigDecimal calculate(Map params)
{
    return BigDecimalConstants.ZERO;
}
```

Because we are only interested in exception counting, the significant part of the code goes in the hasMatchingBehavior() method.

```
    public boolean hasMatchingBehavior(ReturnTicketDTO returnTicket)
            throws CalculatorException
    {
        for (Iterator it = returnTicket.getReturnTicketItems()
                .iterator(); it.hasNext();)
        {
            ReturnTicketItemDTO item = (ReturnTicketItemDTO) it.next();
            if (item.getItemID().equals(itemID))
            {
                return true;
            }
        }
        return false;
    }
```

Now the class is ready. Compile and deploy the new class.

The following is the complete source for the SimpleItemCalculator:

```
package com.yourcompany.returns.kpis;

import oracle.retail.stores.commerceservices.returns.kpi.CalculatorException;
import oracle.retail.stores.commerceservices.returns.kpi.KPIInstanceIfc;
import oracle.retail.stores.commerceservices.returns.kpi.KPIInstanceParameterDTO;
import oracle.retail.stores.commerceservices.returns.kpi.impl.BaseKPICalculator;
import oracle.retail.stores.commerceservices.returns.ticket.ReturnTicketDTO;
import oracle.retail.stores.commerceservices.returns.ticket.ReturnTicketItemDTO;
import oracle.retail.stores.common.utility.BigDecimalConstants;

import javax.ejb.Handle;
import java.math.BigDecimal;
import java.util.Map;
import java.util.TreeSet;
import java.util.Iterator;
/**
 * A KPI Calculator used for exception tracking purposes.
 */
public class SpecificItemCalculator extends BaseKPICalculator
{
    protected String itemID = null;
    public static final String PARAM_ITEM_ID = "itemID";

    public void initialize(Handle handle, KPIInstanceIfc kpiInstance)
            throws CalculatorException {
        if (! isInitialized()) {
            TreeSet kpiParams = kpiInstance.getInstanceParameters();
            Iterator iter = kpiParams.iterator();
            while (iter.hasNext())
            {
                KPIInstanceParameterDTO param =
                        (KPIInstanceParameterDTO) iter.next();
                String paramName = param.getName();
                if (paramName.equals(PARAM_ITEM_ID))
                {
                    itemID = param.getInstanceValue();
                    break;
                }
            }

            setInitialized(true);
        } else {
```

```
            throw new IllegalStateException("Initializing a " +
                "calculator which is already initialized");
        }
    }

    public BigDecimal calculate(Map params)
    {
        // exception counting only KPI
        return BigDecimalConstants.ZERO;
    }

    public boolean hasMatchingBehavior(ReturnTicketDTO returnTicket)
            throws CalculatorException
    {
        for (Iterator it = returnTicket.getReturnTicketItems()
                .iterator(); it.hasNext();)
        {
            ReturnTicketItemDTO item = (ReturnTicketItemDTO) it.next();
            if (item.getItemID().equals(itemID))
            {
                return true;
            }
        }
        return false;
    }
}
```

Database Configuration

The next step is to update the database. In this case, update two tables. These tables are the KPI table (RM_KPI) and the KPI parameter table (RM_KPI_PRMR).

Table 14–2 identifies the six columns to update in the KPI table.

Table 14–2 RM_KPI Columns

Column Name	Data Type	Description	Notes
ID_KPI	Integer	Primary Key	Must be unique
NM_KPI	VARCHAR	Logical Name	Displayed in exception lists
DE_DISP_NM	VARCHAR	Display Name	Name displayed in the UI during user configuration
NM_KPI_CLS	VARCHAR	Class Name	Fully qualified name
CAT_KPI	Integer	KPI Category	One of: ■ Customer ■ Cashier ■ Store ■ Item ■ A combination of the above
TY_KPI	Integer	KPI Type	NA

The KPI type column is an integer value in the database. The value of this column, however, is interpreted as a series of bit flags. That is, a value of **7** in one of these fields means that the first three flags are set while the others are all blank (for example, DEC 7 = BIN 0111).

Table 14–3 identifies the three possible KPI type values.

Table 14–3 KPI Type Flags

Flag Value	Flag Type
1	Rule Evaluation
4	Cumulative Exception Count
8	Alert

Types determine when a KPI is evaluated. A rule needs to have the rule evaluation type flag set.

Categories affect when a KPI is executed during the scoring phase, if it's executed at all. Customer and Cashier KPIs, for example, are executed in two different routines.

Note that the numeric values for both the categories and the types are defined in the interface KPIIfc.

For this class, create a new customer KPI that is used during cumulative exception counting. The SQL to do this looks like the following:

```
INSERT INTO rm_kpi
  (id_kpi, nm_kpi, de_disp_nm, nm_kpi_cls, cat_kpi, ty_kpi)
VALUES
  (100, 'Specific Item KPI', 'When a specific item is returned',
'oracle.retail.stores.commerceservices.returns.kpi.impl.SpecificItemCalculator',
1, 4);
```

Make sure that the ID is unique and that the class name is both fully qualified and correct.

The next table to update is the RM_KPI_PRMR table. In this case, add in a parameter for the specific item being searched for.

Table 14–4 identifies the columns in the RM_KPI_PRMR table.

Table 14–4 RM_KPI_PRMR Columns

Column Name	Data Type	Description	Notes
ID_KPI_PRMR	Integer	Primary Key	Must be unique
ID_KPI	Integer	Foreign Key	Non-null
NM_PRMR	VARCHAR	Parameter Name	Used to identify parameters from the getInstanceParameters() method
TY_PRMR_VAL	Integer	Data Type	▪ Integer ▪ String ▪ Boolean ▪ Date ▪ List
DE_DFLT_VAL	VARCHAR	Default Value	NA
FL_CFG_PRMR	CHAR	Configurability Flag	▪ 0 – configurable ▪ 1 – not configurable
TY_PRMR	Integer	Usage Type	Bit flag with same scheme as TY_KPI

Most of these are obvious. One important note is that the NM_PRMR value here is what is used in the KPI method to extract the parameter. So the name in the database must match the name that you have coded. In this case, that name is **itemID**.

The last two values also bear discussion. The configurable flag, though set in the database, has no effect on where the KPI appears in the exceptions to track screen. The UI determines if a return activity KPI is configurable by counting the number of parameters associated with it. If there are one or more, then the KPI is configurable. If zero, then the KPI is not configurable. If the KPI is used for rules corresponding to a rule, this flag controls if the parameter should be displayed on the Edit Return Policy Rule page. If the parameter is not already in use by other KPIs, additional customization in ruleKPICfg.jsp might be necessary to have the parameter appear in the rule editor.

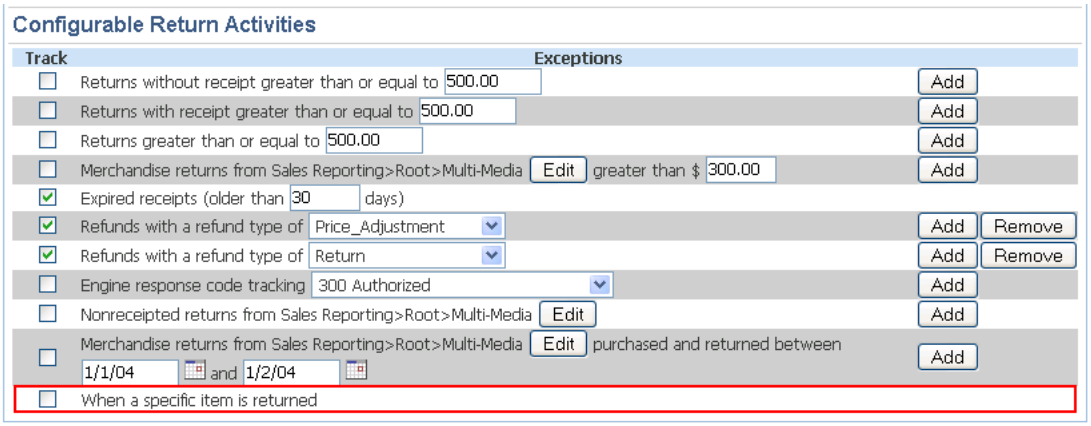
Finally, the TY_PRMR method is another bit field. The idea is that the same KPI can use different parameters in different situations. This field allows the developer to set at which times which parameters will be used.

The following SQL creates an exception tracking only KPI:

