360Store®

Back Office
Developer Guide

Confidential

11400 Burnet Road, Suite 5200, Austin, Texas 78758
(512) 491-2600 • (800) 897-8663 • (512) 491-0078 (fax)
www.360Commerce.com

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODE SAMPLES

PREFACE

Audience

This document is intended for 360Store® Back Office developers who develop code for a Back Office implementation.

Goals

Developers who read this document should be able to:

- Extend Back Office classes
- Create a Back Office UI screen

Feedback

Please e-mail feedback about this document to 360University@360Commerce.com.

Trademarks

The following trademarks may be found in 360Commerce® documentation:

- 360Commerce, 360Store and 360Enterprise are registered trademarks of 360Commerce Inc.
- Unleashed is a trademark of 360Commerce Inc.
- BEETLE is a registered trademark of Wincor Nixdorf International GmbH.
- Dell is a trademark of Dell Computer Corporation.
- IBM, WebSphere and SurePOS are registered trademarks or trademarks of International Business Machines Corporation in the United States, other countries, or both.
- IceStorm is a trademark of Wind River Systems Inc.
- InstallAnywhere is a registered trademark of Zero G Software, Inc.
- Internet Explorer and Windows are registered trademarks or trademarks of Microsoft Corporation.
- Java is a trademark of Sun Microsystems Inc.
- Linux is a registered trademark of Linus Torvalds.
- Mac OS is a registered trademark of Apple Computer, Inc.

- Netscape is a registered trademark of Netscape Communication Corporation.
- UNIX is a registered trademark of The Open Group.

All other trademarks mentioned herein are the properties of their respective owners.

Text Conventions

The following table shows the text conventions used in this document:

Table P-1 Conventions

Sample	**Description**
`Courier Text`	Filenames, paths, syntax, and code
Bold text	Emphasis
<*Italics and angle brackets*>	Text in commands which should be supplied by the user

CHAPTER 1

ARCHITECTURE

Overview

This chapter describes the main layers of the application, and goes into some detail about the middle tier's use of a model-view-controller (MVC) pattern. The remainder of this overview covers the top-level tier organization of the application and how the application relates to other 360Commerce applications in an enterprise environment. This guide assumes a basic familiarity with the J2EE specification and industry standard software design patterns.

The architecture of Back Office reflects its overriding design goals:

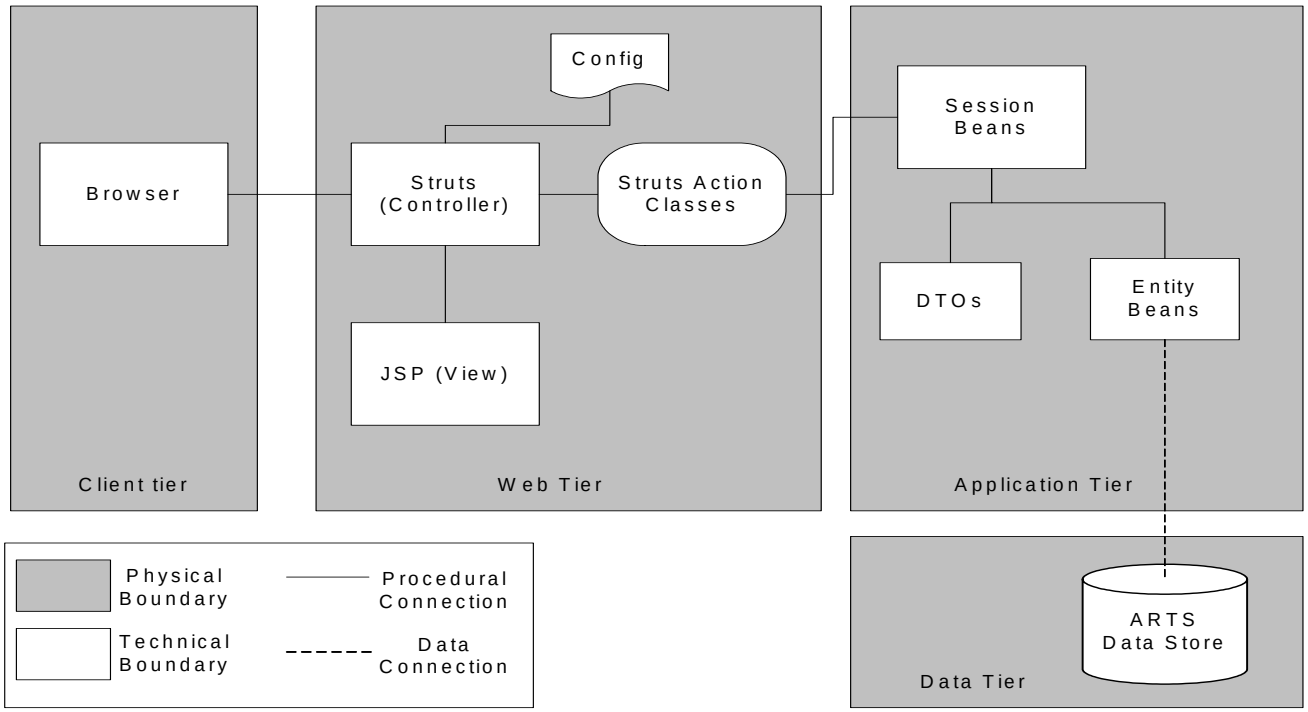
- Well-defined and decoupled tiers
- Use appropriate J2EE standards
- Leverage other open standards where possible

Tier Organization

The architecture of Back Office uses client, middle, and data tiers. The client tier is a Web browser; the middle tier is deployed on an application server; and the data tier is a database deployed by the retailer.

The middle tier is organized in an MVC design pattern, also called a Model 2 pattern. This chapter focuses on the middle tier and the model, view, and controller layers that it is divided into.

Figure 1-1 High-Level Architecture



Client Tier

The client system uses a Web browser to display data and a GUI generated by the application. Any browser which supports JavaScript, DHTML, CSS, and cookies can be used. In practice, only a few popular browsers are tested.

Middle Tier

The middle tier of the application resides in a J2EE application server framework on a server machine. The middle tier implements the MVC pattern to separate data structure, data display, and user input.

Model

The model in an MVC pattern is responsible for storing the state of data and responding to requests to change that state which come from the controller. In Back Office this is handled by a set of Commerce Services, which encapsulates all of the business logic of the application. The Commerce Services talk to the database through a persistence layer of entity beans.

Commerce Services are components that have as their primary interface one or more session beans, possibly exposed as Web services, which contain the shared retail business logic. Commerce Services aggregate database tables into objects, combining sets of data into logical groupings. Commerce Services are organized by business logic categories rather than application functionality. These are services like Transaction, Store Hierarchy, or Parameter that would be usable in any retail-centric application.

These services in turn make use of a persistence layer made up of entity beans. Each Commerce Service talks to one or more entity beans, which map the ARTS standard database schema. Using the bean-managed persistence (BMP) pattern, each entity bean maps to a specific table in the schema, and knows how to read from and write to that table. The Commerce Services thus insulate the rest of the application from changes to the database tables. Database changes can be handled through changes to a few entity beans.

The Commerce Services architecture is designed to facilitate changes without changing the product code. For example:

- You can replace a specific component's implementation. For example, the current Store Hierarchy service persists store hierarchy information to the ARTS database. If a customer site has that information in an LDAP server, the Store Hierarchy could be replaced with one that connected to the LDAP. The interface to the service need not change.
- You can create a new service that wraps an existing service (keeping the interface and source code unchanged), but adds new fields. You might create My Customer Service, which uses the existing Customer Service for most of its information, but adds some specific data. All that you change is the links between the Application Manager and the Customer Service.

For more information, see Chapter 5, "Commerce Services."

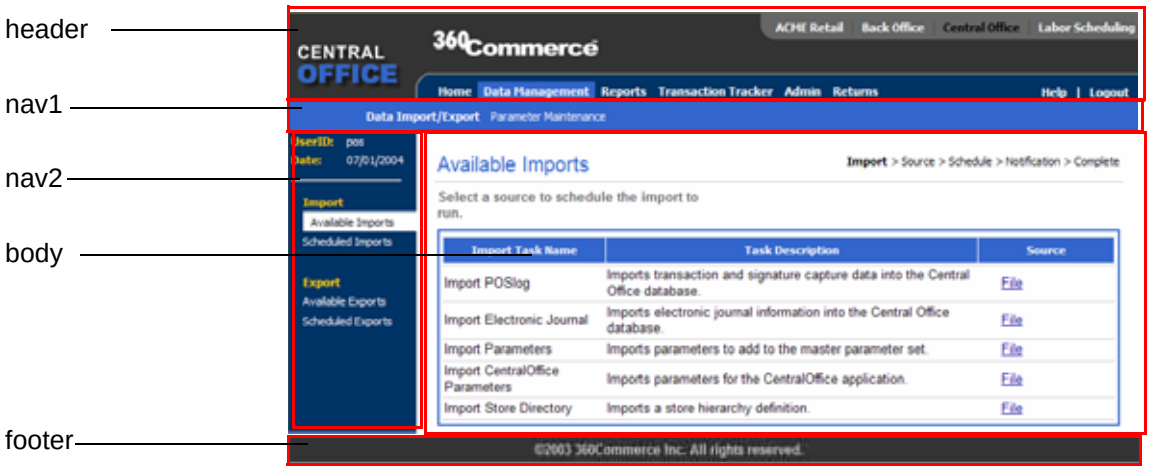
View

The view portion of the MVC pattern displays information to the user. In Back Office this is performed by a Web user interface organized using the Struts/Tiles framework. Using Tiles for page layout allows for greater reuse of the user interface components to enhance the extensibility and customization of the user interface.