```
INSERT INTO rm_kpi_prmr
  (id_kpi_prmr, id_kpi,  nm_prmr, ty_prmr_val, de_dflt_val, fl_cfg_prmr, ty_prmr)
VALUES
  (1000, 100, 'itemID', 2, '1234', '1', 4);
```

Now start the app server and see the new KPI in action by going to **Returns --> Configuration --> Exceptions to Track --> Customer**.

Figure 14–2 Example Customer KPI Screen



Creating the JSP

To display something here, the application does one of the following:

- The JSP tag class BaseKPITag tries to find a JSP along the path of `/returns/kpi/classname.jsp`, where classname is the non-qualified name of the class. In this case, the JSP is named SpecificItemCalculator.jsp.
- If the JSP tag class BaseKPITag cannot find a file of this name it then checks for `/returns/kpi/extensions/classname.jsp`.
- If the JSP tag class BaseKPITag cannot find the JSP there, the JSP tag class BaseKPITag defaults to using the display name value from the KPI and displays it using the JSP `/returns/kpi/displayName.jsp`. That is what is occurring here.

To remedy this, create and deploy a new SpecificItemCalculator.jsp. When creating this JSP, consider the following:

- The JSP is responsible for formatting the remaining table cells for the row.
- The JSP itself is responsible for enabling and disabling the KPI.
- The JSP needs to create an **Add** button if necessary.
- The JSP needs to create a **Remove** button if necessary.

> **Note:** **Add** and **Remove** buttons are used only in the case of cloneable KPIs.

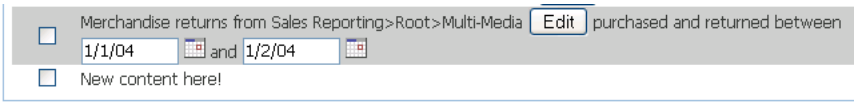
- The JSP is required to provide role-based security.

First, make a simple skeletal file. The first thing to know is that the enclosing JSP expects the JSP in the configurable section to contain three table cells (for example, <TD> elements). Create a new file that looks like the following:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>

<td width="83%" height="20" class="normal">
New content here!
 
</td>

<td width="5%" height="20" class="normal">
<!-- place holder for an add button -->
 
</td>

<td width="5%" height="20" class="normal">
<!-- place holder for a delete button-->
 
</td>
```

Save this as `/returns/kpi/extensions/SpecificItemCalculator.jsp`, redeploy, and now see something a little better looking:

Figure 14–3 Example Customer KPI Screen, continued



Now that the page is ready, add in some editable value to it. In this case, that value is the item ID to track.

The JSP makes use of the nested functionality of the Struts framework. By the time you are in this JSP, you are already nested inside the specific parameter. By continuing to use this idiom, the parameter updates painlessly.

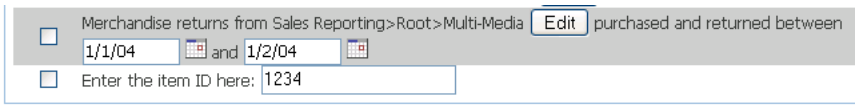
The JSP now looks like the following, with changes in bold:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>

<%@ page
```

```
import="oracle.retail.stores.commerceservices.returns.kpi.impl.SpecificItemCalcula
tor" %>

<td width="83%" height="20" class="normal">
    <nested:iterate property="allParams"
                    id="viewKpiParam"

type="oracle.retail.stores.webmodules.returns.app.kpi.ViewKpiParameterBean">
        <nested:nest property="kpiParam">
            <nested:equal property="name"
                          value="<%= SpecificItemCalculator.PARAM_ITEM_ID %>">
                <!-- include I18N message -->
                Enter the item ID here:
                <!-- include enabled stuff -->
                <nested:text property="value"
                             value="<%=viewKpiParam.getValue()%>"/>
            </nested:equal>
        </nested:nest>
    </nested:iterate>
</td>
<td width="5%" height="20" class="normal">
<!-- place holder for an add button -->
 
</td>

<td width="5%" height="20" class="normal">
<!-- place holder for a delete button-->
 
</td>
```

Notice the default item ID is displayed here in an editable text box. Notice that you can update the item ID and it now gets saved into the database.

Figure 14–4 Example Customer KPI Screen, continued



The code is updating properly due to the use of the nested tags. The section for the KPI looks something like this:

```
<input type="text"
  name="kpiCfgColl[6].kpiBean2.allParams[0].kpiParam.value"
  value="1234">
```

The long name, `kpiCfgColl[6].kpiBean2.allParams[0].kpiParam.value`, is used by Struts on the server side. It lets Struts determine which KPI bean is being talked about and update the bean accordingly.

Now that the KPI is displayed in a reasonable fashion, address security. Returns Management has a weak notion of security at the UI level. In this case, only check to see if the current user has the role necessary to modify KPIs. Check if this is a customer or cashier KPI and set the Boolean flag accordingly. Later, use this to enable or disable the input.

The following is what the JSP now looks like with the security code, with changes in bold:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
```

```
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>

<%@ page import="oracle.retail.stores.commerceservices.returns.kpi.KPIIfc,

oracle.retail.stores.commerceservices.returns.kpi.impl.SpecificItemCalculator,
                 oracle.retail.stores.webmodules.returns.ui.ReturnsConstantIfc,
                 oracle.retail.stores.webmodules.returns.ui.kpi.EditKpiForm"
        %>
<%
    EditKpiForm form = (EditKpiForm)
request.getSession().getAttribute(ReturnsConstantIfc.EDIT_KPI_TAG_FORM);
    boolean disabledIfNoPrivilege =
!(request.isUserInRole(ReturnsConstantIfc.EDIT_KPI_CUSTOMER_PRIVILEGE));
    if (form.getCategory() == KPIIfc.CATEGORY_CASHIER)
    {
        disabledIfNoPrivilege = !(request.isUserInRole(ReturnsConstantIfc.EDIT_
KPI_CASHIER_PRIVILEGE));
    }
%>

<td width="83%" height="20" class="normal">
    <nested:iterate property="allParams"
                    id="viewKpiParam"

type="oracle.retail.stores.webmodules.returns.app.kpi.ViewKpiParameterBean">
        <nested:nest property="kpiParam">
            <nested:equal property="name"
                          value="<%= SpecificItemCalculator.PARAM_ITEM_ID %>">
                <!-- include I18N message -->
                Enter the item ID here:
                <nested:text
                        disabled="<%=disabledIfNoPrivilege%>"
                        property="value"
                        value="<%=viewKpiParam.getValue()%>"/>
            </nested:equal>
        </nested:nest>
    </nested:iterate>
</td>

<td width="5%" height="20" class="normal">
<!-- place holder for an add button -->
 
</td>

<td width="5%" height="20" class="normal">
<!-- place holder for a delete button-->
 
</td>
```

Finally, this KPI is a candidate for being cloneable. That is, being able to track returns on multiple item IDs.

In order to add in the cloning functionality, call one of two javascript functions on the page. The first is selAddSubmit(). The second is selDelSubmit(). These two methods submit the page back to the EditKpiAction class with the appropriate values to either create a clone of the selected KPI or to delete the currently selected clone.

The JSP encoding is currently a bit awkward for this functionality. The following is what the page looks like in final form when the cloneable code is added:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
```

```
<%@ taglib uri="/WEB-INF/struts-nested.tld" prefix="nested" %>

<%@ page import="oracle.retail.stores.commerceservices.returns.kpi.KPIIfc,

oracle.retail.stores.commerceservices.returns.kpi.impl.SpecificItemCalculator,
                oracle.retail.stores.webmodules.returns.ui.ReturnsConstantIfc,
                oracle.retail.stores.webmodules.returns.ui.kpi.EditKpiForm"
        %>

<%
    EditKpiForm form = (EditKpiForm)
request.getSession().getAttribute(ReturnsConstantIfc.EDIT_KPI_TAG_FORM);
    boolean disabledIfNoPrivilege =
!(request.isUserInRole(ReturnsConstantIfc.EDIT_KPI_CUSTOMER_PRIVILEGE));
    if (form.getCategory() == KPIIfc.CATEGORY_CASHIER)
    {
        disabledIfNoPrivilege = !(request.isUserInRole(ReturnsConstantIfc.EDIT_
KPI_CASHIER_PRIVILEGE));
    }

    // please be careful with the quotation...

    // This is a funky work around the problem of JspC not translating the
parameter inside of
    // a substitution.
    // It is a one pass, and we really need it to be a two pass like compiler.

    // Because the kpi.jsp page itself could contain more than one kpi with
merchandise hierarchy, and the user
    // could edit any one of those, we need a way to distinguish which kpi
parameter is being edited.
    String endString = new String("');");
    String comma = new String("', '");
    String callingAddCloneFunction = new String("selAddSubmit('");
    String callingDeleteCloneFunction = new String("selDelSubmit('");
%>

<td width="83%" height="20" class="normal">
    <nested:define id="templateKpiId" property="templateId"/>
    <nested:define id="instanceKpiId" property="instanceId"/>

    <nested:iterate property="allParams"
                    id="viewKpiParam"

type="oracle.retail.stores.webmodules.returns.app.kpi.ViewKpiParameterBean">
        <nested:nest property="kpiParam">
            <nested:equal property="name"
                          value="<%= SpecificItemCalculator.PARAM_ITEM_ID %>">
                <!-- include I18N message -->
                Enter the item ID here:
                <nested:text
                        disabled="<%=disabledIfNoPrivilege%>"
                        property="value"
                        value="<%=viewKpiParam.getValue()%>"/>
            </nested:equal>
        </nested:nest>
    </nested:iterate>
     
</td>
```

```
<td width="5%" height="20" class="normal">
    <nested:submit disabled="<%=disabledIfNoPrivilege%>"
                   onclick="<%=callingAddCloneFunction + templateKpiId +
endString%>">
        <nested:message key="button.add"/>
    </nested:submit>
</td>

<!-- allow removal of clone if we have more than one of the same template type -->
<nested:equal property="allowedRemovalClone" value="true">
    <td width="5%" height="20" class="normal">
        <nested:submit disabled="<%=disabledIfNoPrivilege%>"
                       onclick="<%=callingDeleteCloneFunction + templateKpiId +
comma + instanceKpiId + endString%>">
            <nested:message key="button.remove"/>
        </nested:submit>
    </td>
</nested:equal>
<nested:equal property="allowedRemovalClone" value="false">
    <td width="5%" class="normal"> </td>
</nested:equal>
```

And, finally, see the **Add** and **Remove** buttons:

Figure 14–5 Example Customer KPI Screen, continued

15
Retail Domain

This chapter contains an overview of the Oracle Retail business objects, including steps to create, extend, and use them. The Retail Domain is the set of classes that represent the business objects used by Point-of-Service. Typical domain classes are Customer, Transaction, and Tender.

The Retail Domain is a set of business logic components that implement retail-oriented business functionality in Point-of-Service. The Retail Domain provides a common vocabulary that enables the expression of retail functionality as processes that can be executed by the Oracle Retail Platform engine.

The Retail Domain is a set of retail-oriented objects that have a set of attributes. They do not implement work flow or a user interface. The Tour scripts executed by Oracle Retail Platform provide the work flow, and the UI subsystem provides the user interface. The Retail Domain objects simply define the attributes and logic for application data.

A significant advantage of Retail Domain objects is that they can be easily used as-is or can be extended to include attributes and logic that are specific to a retailer's business requirements. The Domain objects could be used as a basis for many different types of retail applications. The objects serve as containers for the transient data used by the applications. Domain objects do not persist themselves, but they are persisted via the Oracle Retail Store Data Manager interface.

Retail Domain is packaged as domain360.jar and domainconfig.jar, which are installed with the Point-of-Service application. The Data Transactions and Data Operations are also packaged within the Retail Domain jars.

All Retail Domain classes extend EYSDomainIfc. This interface ensures the following interfaces are implemented:

Serializable
This communicates Java's ability to flatten an object to a data stream and, conversely, reconstruct the object from a data stream, when using RMI.

Cloneable
This communicates that it is legal to make a field-for-field copy of instances of this class.

The EYSDomainIfc interface also requires that the following methods be implemented:

equals()
This method accepts an object as a parameter. If the object passed has data attributes equal to this object, the method returns true, otherwise it returns false.

clone()
This method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object.

toString()
This method returns a String version of the object contents for debugging and logging purposes.

New Domain Object

When an existing Retail Domain object contains attributes and methods that are a subset of those required, a new Retail Domain object can extend the existing object. For example, if a new Domain object is necessary for the Tender service, the AbstractTenderLineItem class can be extended. This class implements TenderLineItemIfc, which extends the generic EYSDomainIfc interface. If no similar Domain object exists in the application, create a new Domain object. The usual coding standards apply; reference the Development Standards document.

1. Create a new interface extending EYSDomainIfc. All Retail Domain objects extend EYSDomainIfc, but existing Services have an interface available for Domain objects related to that Service. For example, TenderLineItemIfc, which extends EYSDomainIfc, is the interface implemented by each Retail Domain object interface in the Tender service. The following code sample shows the header of TenderPurchaseOrderIfc, found in *<source_directory>*\modules\domain\src\oracle\retail\stores\domain\tender\TenderPurchaseOrderIfc.java.

Example 15–1 TenderPurchaseOrderIfc.java: Class Header