To make the view aware of its place in the application, the Struts Actions call into the Application Manager layer for all data updates, business logic, and data requests. Any code in the Struts Actions should be limited to formatting data for the Java server pages (JSPs) and organizing data for calls into the Application Manager layer.

Java Server Pages deliver dynamic HTML content by combining HTML with Java language constructs defined through special tags. Back Office's pages are divided into Tiles which provide navigation and page layout consistency.

Figure 1-2 Tiles in a 360Commerce Application



Controller

The controller layer accepts user input and translates that input into calls to change data in the model layer, or change the display in the view layer. Controller functions are handled by Struts configuration files and Application Services.

Struts Configuration

The application determines which modules to call upon an action request based on the `struts-config.xml` file.There are several advantages to this approach:

- The entire logical flow of the application is in a hierarchical text (xml) file. This makes it easier to view and understand, especially with large applications.
- The page designer does not need to read Java code to understand the flow of the application.
- The Java developer does not need to recompile code when making flow changes.

Struts reads the `struts-config.xml` once, at startup, and creates a mapping database (a listing of the relationships between objects) that is stored in memory to speed up performance.

Application Services

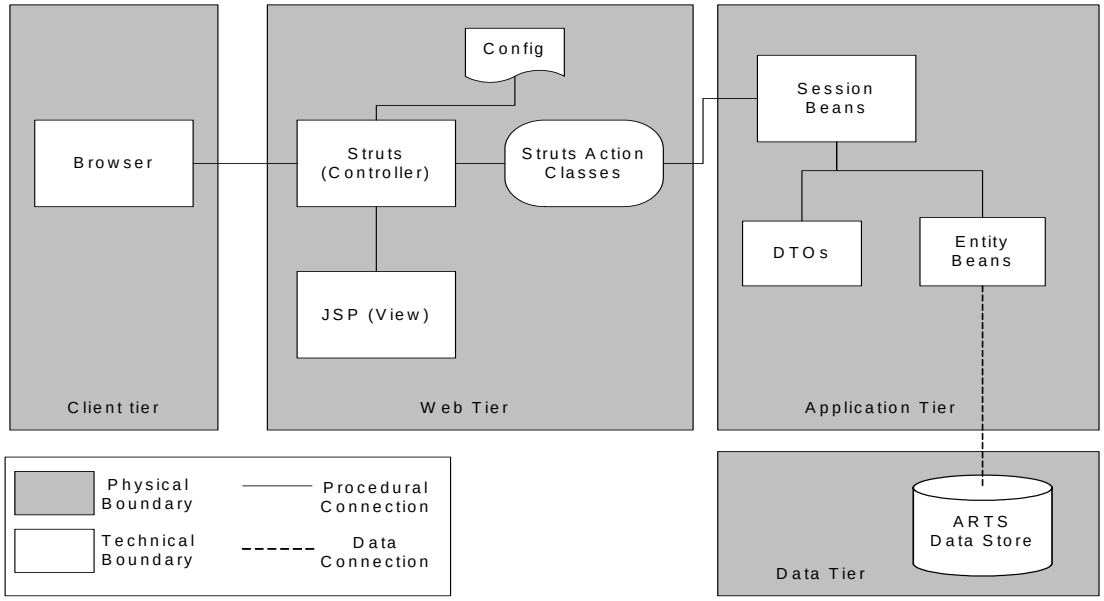
The application services layer contains logical groupings of related functionality specific to the Back Office application components, such as Store Operations. Each grouping is called an application manager. These managers contain primarily application logic. Retail domain logic should be kept out of these services and instead shared from the Commerce Services tier.

The application services use the Session Facades pattern; each Manager is a facade for one or more Commerce Services. A typical method in the Application Services layer aggregates several method calls from the Commerce Services layer, allowing the individual Commerce Services to remain decoupled from each other. This also strengthens the Web user interface tier and keeps the transaction and network overhead to a minimum.

For example, the logic for assembling and rendering a retail transaction into various output formats are handled by separate Commerce Services functions. However, the task of creating a PDF file is modeled in the EJournal Manager, which aggregates those separate Commerce Service functions into a single user transaction, thus decreasing network traffic and lowering maintenance costs.

For more information, see Chapter 4, “Application Services.”

Figure 1-3 Application Manager as Facade for Commerce Services



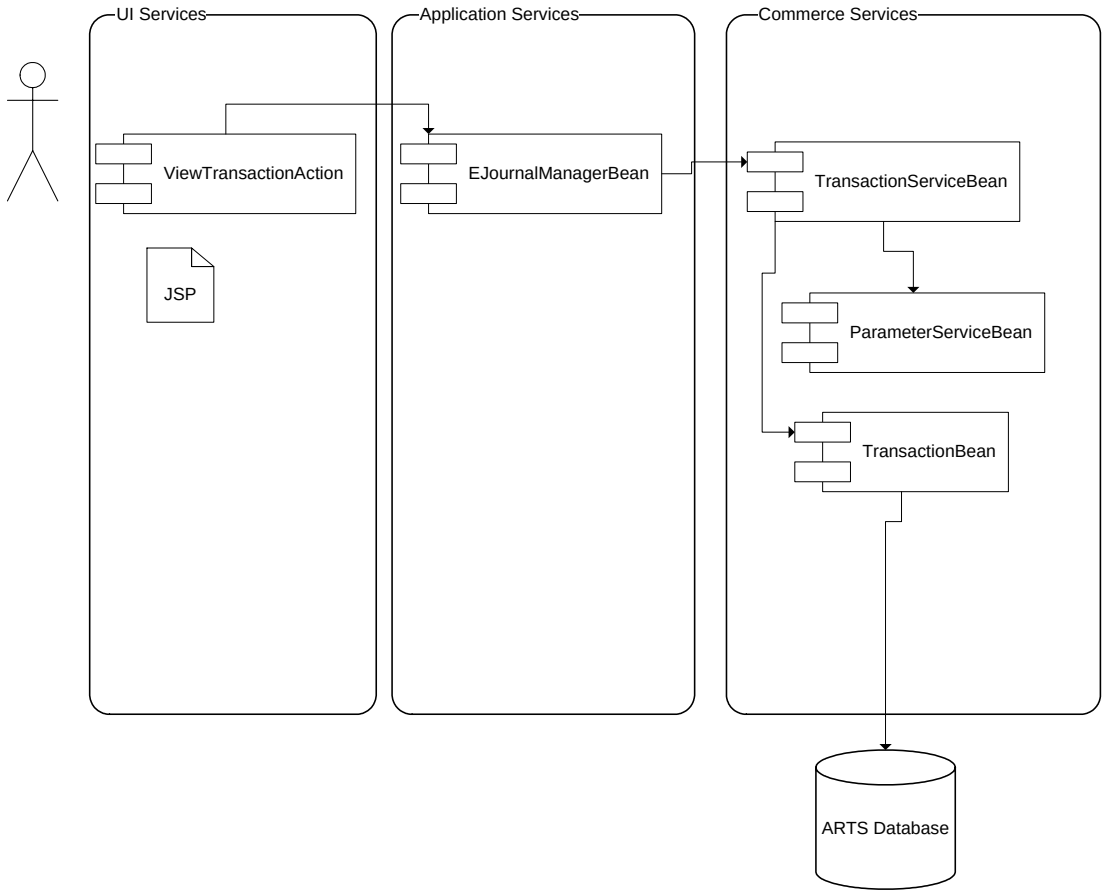
Data Tier

The Data tier is represented by a database organized using the ARTS standard schema. Customer requirements determine the specific database selected for a deployment. For more information, see Chapter 6, “Store Database.”

Dependencies in Application and Commerce Services

The following diagram shows representative components Application Services and Commerce Services. Arrows show the dependencies among various components.