```
public interface TenderPurchaseOrderIfc extends
TenderLineItemIfc, ReversibleTenderIfc
{
}
```

2. Create a new Java class that implements the interface created in the previous step. The class of a brand new object that does not fit an existing pattern should extend AbstractRoutable, which defines a luggage tag for EYS domain classes; otherwise, the class should extend the existing class that represents a similar type of object.

   The following code sample shows the header for the TenderPurchaseOrder Domain object from *<source_directory>*\modules\domain\src\oracle\retail\stores\domain\tender\TenderPurchaseOrder.java.

Example 15–2 TenderPurchaseOrder.java: Class Header

```
public class TenderPurchaseOrder extends AbstractTenderLineItem implements
TenderPurchaseOrderIfc
{
}
```

   In the implementation of the class, make sure to do the following:

   - Define attributes for the class.

     Check the superclass to see if an attribute has already been defined. For example, the AbstractTenderLineItem class defines the amountTender attribute, so amountTender should not be redefined in a new Tender Domain object.

If the new domain object has numerous constants, you might consider defining ObjectNameConstantsIfc.java

- Define get and set methods for the attributes as necessary.
- Implement methods required by EYSDomainIfc: equals(), clone(), toString(). Reference the superclass as appropriate. toString() should indicate the class name and revision number.

3. To return a new instance of the Domain object, add a method to *<source_directory>*\modules\domain\src\oracle\retail\stores\domain\factory\DomainObjectFactoryIfc.java called getObjectNameInstance().

   Domain objects should always be instantiated by the factory. The following code sample shows the method interface to return an instance of the TenderPurchaseOrder object.

Example 15–3 DomainObjectFactoryIfc.java: Method For Instantiating TenderPurchaseOrder

```
public TenderPurchaseOrderIfc getTenderPurchaseOrderInstance();
```

4. To return a new instance of the Domain object, implement the method *<source_directory>*\modules\domain\src\oracle\retail\stores\domain\factory\DomainObjectFactory.java called getObjectNameInstance().

   The following code sample shows the method definition to return an instance of the TenderPurchaseOrder object.

Example 15–4 DomainObjectFactory.java: Method For Instantiating TenderPurchaseOrder

```
public TenderPurchaseOrderIfc getTenderPurchaseOrderInstance()
{
return(new TenderPurchaseOrder());
}
```

Domain Object in Tour Code

Once a Retail Domain class is identified for use, the Java code needs to be written to instantiate the object and call the object's methods. This code is typically located in site, road and aisle classes of application tours. There are two very important things to keep in mind when using Domain objects in Tour code:

- Retail Domain objects cannot be instantiated directly. They must be generated by the factory.
- All interaction with Domain objects take place through the object's interface, even interaction between objects.

Do the following to use the object:

1. Get an instance of the DomainObjectFactory and request the instance of the object from the factory.

   The factory class is instantiated once for the application and returns instances of Retail Domain objects. Since different implementations use different classes to implement the objects, the factory keeps track of which class implements the requested object.

The following line of code from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\ado\tender\TenderCheckADO.java gets an instance of a Check object.

Example 15–5 GetCheckInfoSite.java: Instantiating Check from DomainObjectFactory

```
tenderRDO = DomainGateway.getFactory().getTenderCheckInstance();
```

2. Call methods on the object.

   Now that an instance of the object exists, methods of the class can be called. The following lines of code from TenderCheckADO.java sets attributes on the Check object.

Example 15–6 TenderCheckADO.java: Setting Attributes of Check

```
 if (eCheckAuthRequired())
        {
            ((TenderCheckIfc)
tenderRDO).setTypeCode(TenderLineItemConstantsIfc.TENDER_TYPE_E_CHECK);
        }
```

Domain Object Reference

The Domain Objects discussed below include a description of the purpose of the object, classes and interfaces involved in its construction, a class diagram, and examples in Tour code.

CodeListMap

To implement Point-of-Service metadata such as reasons for return, shipping methods, and departments, the CodeList objects are used. This data is referred to as reason codes from the UI. Codes are read in from the database.

The reason codes are managed in multiple languages/locales simultaneously by loading the ReasonCodes (CodeList) or ReasonCode(CodeEntry) on demand. The reason codes are externalized in the database: table ID_LU_CD_I8. The CodeListManager is used as the single point of access to retrieve the code lists from the database. For performance reasons, code list manager retrieves code lists from local derby database.

The sites and aisles call the following function to retrieve a code list. The code list contains localized text of all supported locales. The function in turn calls getCodeList API in CodeListManagerIfc:

```
CodeListIfc UtilityManager.getReasonCodes(String storeId, String codeListType);
```

Table 15–1 lists files are involved in the formation of CodeLists.

Table 15–1 CodeListMap Object Classes and Interfaces

Class or Interface	Description	Important Methods

CodeEntry	This class handles the functions associated with an entry in a list of codes.	`void setText(String)`
		`void setCode(int)`
		`void setEnabled(boolean)`
		`LocalizedTextIfc getLocalizedText();`
		`String getText(Locale lcl);`
		`void setLocalizedText (LocalizedTextIfc localizedText);`
		`void setText(Locale lcl, String value);`

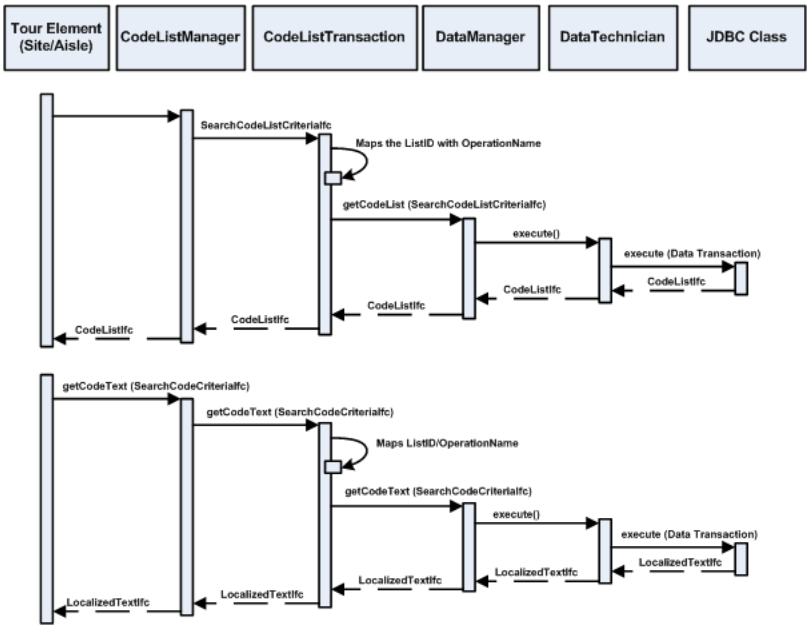
CodeList	This class is used for handling lists of codes which map to strings, such as reason codes.	`Vector<String> getTextEntries(Locale lcl)`
		`String[] getTextStrings(Locale lcl);`
		`CodeEntryIfc[] getEntries();`
		`CodeEntryIfc findListEntry(String str, boolean useDefault,Locale lcl);`
		`addEntry(CodeEntryIfc value);`
		`addEntry(LocalizedTextIfc text, String code, int index, boolean enable, String ref);`

CodeConstantsIfc	This class defines constants used for the implementation of CodeList and CodeEntry. It includes the constants for the lists currently defined, such as TimekeepingManagerEditReasonCodes and TillPayOutReasonCodes.	This class does not contain methods.

LocalizedCode	This class contains the localized code/text. All the Domain Objects, which contain a reason code, will use this class.	`void setCode (String code);`
		`String getCode ();`
		`void putText (Locale lcl, String text);`
		`String getText (Locale lcl);`
		`LocalizedTextIfc getText ();`
		`setText (LocalizedTextIfc text);`

Figure 15–1 illustrates loading CodeList / CodeText on Demand.

Figure 15–1 Loading CodeList / CodeText on Demand



To get the CodeListIfc, the Utility Manager provides two methods:

- CodeListIfc getReasonCodes(String storeId, String codeListType)
- String getReasonCodeText(CodeListIfc list, int reasonCode)

Tour code that requires a code entry would retrieve it as in the following code from *<source_directory>*\applications\pos\src\oracle\retail\stores\pos\services\returns\returnitem\ItemInfoEnteredAisle.java.

Example 15–7 ItemInfoEnteredAisle.java: CodeListIfc in Tour Code

```
UtilityManagerIfc utility =
          (UtilityManagerIfc) bus.getManager(UtilityManagerIfc.TYPE);
        Locale locale = LocaleMap.getLocale(LocaleConstantsIfc.USER_INTERFACE);
        if (model.getDepartmentName()!=null)
        {
        item.setDepartmentID
          (((utility.getReasonCodes(cargo.getStoreID(), CodeConstantsIfc.CODE_
LIST_DEPARTMENT)).
            findListEntry(model.getDepartmentName(), false, locale)).getCode());
        }
```

Currency

All currency representation and behavior is abstracted, so any currency can be implemented. Currency is a Domain Object that handles the behaviors and attributes of money used as a medium of exchange. It is important to use Currency objects and methods to compare and manipulate numbers instead of primitive types.

Table 15–2 lists Currency Object Classes and Interfaces.

Table 15–2 Currency Object Classes and Interfaces

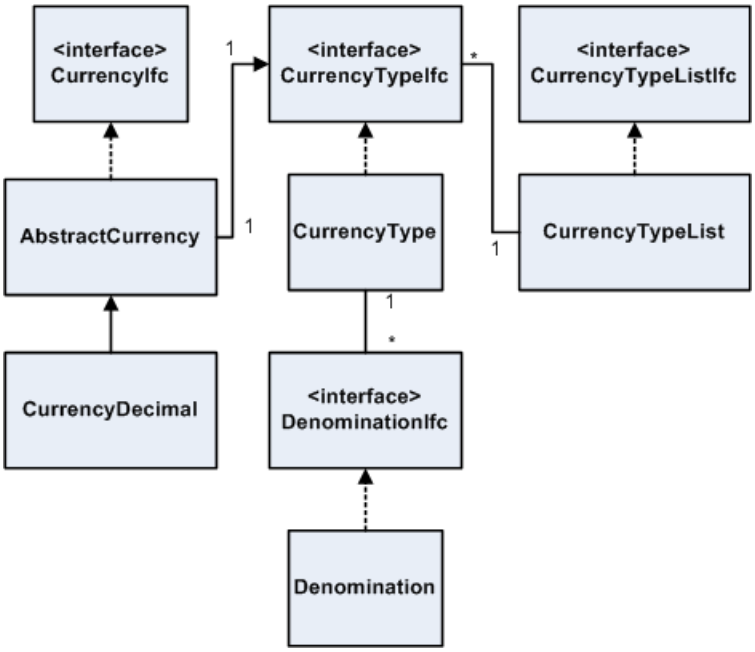
Class or Interface	Description	Important Methods

CurrencyIfc	This interface defines a common interface for currency objects.	`CurrencyIfc add(CurrencyIfc)`
		`CurrencyIfc negate()`
		`String getCountryCode()`

AbstractCurrency	This abstract class contains the behaviors and attributes common to all currency.	`BigDecimal getBaseConversionRate()`
		`void setNationality(String)`
		`String getNationality()`

CurrencyDecimal	This class contains the behaviors and attributes common to all decimal-based currency.	`CurrencyIfc add(CurrencyIfc)`
		`CurrencyIfc negate()`
		`String getCountryCode()`

Figure 15–2 Currency Class Diagram



Example 15–8 is an example of the Currency object used in *<source_directory>*`\applications\pos\src\oracle\retail\stores\pos\services\modifytransaction\discount\AmountEnteredAisle.java`.

Example 15–8 AmountEnteredAisle.java: CurrencyIfc in Tour Code

```
POSUIManagerIfc uiManager = (POSUIManagerIfc)bus.getManager(UIManagerIfc.TYPE);
        DecimalWithReasonBeanModel beanModel =
(DecimalWithReasonBeanModel)uiManager
                .getModel(POSUIManagerIfc.TRANS_DISC_AMT);
        BigDecimal discountAmount = beanModel.getValue();
CurrencyIfc amount =
DomainGateway.getBaseCurrencyInstance(String.valueOf(discountAmount));
```

Transaction

A Transaction is a record of business activity that involves a financial and/or merchandise unit exchange or the granting of access to conduct business with an external device. There are various types of Transactions found in *<source_directory>*`\modules\domain\src\oracle\retail\stores\domain\transaction` such

as LayawayTransaction, StoreOpenCloseTransaction, and BankDepositTransaction. SaleReturnTransaction is a commonly used Domain Object that extends AbstractTenderableTransaction. The classes involved in the implementation of a SaleReturnTransaction and its behaviors are described in the following table.

Table 15–3 lists classes involved in the implementation of a SaleReturnTransaction and its behaviors.

Table 15–3 Transaction Object Classes

Class or Interface	Description	Important Methods

SaleReturnTransaction	This class is a sale or return transaction.	`void SaleReturnLineItemIfc addPLUItem(PLUItemIfc pItem, BigDecimal qty);`
		`void addLineItem(AbstractTransactionLineItemIfc lineItem)`
		`void linkCustomer(CustomerIfc value)`
		`TransactionTotalsIfc getTenderTransactionTotals()`
		`void addLineItem(SaleReturnLineItemIfc);`
		`addLineItem(AbstractTransactionLineItemIfc)`

AbstractTenderableTransaction	This class contains the behavior associated with a transaction that involves the tendering of money.	`void addTenderLineItem(TenderLineItemIfc item);`
		`CustomerIfc getCustomer();`
		`CustomerIfc getCustomer()`
		`void addTender(TenderLineItemIfc item);`

Transaction	This class represents a record of business activity that involves a financial and/or merchandise unit exchange or the granting of access to conduct business at a specific device, at a specific point in time for a specific employee.	`CustomerInfoIfc getCustomerInfo()`
		`String getTillID()`
		`void setCashier(EmployeeIfc)`

Example 15–9 is a code sample from *<source_directory>*`\modules\domain\src\oracle\retail\stores\domain\arts\JdbcSaveTenderLineItems.java` that shows how SaleReturnTransaction is used in Tour code.

Example 15–9 JdbcSaveTenderLineItems.java: SaleReturnTransactionIfc in Tour Code