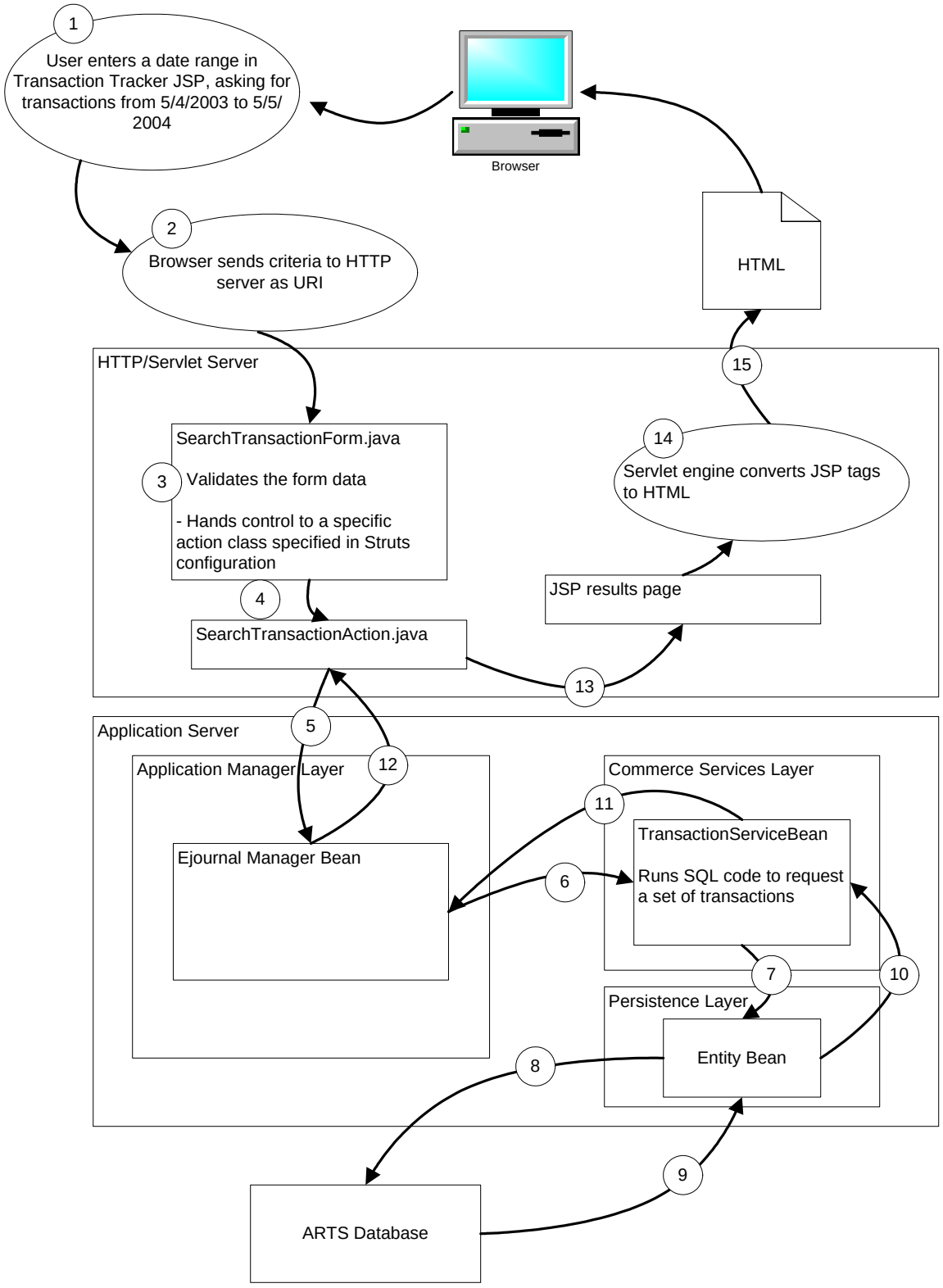
Figure 1-4 Dependencies in Back Office



Example of Operation

The following diagram describes a trip through the Back Office architecture, starting from a user's request for specific information and following through until the system's response is returned to the user's browser.

Figure 1-5 Operation of Back Office



CHAPTER 2

CODING YOUR FIRST FEATURE

Overview

This chapter describes how to add a feature to Back 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.

Related Materials

See the `_resources` directory provided with your Back Office documentation for sample requirements documents for the project described in this chapter.

See "Example of Operation" on page 1-6 for a diagram that shows a search query's trip through the Back Office architecture and the Transaction Tracker to return transaction data.

Before You Begin

Before you attempt to develop new Back Office code, set up your development environment as described in the preceding chapter. 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 Back 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 allows 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

As an example of how to extend Back 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 allows the user to set a quantity value and search for transactions whose quantity of any item compares appropriately to a chosen quantity (i.e., greater than, greater than or equal to, less than, etc.).

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.

Web UI Framework

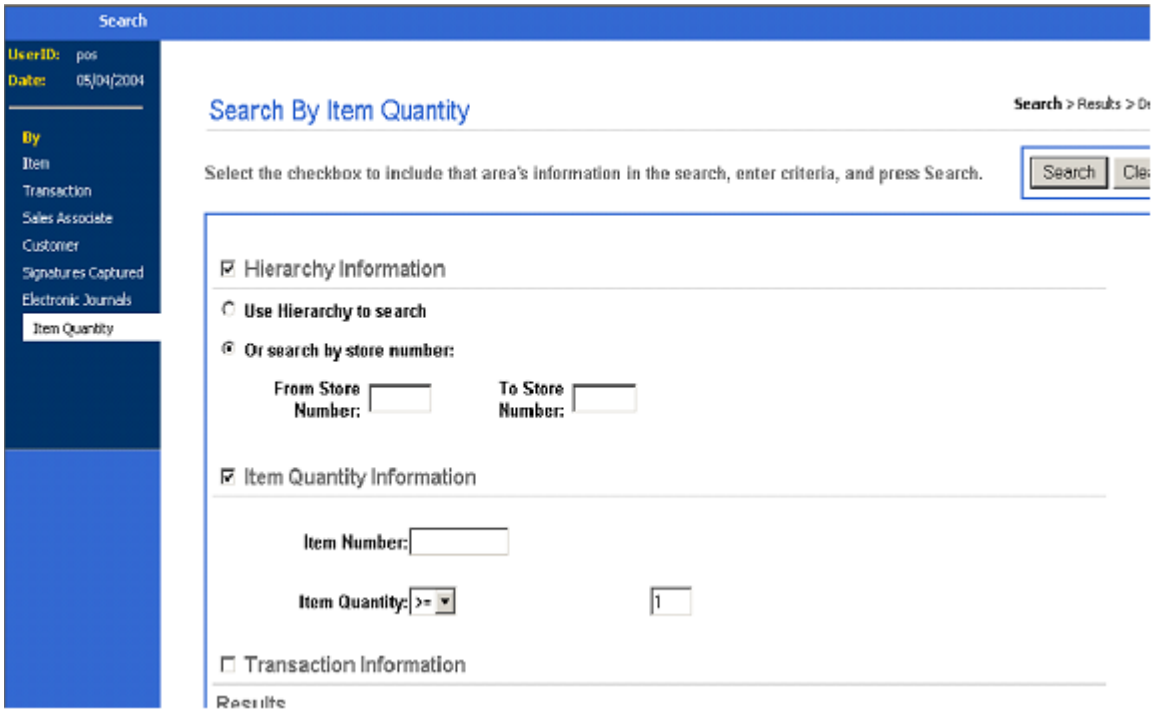
To add a new search criteria page, you must create a new JSP file for the page, edit workflow and Struts/ Tiles configuration files to register the page, and add appropriate classes to handle the page.

Create a New JSP file

Create a new JSP file and edit its 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 `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 2-1 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 `/webmodules/i18n/ui/src/com/_360commerce/webmodules/i18n/transaction.properties` file.

to add these 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 file `/webmodules/transaction/ui/deploy/360/tiles-transaction_tracker.xml`, 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:

Code Sample 2-1 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.

Code Sample 2-2 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>
```

Configure Action Mapping

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 `/webmodules/transaction/ui/deploy/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.

Code Sample 2-3 Struts Action Configuration for Item Quantity

```
<action path="/centralizedElectronicJournal/ejItemQuantitySearch"
       type="com._360commerce.webmodules.transaction.ui.StartSearchAction">
       <forward name="success" path="centralizedElectronicJournal.ejItemQuantitySearch"/>
</action>

<action path="/centralizedElectronicJournal/searchTransactionByItemQuantity"
       type="com._360commerce.webmodules.transaction.ui.SearchTransactionByItemQuantityAction"
       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 `webmodules/transaction/ui/src/com/_360commerce/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.

Code Sample 2-4 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.

Code Sample 2-5 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.

Code Sample 2-6 Code to Add to Validate Method

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

Code Sample 2-7 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 `/centraloffice/deploy/wepapp/classes/ApplicationResources.properties`.

   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 `webmodules/transaction/ui/src/com/_360commerce/webmodules/transaction/ui/`. You can start by copying and modifying `SearchTransactionByItemQuantityAction.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.

Code Sample 2-8 Call a New Method to Get Item Quantity Criteria

```
searchCriteria = new SearchCriteria(getTransactionCriteria(searchTransactionForm,
```

```
request.getParameterValues(
                                                                "transactionType")),

getTenderCriteria(searchTransactionForm),

getSalesAssociateCriteria(searchTransactionForm),

getLineItemCriteria(searchTransactionForm),

getLineItemQuantityCriteria(searchTransactionForm),

getCustomerCriteria(searchTransactionForm),

getSignatureCaptureCriteria(searchTransactionForm));
```

2. Add the method implementation.

Code Sample 2-9 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;
    }
```

Verify Application Manager Implementation

Verify that the application manager appropriately calls for information from Commerce Services. In the Item Quantity search criteria example, the `EJournalManagerBean.java` class is used. This class already contains the necessary method implementation for a getTransactions() method.

Add Business Logic to Commerce Service

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

The example below, established in `\suite\transaction\src\com\_360commerce\commerceservices\transaction\LineItemQuantityCriteria.java` handles these requirements.

Code Sample 2-10 LineItemQuantityCriteria.java

```
    private String    itemNumber;
    private String    ItemQuantityLimitOperator;
    private int       ItemQuantityLimit;
    private boolean searchByItemNumber;
    private boolean searchByItemQuantity;

    /**
     * Returns the itemNumber to include in the
     * search criteria.
     *
     * @return String
     */
    public String getItemNumber()
    {
        return itemNumber;
    }

       /**
        * Sets the itemNumber.
        * @param itemNumber The itemNumber to set
        */
       public void setItemNumber(String itemNumber)
       {
              this.itemNumber = itemNumber;
        searchByItemNumber = true;
       }

    /**
     * Returns the itemQuantityLimit to include in the
     * search criteria.
     *
     * @return String
     */
    public int getItemQuantityLimit()
    {
        return ItemQuantityLimit;
    }

       /**
        * Sets the itemNumber.
        * @param itemNumber The itemQuantityLimit to set
```

```
         */
        public void setItemQuantityLimit(int ItemQuantityLimit)
        {
               this.ItemQuantityLimit = ItemQuantityLimit;
         searchByItemQuantity = true;
        }

    /**
     * Returns the itemQuantityLimitOperator to include in the
     * search criteria.
     *
     * @return String
     */
    public String getItemQuantityLimitOperator()
    {
        return ItemQuantityLimitOperator;
    }

        /**
         * Sets the ItemQuantityLimit.
* @param ItemQuantityLimit The ItemQuantityLimitOperator to
*  set
         */

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

    /**
     * Returns the searchByItemNumber.
     * @return boolean
     */
    public boolean isSearchByItemNumber()
    {
        return searchByItemNumber;
    }

    /**
     * Returns the searchByItemQuantity.
     * @return boolean
     */

    public boolean isSearchByItemQuantity()
    {
        return searchByItemQuantity;
    }
    /**
     *
     * Indicates whether line item count criteria should be
     * included in a database query.
     *
     * @return boolean
     *
     */
    public boolean use()
    {
        return (isSearchByItemQuantity());
    }
    /**
     *
*  Resets criteria values to defaults and isSearchBy flags to
```

```
*  false.
     *
     */
    public void reset()
    {
        itemNumber          = null;
        ItemQuantityLimitOperator = null;
        ItemQuantityLimit      = 0;
        searchByItemNumber  = false;
        searchByItemQuantity = false;
    }
```

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
`\suite\transaction\src\com\_360commerce\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 the following code sample:

Code Sample 2-11 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

`\suite\transaction\src\com\_360commerce\commerceservices\transaction\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.

Code Sample 2-12 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.

Code Sample 2-13 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.

Code Sample 2-14 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;
    }
```

CHAPTER 3

DEVELOPMENT ENVIRONMENT

Overview

This chapter describes how to set up a single-user development environment for Back Office. The setup described here provides all the files, tools, and resources necessary to build and run the Back Office application.

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

This chapter assumes that you are using the JBoss application server and the MySQL database (version 4.104 or later). These are free open-source tools whose footprint is well suited for an individual developer environment. You may use different tools for deployment.

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.

Using the Apache Ant Build Tool

360Commerce uses the Apache Ant build tool to compile builds. Ant uses build information defined in various `build.xml` files and properties files. Each top-level directory in the source control system contains a `build.xml` file that specifies a variety of *targets*, or build tasks, for use with Ant.

Since each module depends on other modules, a `\build` directory is included whose `build.xml`file 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.

Prerequisites for the Development Environment

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

- A source control system, with access to a copy of the Back Office source code.
- A database server and database. The default database name is bo01. 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 1.4.1. Downloads and instructions are available at http://sun.com. The `JAVA_HOME` environment variable needs to be set in the OS and the `%JAVA_HOME%\bin` directory needs to be added to the path.

Setting Up the Development Environment

Follow these steps to set up a Back Office development environment on your computer system. Paths in these instructions refer to directories in your local source code workspace, unless otherwise specified.

1. Check out each of the following directories from your source code control system to your local working directory:
   - thirdparty
   - 360common
   - commerceservices
   - suite
   - webmodules
   - backoffice_ee
   - build
   - user
2. Install Apache Ant 1.5.4.

   Get it from the `\thirdparty` module's `\dist` directory. The version stored there includes some additional `.jar` files required for the tool to work with 360Commerce software. More information on Ant is available at http://ant.apache.org/.

   Set the `ANT_HOME` environment variable to the name of the directory you installed Ant in, and add the `%ANT_HOME%\`bin directory to the `PATH` environment variable.

3. Install JBoss 3.2.1.

   Get it from the `\thirdparty` module's `\dist` directory. More information on JBoss is available at http://jakarta.com.

   Set the `ANT_HOME` environment variable to the name of the directory you installed Ant in, and add the `%ANT_HOME%\bin` directory to the `PATH` environment variable.

4. Copy the sample `build.properties` file from `\user\examples` to your user directory (such as `c:\Documents and Settings\`*`username`* on a Windows system).

5. Change the value of `test.envs` in the `build.properties` file in your user directory to point to the `\user` directory in your workspace.

   Code Sample 3-1 build.properties: Setting test.envs Property

```
#Generic build stuff
deprecation=off

#test stuff
test.envs=<workspace>/user/${ant.project.name}/jboss
```

6. Edit the `user.properties` file in the `\user\backoffice\jboss` directory to point to the correct database and deployment directory.

   Code Sample 3-2 user.properties: Editing Database Property

```
# my props
db.product=<database_name>
jboss.deploy.dir=C:/jboss-3.2.1/server/default/deploy
```

7. Edit the `log4j.properties` file in the `\user\backoffice\jboss` directory to point to the appropriate log location in your workspace.

   Code Sample 3-3 log4j.properties: Editing Log Location

```
... code omitted here...
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%d %-5p [%c] %m%n
log4j.appender.R.File=C:/eclipse/workspace/user/jboss/test.log
...code omitted here...
```

   This change causes logs created when you run functional tests to end up in an appropriate location.

8. Edit the `mysql.properties` file in the `\user\db\` directory as follows, making sure the properties shown in bold have the correct values.

   Code Sample 3-4 mysql.properties: Editing Database Properties

```
# mysql
db.product=mysql
db.version=4.0
db.jdbc-driver=com.mysql.jdbc.Driver
db.jdbc-jar=<mysql directory path>/mysql-connector-java-3.0.8-stable-bin.jar
db.datafilter=com._360commerce.datafilters.MySQLDataFilter
db.jdbc-url=jdbc:mysql://localhost:3306/bo01
db.user=<username>
db.password=<password>
```

9. From a command prompt, change to the `\build` directory and execute the following command:

```
ant backoffice_ee
```

   This compiles Back Office based on the instructions in `build.xml` and the settings in `build.properties`. The compile process generates a `backoffice.ear` file located in the `dist` directory.

10. From a command prompt, change to the `\backoffice_ee` directory and execute the following command:
```
ant create_db
```
This causes the system to run a series of SQL scripts to create the database tables and seed data.

11. From a command prompt, execute the following command:
```
ant deploy
```
This copies the `backoffice.ear` file to the `<JBOSS_HOME>\default\deploy` directory.

12. Copy the contents of your `\backoffice_ee\appservers\jboss\3.2.1\server` directory to the `<JBOSS_HOME>\server` directory.

These files configure JBoss. They include:

- JMS queue definitions
- Default security
- Quartz properties
- Data source definitions
- Executable `.jar` files for database connections and third-party applications
- Modified startup script

13. Copy appropriate `.jar` files for your database from the `\suite\3rdparty\special_jars\` directory to the `<JBOSS_HOME>\server\default\lib` directory.

For example, if you are using MySQL, copy the `mysql-connector-java-3.0.8-stable-bin.jar` file; this installs the Connector/J JDBC driver.

14. Define the data source for the correct database in JBoss by editing a data source file found in the `<JBOSS_HOME>\server\default\deploy` directory. The data source file has a filename of the form `<databasename>-ds.xml`.

For example, for MySQL, the filename is `mysql-ds.xml`.

- Uncomment the `jndi-name` tag and modify the value to `jdbc/DataSource`.
- Uncomment the `connection-url` tag and specify the correct hostname or IP address for your database.
- Update the `user-name` and `password` tag values to include the correct database username and password.