```
public void saveTenderLineItems(JdbcDataConnection
dataConnection,TenderableTransactionIfc transaction) throws DataException
{
  if (transaction instanceof SaleReturnTransactionIfc)
  {
    SaleReturnTransactionIfc srt = (SaleReturnTransactionIfc) transaction;
    int numDiscounts = 0;
    if (srt.getTransactionDiscounts() != null)
    {
      numDiscounts = srt.getTransactionDiscounts().length;
    }
  lineItemSequenceNumber = srt.getLineItems().length + 1 + numDiscounts;
  }
...code to handle different transaction types...
```

```
}
```

16

Extending Intra Store Data Distribution

This chapter describes how to extend the Intra Store Data Distribution.

Intra Store Data Distribution Extensibility

Extensibility is supported through the interface-based design and the use of the Spring Framework. From an extensibility stand point, an alternate implementation of any of the exposed interfaces could inherit from one of the out-of-the-box implementation classes and be injected into the system through Spring.

Additionally, the schema has been designed to enable the addition of datasets and dataset tables.

Adding New Table To Existing DataSet

Adding a new dataset table to the data model is as simple as adding a new row to the table CO_DT_ST_TB_IDDI and creating table script in CreateSchema.sql.

Adding More Tables To Existing DataSet Types

The following example walks through the process of adding more tables to the existing DataSet in IDDI.

1. Insert the tables to be associated with the existing DataSet by adding records to CO_DT_ST_TB_IDDI using SQL.

   Run the following queries to insert the table association to DataSet.

Example 16–1 Adding Table Association To Employee DataSet

```
insert into CO_DT_ST_TB_IDDI
(ID_DT_ST, ID_STR_RT, NM_TB, NM_FL,AI_LD_SEQ)
values
(<<Employee DataSet ID>>, <<'Store ID'>>,<<'Table1'>>,<<'Table1.txt'>>,1 );

TableName: CO_DT_ST_TB_IDDI

Column Description
ID_DT_ST : DataSet ID
ID_STR_RT: Store ID
NM_TB    : Table Name
NM_FL    : File Name of the Flat file to be generated
AI_LD_SEQ: Table Order in which the data to be exported and imported

eg: Get the Employee DataSet ID from CO_DT_ST_IDDI table
```

```
insert into CO_DT_ST_TB_IDDI
(ID_DT_ST, ID_STR_RT, NM_TB, NM_FL,AI_LD_SEQ)
values
(1,'04241','TABLE1','TABLE1.TXT',1 );

insert into CO_DT_ST_TB_IDDI
(ID_DT_ST, ID_STR_RT, NM_TB, NM_FL,AI_LD_SEQ)
values
(1,'04241','TABLE2','TABLE2.TXT',2 );
```

2. Add CREATE TABLE scripts in CreateSchema.sql.

```
CREATE TABLE "offlinedb"."TABLE1"
    ("COLUMN1" <<TYPE>> <<Constraint>>,
    "COLUMN2, <<TYPE>> <<Constraint>>)
CREATE TABLE "offlinedb"."TABLE2"
    ("COLUMN1" <<TYPE>> <<Constraint>>,
    "COLUMN2, <<TYPE>> <<Constraint>>)
```

Adding a Table to an Existing Data Set Using the Stores Build Scripts

Do the following to add a table using the build script:

1. Open `<source_directory>\modules\utility\build.xml`.
2. Find the target dataset's offline table list:

   `ordered.<data set name>.tables`

3. Add the name of the SQL file that contains the create script.

   The create scripts are located at `<source_directory>\modules\common\deploy\server\common\db\sql\Create`.

Adding a New DataSet

Do the following to add new DataSet:

1. Add DataSet information in CO_DT_ST_IDDI.
2. Add DataSet tables to CO_DT_ST_TB_IDDI.
3. Create `<DataSetKey>Producer` and `<DataSetKey>Consumer` classes extending from AbstractDataSetProducer and AbstractDataSetConsumer respectively.
4. Define `service_config_<<DataSetKey>>` in ServiceContext.xml
5. Define `service_<<DataSetKey>>Producer` with `class=<DataSetKey>Producer` and `service_<<DataSetKey>>Consumer` wit h `class=<DataSetKey>Consumer` in ServiceContext.xml
6. Add to `service_<<DataSetKey>>Producer` and `service_<<DataSetKey>>Consumer` to service_DataSetService and service_ClientDataSetService respectively in ServiceContext.xml
7. Add DataSet key to `service_FrequentProducerJob/service_InfrequentProducerJob` and `service_FrequentConsumerJob/service_InfrequentConsumerJob` in ServiceContext.xml
8. Add create table scripts and insert script for newly added DataSet in CreateSchema.sql.

Adding a New DataSet Using the Stores Build Scripts

Do the following to add a new dataset using the build script:

1. Open <source_directory>\modules\utility\build.xml .
2. Find the section that defines the offline table lists (target **assemble.iddi**).
3. Create the ordered list of tables, following the pattern established in the file. All create scripts are located at <source_directory>\modules\common\deploy\server\common\db\sql\Create.
4. Add a call to concat.file for the new data set schema, following the other calls in the file:

```
        <antcall target="concat.file">
            <param name="target.file" value="${raw.sql.file}"/>  -- The path
and name of the file being generated
            <param name="file.comment" value="-- Employee DataSet Tables"/> --
Comment added to the file ahead of the create SQL
            <param name="src.dir" value="${sql.src.dir}"/> -- Path to the
create scripts listed in the "ordered.<data set name>.tables" list
            <param name="file.list" value="${ordered.employee.tables}"/> --
Variable holding the ordered list of create scripts
            <reference refid="comment.filter" torefid="filter"/>
        </antcall>
```

Configuring Schedule for DataSet Producer and Consumer

Any existing DataSet Producer and Consumer can be individually configured to run on scheduled basis.

Configure DataSet Producer

Do the following to configure DataSet Producer:

1. Add JobDetailBean bean configuration service_<<DataSet>>ProducerJob.