The following code sample illustrates these changes:

Code Sample 3-5 Sample MySQL Data Source Configuration

```
<datasources>
  <local-tx-datasource>
    <jndi-name>jdbc/DataSource</jndi-name>
    <connection-url>jdbc:mysql://localhost:3306/bo01</connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>username</user-name>
    <password>password</password>
  </local-tx-datasource>
</datasources>
```

15. Define the secondary data source in `other-ds.xml`. Edit it as you did the data source file named after your database type; it should be identical, except that the `<jndi-name>` tag should read `<jndi-name>jdbc/Other</jndi-name>`.
16. Delete all files named `*-ds.xml` in the `<JBOSS_HOME>\server\default\deploy` directory *except* for these:
    - `other-ds.xml`
    - `hsql-ds.xml`
    - The one for your database (if you are using MySQL, keep the `mysql-ds.xml` file)

Run and Configure Back Office

To verify the setup and provide it with configuration data, run the Back Office application using the following steps:

1. Start JBoss with a run script from its \bin directory:

   `c:\<JBOSS_HOME>\bin\run.bat` (for Windows)

   or

   `<JBOSS_HOME>/bin/run.sh` (for Linux)
2. Load parameter by issuing the following command:
   `ant load_parameters`
3. In a browser, load the following URL to display Back Office and verify that it works.

   http://localhost:8080/backoffice
4. Log in to the application with the default login (username pos, password pos) to verify that it works.

CHAPTER 4

APPLICATION SERVICES

Overview

Application Services have the job of requesting information from Commerce Services and returning that information to the Web UI in a format that can be displayed by it.

360Commerce implements application services in the form of application managers. Application managers aggregate services from multiple Commerce Services into a smaller number of interfaces, and correspond generally to a specific portion of the application user interface.

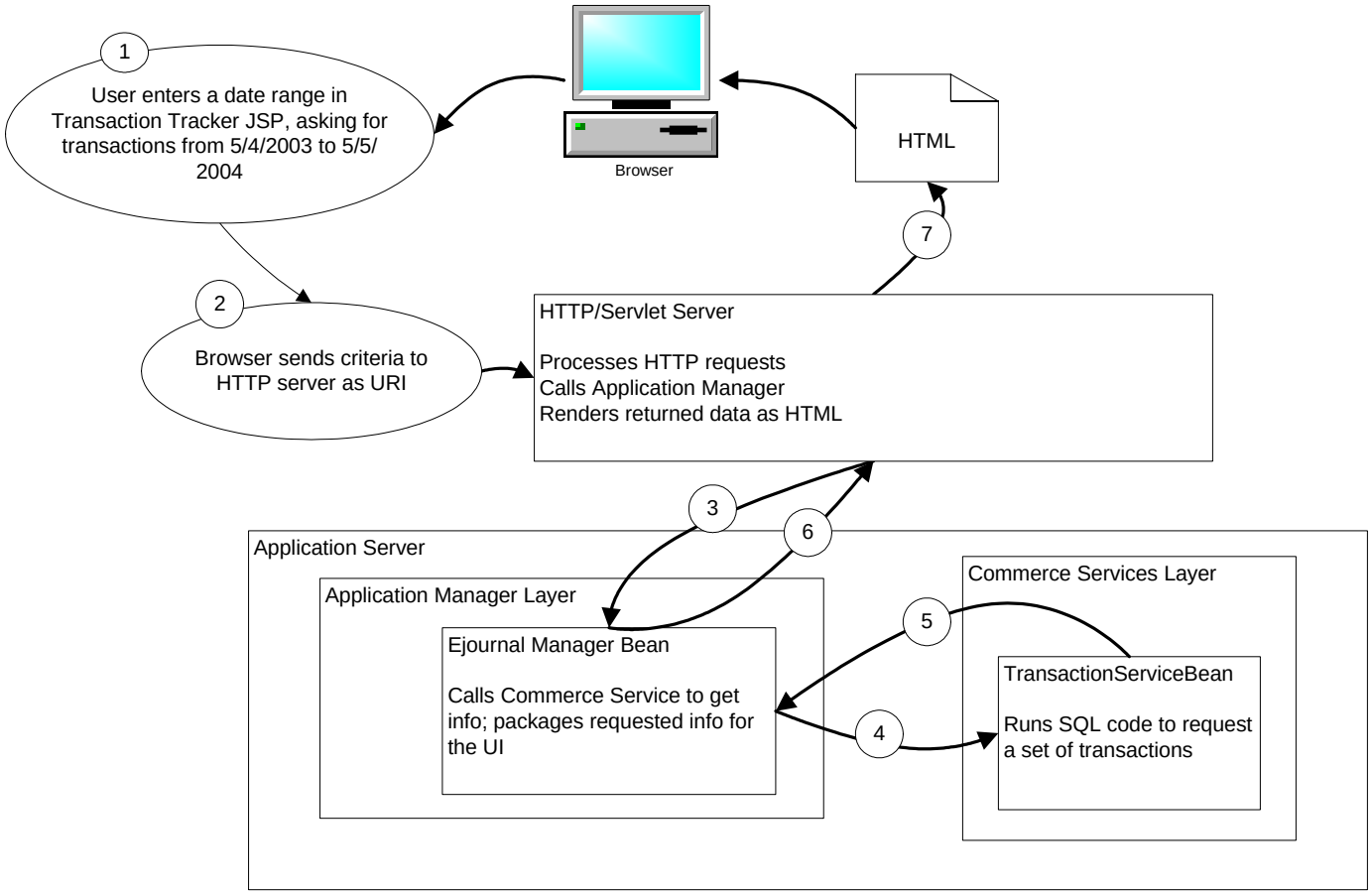
The presence of the Application Services layer offers opportunities for customization that can make your implementation of Back Office more stable across upgrades. This pattern optimizes network traffic, as requests for multiple Commerce Services tend to be funneled through a smaller number of application managers.

These services contain primarily application logic. Business logic should be kept out of these services and instead shared from the Commerce Services tier. In many cases the only function of an Application Service method is to call one or more Commerce Services. Each manager is a facade for one or more Commerce Services. A typical method in the Application Services layer aggregates several method calls from the Commerce Services layer, allowing the real retail business components to remain decoupled from each other.

Application managers are called by Struts Action classes to execute functionality that ultimately derives from Commerce Services. Struts Action classes should not call Commerce Services directly.

Figure 4-1, “Application Manager in Operation” on page 4-2 shows how an Application Manager functions within the application.

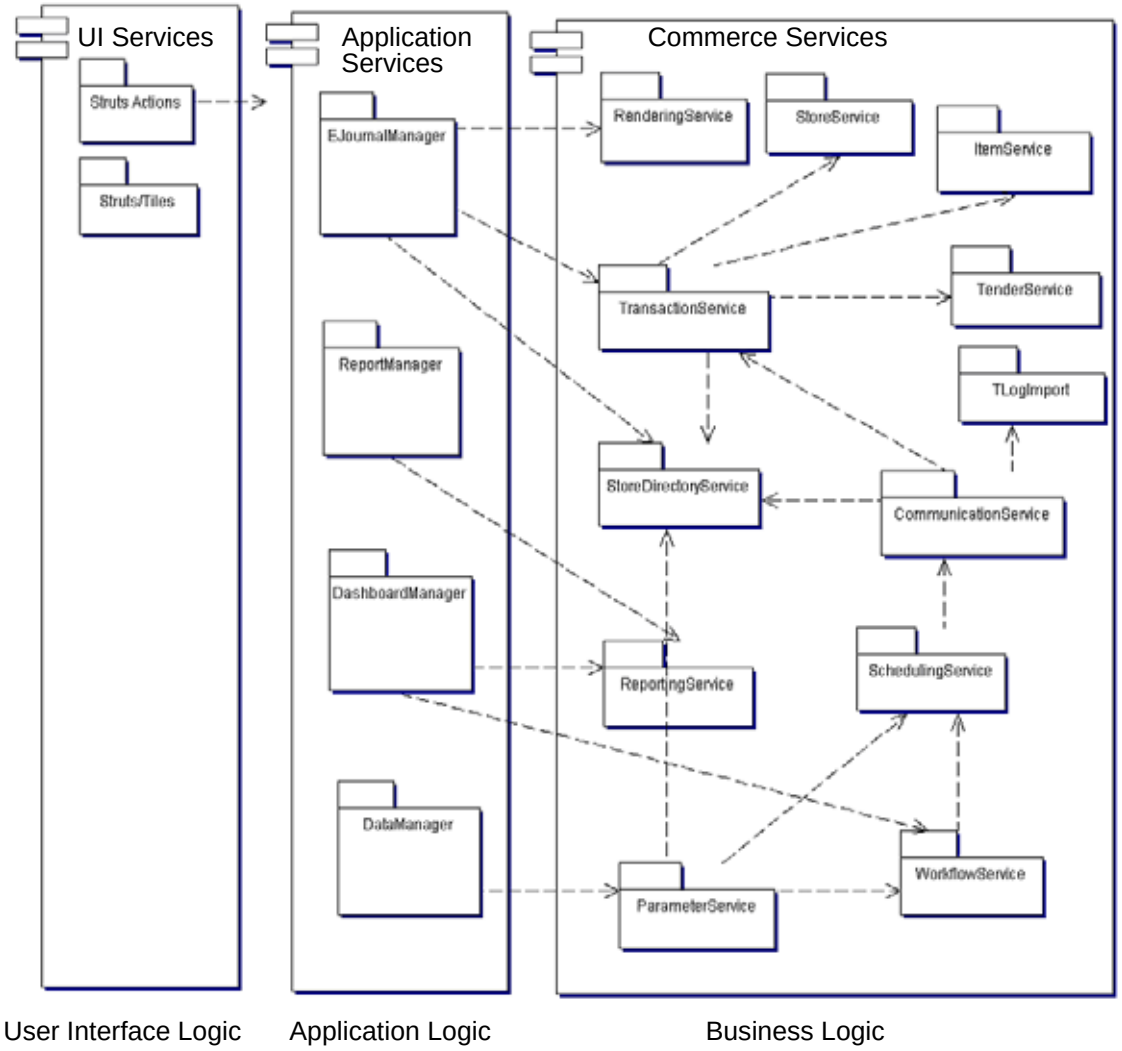
Figure 4-1 Application Manager in Operation



Application Service Architecture

The following diagram shows the relationship between the user interface, the application services, and the Commerce Services.