```
    <bean id="service_<<DataSet>>ProducerJob"
class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass">

<value>oracle.retail.stores.foundation.iddi.DataSetProducerJob</value>
        </property>
        <property name="jobDataAsMap">
            <map>
                <entry key="producer" value-ref="service_DataSetService"/>
                <entry key="dataSets">
                    <list>
                        <ref local="service_config_<<DataSetKey>>"/>
                    </list>
                </entry>
            </map>
        </property>
    </bean>
```

Note: service_config_<<DataSetKey>> should have been configured with the DataSetKey

2. Add CronTriggerBean bean configuration service_Trigger<<DataSet>>Producer

```
<bean id="service_Trigger<<DataSet>>Producer" class =
```

```
"org.springframework.scheduling.quartz.CronTriggerBean">
        <property name = "jobDetail">
            <ref local="service_<<DataSet>>ProducerJob"/>
        </property>
        <property name="cronExpression" value="0 0,15,30,45 0 * * ?"/>
    </bean>
```

The above DataSet is configured to run once every 15 minutes.

For more information about the Quartz scheduler, see the documentation at http://www.quartz-scheduler.org/documentation.

3. Add `service_Trigger<<DataSet>>Producer` to the SchedulerFactoryBean bean configuration:

```
    <bean id="service_ProducerSchedulerFactory"
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <property name="triggers">
            <list>
                <ref local="service_TriggerFrequentProducer"/>
                <ref local="service_TriggerInfrequentProducer"/>
                <ref local="service_Trigger<<DataSet>>Producer"/>
            </list>
        </property>
    </bean>
```

Configure DataSet Consumer

Do the following to configure DataSet Consumer:

1. Add JobDetailBean bean configuration `service_<<DataSet>>ConsumerJob`:

```
    <bean id="service_<<DataSet>>ConsumerJob"
class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass">

<value>oracle.retail.stores.foundation.iddi.ClientDataSetController</value>
        </property>
        <property name="jobDataAsMap">
            <map>
               <entry key="dataSets">
                    <list>
                        <ref local="service_config_<< DataSetKey>>"/>
                    </list>
                </entry>
            </map>
        </property>
    </bean>
```

Note: `service_config_<<DataSetKey>>` should have been configured with the DataSetKey.

2. Add CronTriggerBean bean configuration `service_Trigger<<DataSet>>Consumer`:

```
    <bean id="service_Trigger<<DataSet>>Consumer" class =
"org.springframework.scheduling.quartz.CronTriggerBean">
        <property name = "jobDetail">
            <ref local="service_<<DataSet>>ConsumerJob"/>
        </property>
        <property name="cronExpression" value="0 0,15,30,45 0 * * ?"/>
    </bean>
```

The DataSet is configured to run once every 15 minutes.

3. Add `service_Trigger<<DataSet>>Consumer` to the SchedulerFactoryBean bean configuration:

```
    <bean id=" service_clientSchedulerFactory"
class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
        <property name="triggers">
            <list>
                <ref local="service_TriggerFrequentConsumer"></ref>
                <ref local="service_TriggerInfrequentConsumer"></ref>
                <ref local="service_Trigger<<DataSet>>Consumer"/>
            </list>
        </property>
    </bean>
```

Adding New DataSet Type

The following example walks through the process of adding a new DataSet to the existing IDDI.

- Insert the new DataSet information in into the databaset table CO_DT_ST_IDDI using SQL:
- Insert the tables associated with the DataSet added to CO_DT_ST_TB_IDDI using SQL.

1. Run the following queries to insert new DataSet information and table association to DataSet.

Example 16–2 Adding New DataSet

```
insert into CO_DT_ST_IDDI
(ID_DT_ST, ID_STR_RT, NM_DT_ST)
values
(maxid+1,<<'StoreID'>> ,<<'DataSetName'>>);

TableName: CO_DT_ST_IDDI

Column Description
ID_DT_ST : DataSet ID
ID_STR_RT: Store ID
NM_DT_ST : DataSet Name

eg:
insert into CO_DT_ST_IDDI
(ID_DT_ST, ID_STR_RT, NM_DT_ST)
values
(6,'04241','NEW');
```

Example 16–3 Adding Table association to New DataSet

```
insert into CO_DT_ST_TB_IDDI
(ID_DT_ST, ID_STR_RT, NM_TB, NM_FL,AI_LD_SEQ)
values
(<<New DataSet ID>>, <<'Store ID'>>,<<'Table1'>>,<<'Table1.txt'>>,1 );

eg:
insert into CO_DT_ST_TB_IDDI
(ID_DT_ST, ID_STR_RT, NM_TB, NM_FL,AI_LD_SEQ)
values
```

```
(6,'04241','TABLE1','TABLE1.TXT',1 );

insert into CO_DT_ST_TB_IDDI
(ID_DT_ST, ID_STR_RT, NM_TB, NM_FL,AI_LD_SEQ)
values
(6,'04241','TABLE2','TABLE2.TXT',2 );
```

2. Create <DataSetKey>Producer and <DataSetKey>Consumer classes extending from AbstractDataSetProducer and AbstractDataSetConsumer respectively.

Example 16–4 DataSetProducer Code

```
package oracle.retail.stores.domain.iddi;

import oracle.retail.stores.foundation.iddi.AbstractDataSetProducer;
import oracle.retail.stores.foundation.iddi.DataSetMetaData;
import oracle.retail.stores.foundation.iddi.TableQueryInfo;
import oracle.retail.stores.foundation.iddi.ifc.DataSetMetaDataIfc;

public class NewDataSetProducer extends AbstractDataSetProducer
{

private final String[] TABLE_FIELDS={"*"};

/**

 * NewDataSetProducer constructor
 */

public NewDataSetProducer ()
{

}
/**
* Get DataSetMetatIfc reference
*
*/
public DataSetMetaDataIfc getDataSetMetaData()
{
// Get the table names for the Key
return dataSetMetaData;
}
/**
* Initialize the MetaData for the DataSetProducer
*/
public void initializeDataSet()
{
dataSetMetaData = new DataSetMetaData(dataSetKey);
}
/**
* Create TableQueryInfo object with the column names to fetch
* @param TableName
* @return TableQueryInfo Object
*/
public TableQueryInfo getTableQueryInfo(String tableName)
{
TableQueryInfo tableQueryInfo = new TableQueryInfo(tableName);
tableQueryInfo.setTableFields(TABLE_FIELDS);
return tableQueryInfo;
}
```

```
/**
 * Finalize DataSet Method
 *
 */
public void finalizeDataSet()
{

}
}
```

Example 16–5 DataSetConsumer Code

```
package oracle.retail.stores.domain.iddi;

import oracle.retail.stores.foundation.iddi.AbstractDataSetConsumer;

//------------------------------------------------------------------------
/**
    The NewDataSetConsumer defines methods that the
    application calls to import Employee dataset files into
    offline database.
     @version $Revision: $
**/
//------------------------------------------------------------------------

public class NewDataSetConsumer extends AbstractDataSetConsumer
{         /**  DataSet key name for currency dataset.

    */    private String dataSetKey = null;

   //  ------------------------------------------------------------------------
    /**
        @return Returns the dataSetKey
    **/
    //------------------------------------------------------------------------

    public String getDataSetKey()
    {

return dataSetKey;
    }

    //  ------------------------------------------------------------------------
    /**
      @param dataSetKey The DataSetKey to set
    **/
    //------------------------------------------------------------------------

    public void setDataSetKey(String dataSetKey)
    {

this.dataSetKey = dataSetKey;

  }
}
```

3. Define `service_config_<<DataSetKey>>` in ServiceContext.xml:

```
        <bean id="service_config_<<datasetKey>> " class="java.lang.String">
            <constructor-arg type="java.lang.String" value="<<DataSetKey>>"/>
        </bean>eg:    <bean id="service_config_NEW_KEY" class="java.lang.String">
```

```
        <constructor-arg type="java.lang.String" value="NEW"/>
    </bean>
```

4. Define service_<<DataSetKey>>Producer with class=<DataSetKey>Producer and service_<<DataSetKey>>Consumer with class=<DataSetKey>Consumer in ServiceContext.xml:

```
<bean id="service_NewProducer"
class="oracle.retail.stores.domain.iddi.NewDataSetProducer" lazy-init="true"
singleton=”true”>
        <property name="dataSetKey" ref="service_config_NEW_KEY"/>
        <property name="dataExportFilePath" ref="service_config_
DataExportFilePath"/>
        <property name="dataExportZipFilePath" ref="service_config_
DataExportZipFilePath"/>
    </bean>
    <bean id="service_NewConsumer"
 class="oracle.retail.stores.domain.iddi.NewDataSetConsumer"
             lazy-init="true"
             singleton="true">
        <property name="dataSetKey" ref="service_config_NEW_KEY"/>
        <property name="dataImportFilePath" ref="service_config_
DataImportFilePath"/>
    </bean>
```

5. Add to service_<<DataSetKey>>Producer and service_<<DataSetKey>>Consumer to service_DataSetService and service_ClientDataSetService respectively in ServiceContext.xml

```
   <bean id="service_DataSetService"
class="oracle.retail.stores.foundation.iddi.DataSetService" singleton="true">
        <property name="producers">
            <map>
                <entry key-ref="service_config_EMP_KEY" value-ref="service_
EmployeeProducer"/>
                <entry key-ref="service_config_ITM_KEY" value-ref="service_
ItemProducer"/>
                <entry key-ref="service_config_PRC_KEY" value-ref="service_
AdvancedPricingProducer"/>
                <entry key-ref="service_config_TAX_KEY" value-ref="service_
TaxProducer"/>
                <entry key-ref="service_config_CUR_KEY" value-ref="service_
CurrencyProducer"/>
                <entry key-ref="service_config_NEW_KEY" value-ref="service_
NewProducer"/>
            </map>
        </property>
    </bean>
    <bean id="service_ClientDataSetService"
     class="oracle.retail.stores.foundation.iddi.ClientDataSetService"
singleton="true">
        <property name="consumers">
            <map>
                <entry key-ref="service_config_EMP_KEY" value-ref="service_
EmployeeConsumer"/>
                <entry key-ref="service_config_CUR_KEY" value-ref="service_
CurrencyConsumer"/>
                <entry key-ref="service_config_TAX_KEY" value-ref="service_
TaxConsumer"/>
                <entry key-ref="service_config_ITM_KEY" value-ref="service_
ItemConsumer"/>
```

```
                <entry key-ref="service_config_PRC_KEY" value-ref="service_
AdvancedPricingConsumer"/>
                <entry key-ref="service_config_NEW_KEY" value-ref="service_
NewConsumer"/>
            </map>
        </property>
        <property name="dataImportFilePath" ref="service_config_
DataImportFilePath"/>
    </bean>
```

6. Add DataSet key to `service_FrequentProducerJob/service_InfrequentProducerJob` and `service_FrequentConsumerJob/service_InfrequentConsumerJob` in ServiceContext.xml

```
    <bean id="service_FrequentProducerJob"
class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass">

<value>oracle.retail.stores.foundation.iddi.DataSetProducerJob</value>
        </property>
        <property name="jobDataAsMap">
            <map>
                <entry key="producer" value-ref="service_DataSetService"/>
                <entry key="dataSets">
                    <list>
                        <ref local="service_config_EMP_KEY"/>
                        <ref local="service_config_PRC_KEY"/>
                        <ref local="service_config_TAX_KEY"/>
                        <ref local="service_config_NEW_KEY"/>
                    </list>
                </entry>
            </map>
        </property>
    </bean>

<bean id="service_FrequentConsumerJob"
class="org.springframework.scheduling.quartz.JobDetailBean">
        <property name="jobClass">

<value>oracle.retail.stores.foundation.iddi.ClientDataSetController</value>
        </property>
        <property name="jobDataAsMap">
            <map>
               <entry key="dataSets">
                    <list>
                        <ref local="service_config_EMP_KEY"/>
                        <ref local="service_config_PRC_KEY"/>
                        <ref local="service_config_TAX_KEY"/>
                        <ref local="service_config_NEW_KEY"/>
                    </list>
                </entry>
            </map>
        </property>
    </bean>
```

7. Add CREATE TABLE scripts and insert scripts to newly added DataSet in CreateSchema.sql.

```
CREATE TABLE "offlinedb"."TABLE1"
    ("COLUMN1" <<TYPE>> <<Constraint>>,
 "COLUMN2, <<TYPE>> <<Constraint>>)
```

```
CREATE TABLE "offlinedb"."TABLE2"
    ("COLUMN1" <<TYPE>> <<Constraint>>,
 "COLUMN2, <<TYPE>> <<Constraint>>)
insert into CO_DT_ST_IDDI(ID_DT_ST, ID_STR_RT, NM_DT_ST)
values(6,'04241','NEW');
```

Adding a New DataSet Type Using the Stores Build Scripts

Do the following to add a new dataset type using the build script:

1. Open `<source_directory>\modules\utility\build.xml` .
2. Find the section that defines the offline table lists (target **assemble.iddi**).
3. Create the ordered list of tables, following the pattern established in the file. All create scripts are located at `<source_directory>\modules\common\deploy\server\common\db\sql\Create`.
4. Add a call to concat.file for the new data set schema, following the other calls in the file:

```
        <antcall target="concat.file">
            <param name="target.file" value="${raw.sql.file}"/>  -- The path
and name of the file being generated
            <param name="file.comment" value="-- Employee DataSet Tables"/> --
Comment added to the file ahead of the create SQL
            <param name="src.dir" value="${sql.src.dir}"/> -- Path to the
create scripts listed in the "ordered.<data set name>.tables" list
            <param name="file.list" value="${ordered.employee.tables}"/> --
Variable holding the ordered list of create scripts
            <reference refid="comment.filter" torefid="filter"/>
        </antcall>
```

Changing Oracle Retail Point-of-Service Client Database Vendor

Currently the Oracle Retail Point-of-Service client uses Derby Database. However, the modifications to the code are minimal for replacing the Oracle Retail Point-of-Service client database from Derby to another database. Do the following to change the Oracle Retail Point-of-Service client database:

1. Add `Offline<<DBName>>Helper` class which implements `offlineDBHelperIfc`.
2. Change the installer to have new database driver jar file paths.
3. Update the `"<POOL name="jdbcpool class="DataConnectionPool" package="oracle.retail.stores.foundation.manager.data">"` section of PosLFFDataTechnician.xml file with the driver, databaseUrl, userid, password.

Index

M

P

R

S

T

U

W