Figure 4-2 Example Application Service Interactions



Application Manager Mapping

Table 4-1 shows how individual Application Managers map to various parts of the application.

Table 4-1 Application Manager Mapping

Tab	Manager	API	Functions
Home	Dashboard Manager	DashboardManagerIfc.java	Manipulating the employee task list on the Dashboard
Reports	Report Manager	ReportManagerIfc.java	Displaying, executing, and scheduling the reports

Table 4-1 Application Manager Mapping

Tab	Manager	API	Functions
Admin	Task Manager	TaskManagerIfc.java	Scheduling tasks
	Employee Manager	EmployeeManagerIfc.java	Managing security groups, users, and employees
Pricing	Pricing Manager	PricingManagerIfc.java	Managing price changes, promotions, and discount rules
Item	Item Manager	ItemManagerIfc.java	Searching for items
Store Ops	Store Ops Manager	StoreOpsManagerIfc.java	Opening and closing the store, registers, and tills

Extending an Application Manager

The application manager layer provides an opportunity for customizing application behavior without changing the underlying Commerce Services. Some examples of reasons to extend or modify an application manager include:

- To change content that comes from Commerce Services. You can remove data or change how it is handled, formatted, or displayed by changing the logic in the application manager.
- You can provide additional data to your JSPs via the application managers, either by supplying data that comes from existing Commerce Services functions but is not displayed by the default user interface, or by calling new, custom Commerce Services.
- When you add input fields to the user interface, you must make sure that the appropriate application manager knows about those fields and knows how to handle them. If you are extending search criteria, for example, the application manager has to be able to pass those criteria on to the Commerce Service layer.

Creating a New Application Manager

The following steps outline the requirements for making a new application manager:

1. Make new `EJB.jar` for the application manager.
   - New directory `\webmodules\<new_app_manager_name>`
   - `build.xml` file for ant configurations
   - `\app` directory that contains `\classes`, `\deploy`, `\dist`, `\src` and `\test` directories
   - `\ui` directory that contains Struts/Tiles `.jsp` files
2. Edit application configuration files
   - Edit `build.xml` file for `\backoffice` to add your new module to the `suite.modules` property list
   - `application.xml`: add a tag for your EJB to the list of EJBs, as `manager_name-admin-ejb.jar`

3. Edit UI files

Create UI references in Struts configuration files, as described in Chapter 2, “Coding Your First Feature.”

Application Manager Reference

All of the managers are stateless session facades which provide functionality in a UI-centric form to be called by Struts Action classes associated with various JSPs. The topics in this section describe the individual application managers.

Dashboard Manager

Provides functions for displaying and manipulating the employee task list on the Dashboard, displayed when users click the Home tab in the user interface.

Dependencies

- Workflow/Scheduling Service
- Reporting Service
- Employee/User Service

EJournal Manager

EJournal Manager handles functionality related to the Transaction Tracker tab in the user interface. It allows searches for transactions based on a variety of criteria and combinations of criteria.

Dependencies

- Parameter Service
- Tender Service
- Transaction Service
- Customer Service
- Store Directory
- Reporting Service

ItemManager

Item Manager handles item search functions for the Item tab in Back Office.

Dependencies

- Item Service

- Store Directory

Report Manager

Provides functions for displaying, executing, and scheduling the reports, as well as managing lists of user favorite reports. Supports the application's Reports tab.

Dependencies

- Workflow/Scheduling Service
- Store Directory
- Reporting Service
- ReportGroupTaskExecutionMDB

Store Manager

Provides the ability to read and write information about a store to and from the database. This includes store address and store hierarchy information.

Dependencies

- Workflow/Scheduling Service
- Store Directory
- Employee/User Service

StoreOps Manager

Provides store operations functions. This includes Start of Day, End of Day, and Deposit operations, as well as opening and closing registers and opening and reconciling tills. This manager handles tasks for the Store Ops tab in Back Office.

Dependencies

- StoreOps Service
- Parameters Service
- Currency Service

Task Manager

Handles workflow and displays job information.

Dependencies

- Workflow Service

- File Transfer Service

CHAPTER 5

COMMERCE SERVICES

Overview

The topics in this chapter describe each of the available Commerce Services. The Commerce Services in Back Office provide the model component of the MVC pattern; they store the state of data and respond to requests to change that state which come from the controller. The Commerce Services are intended to encapsulate all of the business logic of the application. They are built as session beans, sometimes exposed as Web services, which contain the shared retail business logic. Commerce Services aggregate database tables into objects, combining sets of data into logical groupings. They are organized by business logic categories rather than application functionality. These are services like Transaction, Store Hierarchy, or Parameter, which could be used with any retail-centric application. The Commerce Services talk to the database through a persistence layer of entity beans, described in Chapter 6, “Store Database.”

For each service, this chapter includes a description, a listing of the database tables used by the service, plus notes on extending the service and a list of dependencies on other services. The database tables listed are those which are updated by the service directly, excluding any services merely accessed by the service, or which are updated through other services.

This chapter covers the following services:

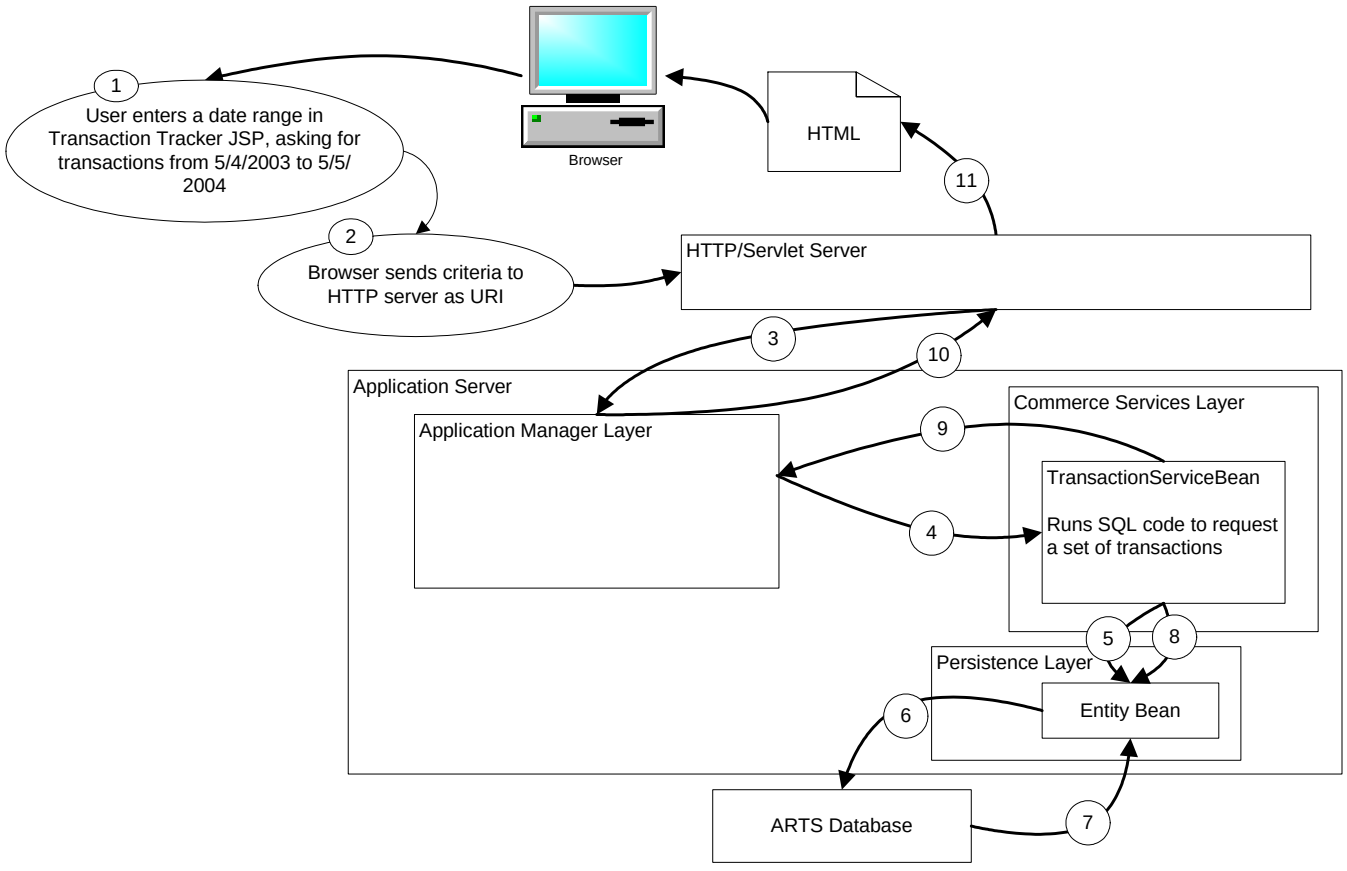
- Calendar Service
- Code List Service
- Currency Service
- Customer Service
- Employee/User Service
- File Transfer Service
- Financial Totals
- Item Service
- Parameter Service
- Party Service
- POSlog Import Service

- Post-Processor Service
- Pricing Service
- Reporting Service
- Store Directory Service
- Store Service
- Store Ops Service
- Tax Service
- Time Maintenance Service
- Transaction Service
- Workflow/Scheduling Service

Commerce Services in Operation

Figure 5-1, “Commerce Services in Operation” on page 5-2 shows how the Commerce Services function within the application.

Figure 5-1 Commerce Services in Operation



Creating a New Commerce Service

To create a new Commerce Service, use the following basic steps:

1. Make a new `EJB.jar` with the following components:
   - New directory `\suite\<new_service_name>`
   - A `build.xml` file for ant configurations
   - `\classes, \deploy, \dist, \src` and `\test` directories
2. Edit application configuration files
   - Edit the `build.xml` file for `\backoffice` to add the module to the `suite.modules` property list
   - `application.xml`: add a tag for the EJB to the list of EJBs
3. Edit Application Service and UI files.
   - Update Application Service to call methods in the Commerce Service.
   - Create UI references in Struts configuration files.

Calendar Service

Package of business-calendar-related functionality for reporting.

Database Tables Used

- CA_CLD (Calendar)
- CA_CLD_LV (Calendar Level)
- CA_CLD_PRD (Calendar Period)
- CA_PRD_RP_V4 (Calendar Reporting Period V4)

Interfaces

Access the service through `CalendarServiceIfc.java`:

Code Sample 5-1 CalendarServiceIfc.java: Methods

```
CalendarReportRangeDTO getReportingPeriods(int calendarId, CalendarLevel level, Date startDate, Date
endDate) throws RemoteException, FinderException;
Collection getReportingPeriodsAllLevels(int calendarId, Date transactionTime) throws RemoteException,
FinderException, CreateException;
void createCalendar(int id, String name) throws RemoteException, CreateException;
void removeCalendar(int id) throws RemoteException, RemoveException;
```

Extending This Service

You could extend this service to change how dates are handled. For example, the default service provides year, month, week, and day as units for reporting. You might want to add quarters to this list. Doing so would require adding code to the service to handle resolving data to the new unit. If, on the other hand, you merely wanted to remove one of these units, for example to remove reporting by week, you could do so by changing the database alone.

Dependencies

None.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Code List Service

The Code List Service allows Web applications to retrieve code lists from various sources:

- Inventory codes
- Advanced Pricing codes
- POS Department codes
- Suspended Transactions codes

Database Tables Used

ID_LU_CD (CodeList)

Interfaces

Access this interface through `CodeListServiceIfc.java`. The following code sample shows the available methods:

Code Sample 5-2 CodeListServiceIfc.java: Methods

```
public interface CodeListServiceIfc
{

    /**
     * Retrieve the inventory Reason Codes
     */
    public Collection getInventoryCodeList() throws RemoteException, FinderException;

    /**
```

```
     * Retrieve reason codes by store and description
     * @param storeId
     * @param description
     * @return
     * @throws RemoteException
     * @throws FinderException
     */
    public Collection getReasonCodeByStoreAndDescription(String storeId, String description) throws
RemoteException, FinderException;

    /**
     * Retrieve reason codes by store and description and group
     * @param storeId
     * @param description
     * @param group
     * @return
     * @throws RemoteException
     * @throws FinderException
     */
    public Collection getReasonCodeByStoreAndDescriptionAndGroup(String storeId, String description,
String group) throws RemoteException, FinderException;

    /**
     * Retrieve reason code value for a given entry name
     * @param storeId
     * @param description
     * @param group
     * @param entryName
     * @return
     * @throws RemoteException
     * @throws FinderException
     */
    public ReasonCodeDTO  getReasonCodeByName(String storeId, String description, String group, String
entryName) throws RemoteException, FinderException;

    /**
     * Retrieve reason code for a given entry value
     * @param storeId
     * @param description
     * @param group
     * @param entryValue
     * @return
     * @throws RemoteException
     * @throws FinderException
     */
    public ReasonCodeDTO  getReasonCodeByValue(String storeId, String description, String group, String
entryValue) throws RemoteException, FinderException;

    /**
     * Retrieve default reason code in a group
     * @param storeId
     * @param description
     * @param group
     * @return
     * @throws RemoteException
     * @throws FinderException
     */
    public ReasonCodeDTO  getDefaultReasonCode(String storeId, String description, String group) throws
RemoteException, FinderException;

    /**
     * Returns a collection of PosDepartmentDTOs.
     * @return Collection of PosDepartmentDTO
     * @throws RemoteException
```

```
     * @throws FinderException
     */
    public Collection getPosDepartments() throws RemoteException, FinderException;

    DBUtilsIfc getDBUtils() throws RemoteException;

}
```

Extending This Service

You can add additional codes to the system without extending this service, as it simply retrieves the set of codes that exist.

Dependencies

None.

Tier Relationships

The functionality of this service is the same whether it is used in Central Office or Back Office.

Currency Service

The Currency Service allows you to query for the base local currency setting (from the database). It also handles addition of currency, including currency in multiple denominations.

Database Tables Used

- CO_CNY (Currency List)
- CO_RT_EXC (Exchange Rates)

Interfaces

Access this interface through `CurrencyIfc.java`. The following code sample shows a few of the available methods:

Code Sample 5-3 CurrencyIfc.java: Some Methods

```
/**
        Adds this object to another CurrencyIfc object. <P>
        @param addCurrency object
        @return new value as object
    */
    //--------------------------------------------------------------------
    public CurrencyIfc add(CurrencyIfc addCurrency);

    //--------------------------------------------------------------------
    /**
```

```
    Subtracts CurrencyIfc object from this object. <P>
    @param subCurrency object
    @return new value as object
*/
//---------------------------------------------------------------------
public CurrencyIfc subtract(CurrencyIfc subCurrency);

//---------------------------------------------------------------------
/**
    Multiplies this object times another CurrencyIfc object. <P>
    @param multCurrency object
    @return new value as object
*/
//---------------------------------------------------------------------
public CurrencyIfc multiply(CurrencyIfc multCurrency);

//---------------------------------------------------------------------
/**
    Multiplies this object times another CurrencyIfc object. <P>
    @param multCurrency object
    @return new value as object
*/
//---------------------------------------------------------------------
public CurrencyIfc multiply(BigDecimal multCurrency);
```

Extending This Service

The default service supports U.S. dollars, Canadian dollars, Mexican pesos, Japanese yen, and British pounds (pound sterling); it could be extended to handle additional currencies, and to allow the addition of multiple currencies to each other, with appropriate handling of exchange rates.

The service could also be extended to connect to a ASP to get exchange rates or other currency information.

Dependencies

None.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office; however, it is currently only used by Back Office.

Customer Service

The Customer Service is used to locate customers' information. Typically this information is displayed as additional details to a transaction.

Database Tables Used

PA_CT (Customer)

Interfaces

Access the service through `CustomerServiceIfc.java`:

Code Sample 5-4 CustomerServiceIfc.java: Methods

```
public CustomerDTO getCustomer(String customerID) throws RemoteException, SearchException;
```

Extending This Service

In a deployment, you could extend this service by connecting it to a Customer Relationship Management (CRM) application. The service encapsulates the 360Commmerce customer data function so that other portions of the application do not have to change should such a connection be implemented.

Dependencies

Party Service.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Employee/User Service

Searches for employees and updates their details, including security permissions.

Database Tables Used

- CO_ACS_GP_RS (GroupResourceAccess)
- CO_GP_WRK (WorkGroup)
- PA_EM (Employee)

Interfaces

Access the service through `EmployeeServiceIfc.java`, which offers methods for finding, adding, and updating employee records. The following code sample provides some examples:

Code Sample 5-5 EmployeeServiceIfc.java: Some Methods

```
   /**
     * Finds the employee with the specified employee ID.
```

```
     *
     * @param employeeId the ID of the employee to find.
     * @return A DTO containing the employee data.
     * @throws RemoteException
     */
    EmployeeDTO getEmployee(String employeeId) throws EmployeeNotFoundException, RemoteException;

    /**
     * Finds the employees whose first and last names begin with the specified strings.
     *
     * @param firstName the beginning characters of the employee's first name.
     * @param lastName the beginning characters of the employee's last name.
     * @return an array of DTO's containing data about employees that match the search criteria.
     * @throws RemoteException
     */
    EmployeeDTO[] searchEmployees(String firstName, String lastName) throws RemoteException;
```

Extending This Service

You could extend this service by replacing it with a connection to a personnel database or application, such as an LDAP system.

Dependencies

Party Service.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

File Transfer Service

The File Transfer Service passes arbitrary files from one component of the system to another. It stores the files in the database.

Database Tables Used

- FILE_SET (File Set)
- FILE_SET_ITEM (File Set Item)

Interfaces

Access the service through `FileTransferIfc.java`:

Code Sample 5-6 FileTransferServiceIfc.java: Methods

```
public interface FileTransferServiceIfc
```

```
{
    FileSetDTO createFileSet(String name, String description) throws RemoteException;

    FileSetDTO getFileSet(int id) throws RemoteException;

    FileSetDTO addItemToFileSet(int id, FileSetItemDTO fileSetItemDTO) throws RemoteException;

    FileSetDTO removeItemFromFileSet(int id, String fileName) throws RemoteException;

    DistributionPayload getDistributionPayload(String source) throws RemoteException;

}
```

Extending This Service

If your project has a particularly optimized solution for storing files on a server, you might want to replace this service with your own solution.

Dependencies

Store Directory.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Financial Totals

This service provides functions for getting financial totals for transactions and till history. Each type of transaction and history has an associated Financial Total calculator to derive the various values in the Financial Totals classes from the particular transaction type.

Database Tables Used

This service does not have persistent storage of its own; it relies on data from other services.

Interfaces

Access the service through `FinancialTotalsServiceIfc.java`. The following code sample shows the available methods:

Code Sample 5-7 FinancialTotalsServiceIfc.java

```
       FinancialTotalsIfc getFinancialTotals(Collection transactions) throws
FinancialTotalCalculationException, RemoteException;

    TransactionFinancialTotalsDTO getFinancialTotals(TransactionDTO dto) throws
FinancialTotalCalculationException, RemoteException;
```

```
    TillHistoryFinancialTotalsDTO getFinancialTotals(TillHistoryDTO tillHistory, TillTenderHistoryDTO[]
tillTenderHistory) throws FinancialTotalCalculationException, RemoteException;

    UpdatedTillHistoryFinancialTotalsDTO getFinancialTotalsAtReconcile(WorkstationDTO workstation,
TillHistoryDTO tillHistory, TillTenderHistoryDTO[] tillTenderHistory, TenderAmountDTO endFloat,
TenderAmountDTO[] tillCounts) throws FinancialTotalCalculationException, RemoteException;

    FinancialUtilIfc getFinancialUtil() throws RemoteException;
```

Extending This Service

You could extend this service to perform additional financial aggregations. You could add calculators for additional transaction types or alter the existing calculators.

Dependencies

- Item Service
- Currency Service
- Transaction Service

Tier Relationships

The functionality of this service is the same whether it is used in Central Office or Back Office.

Item Service

The Item Service provides item creation, item search, and item record maintenance. Includes the ability to import item information: a flat file or XML file of item information is imported and can then be processed immediately or scheduled for later upload. The Item Service can take one of three actions on each item listed in an import: Add, Update, or Delete.

Database Tables Used

- AS_ITM (Item)
- AS_ITM_RTL_STR (Retail Store Item)
- AS_ITM_STK (Stock Item)
- CO_CLN_ITM (Item Collection)
- CO_CLR (Item Color)
- CO_STYL (Item Style)
- CO_SZ (Item Size)
- CO_UOM (Item Unit Of Measure)

- ID_IDN_PS (POS Identity)

Interfaces

Use `ItemServiceIfc.java`, which offers methods to get, update, and import items, search for items, and to get specific information about items, such as units of measure, available colors, and available locations.

Code Sample 5-8 ItemServiceIfc.java: Some Methods

```
     /**
      * @param itemID
      */
    ItemDTO getItem(String itemID) throws RemoteException, FinderException;

     /**
      * updates the passed in dto, including possibly changing default data
      *
      * @param dto
      */
    void updateItem(ItemInfoDTO dto) throws RemoteException;


    /**
     *  Imports items to the database.
     *
     *  @param content - Content containing items to be processed.
     */
    void importItem(String content) throws RemoteException;

    /**retrieves an item by its full key, if the item isn't found it returns default item data
     *
     * @param storeID
     * @param posItemID
     */
    ItemInfoDTO getAllItemInfo(String storeID, String posItemID, String itemID) throws RemoteException;

    /** retrieves item data, the collection may contain 0-n iteminfodtos
     *
     * @param storeID
     * @param posItemID
     */
    Collection searchForItems(Collection storeIDs, ItemSearchCriteria criteria) throws RemoteException;

    /**
     * @param storeID
     * @param description
     */
    Collection findByDescription(String storeID, String description) throws RemoteException,
FinderException;

    /**
     * @param storeID
     * @param itemID
     */
    Collection findByPOSItemID(String storeID, String posItemID) throws RemoteException,
FinderException;

    /**
     *  Returns a collection of ColorDTOs representing all the available colors for items.
     *
     */
    Collection getAvailableColorsForItems() throws RemoteException, FinderException;
```

```
    /**
     *  Returns a collection of UnitOfMeasureDTOs representing all the available sizes for items.
     *
     */
    Collection getAvailableUnitOfMeasuresForItems() throws RemoteException, FinderException;

    /**
     *  Returns a collection of SizesDTOs representing all the available sizes for items.
     *
     */
    Collection getAvailableSizesForItems() throws RemoteException, FinderException;

    /**
     *  Returns a collection of StylesDTOs representing all the available styles for items.
     *
     */
    Collection getAvailableStylesForItems() throws RemoteException, FinderException;

    /**
     *  Returns a collection of LocationDTOs representing all the available locations for items at a
given store.
     *
     */
    Collection getAvailableLocationsForItems(String storeID) throws RemoteException, FinderException;

    /**
     * Finds a collection of MerchandiseClassificationDTOs that an item can belong to.
     * @return
     * @throws RemoteException
     * @throws FinderException
     */
    Collection getMerchandiseClassifications() throws RemoteException, FinderException;

    /**
     *  Keys used for importing items.
     *
     *  currently maps to the ItemFileTableDef fields
     */
    String ITEM_ADD     = "ADD";
    String ITEM_DELETE  = "DEL";
    String ITEM_UPDATE  = "CHG";
}
```

Extending This Service

You could extend this service to add item information not carried by the default service. You could change it to delegate to a merchandising system for item classification and/or item information. Either of these changes could be made by replacing the default service with a new one which adds the new material and references the default service for the rest of its data.

Dependencies

Party Service.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office. Central Office does not make use of as many of the available functions as Back Office does; currently, only Back Office can update and add items. However, Central Office can import items.

Parameter Service

Stores application configuration data and provides methods for creating and distributing that data to store systems.

Database Tables Used

- PARAMETER (Parameter)
- PARAMETER_SET (Parameter Set)
- PARM_EDITOR (Parameter Editor)
- PARM_GROUP (Parameter Group)
- PARM_SET_PARM (Parameter Set Member)
- PARM_SET_TYPE (Parameter Set Type)
- PARM_TYPE (Parameter Type)
- PARM_VAL_PROP (Parameter Possible Values)
- PARM_VALIDATOR (Parameter Validator)
- PARM_VALUE (Parameter Value)
- VAL_PROP_NAME (Validator Property Name)
- VAL_TYPE (Validator Type)

Interfaces

Methods for the service can be found in `ParameterServiceIfc.java`:

Code Sample 5-9 ParameterServiceIfc.java: Sample Methods

```
ParameterSetDTO getMasterSet() throws RemoteException;

    ParameterSetDTO getMasterSet(String group) throws RemoteException;

    Set getDistributionSets() throws RemoteException;

    ParameterSetDTO getParameterSet(int id) throws RemoteException;

    ParameterSetDTO getParameterSet(int id, boolean retrieveParameters) throws RemoteException;

    void deleteParameterSet(int id) throws RemoteException;
```

```
ParameterIfc getApplicationParameter(String param, String defaultValue) throws RemoteException;

ParameterIfc getApplicationParameter(String param, List defaultValues) throws RemoteException;
```

Extending This Service

Parameters can be added or removed without changing the Parameter Service, by importing a new master set of parameters.

Dependencies

None.

Tier Relationships

When used in Back Office, this service distributes parameters to Back Office itself and to Point-of-Sale. When used in Central Office, the service distributes parameters to Central Office, Back Office, and Point-of-Sale.

Party Service

The Party Service collects shared party data like addresses, phone number and other contact information. Parties are any person or entity that is a party to a transaction, such as an employee, store, or vendor.

Database Tables Used

- LO_ADS (Address)
- PA_CNCT (Contact)

Interfaces

The Party Service has no explicit interface file; it is a collection of entities, such as Address and Contact.

Extending This Service

This service could be extended to connect to a third-party contact database to collect its data.

Dependencies

None.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

POSlog Import Service

Imports POSlog-formatted XML into the database.

Database Tables Used

Table 5-1 POSLog Import Service Database Tables

- AS_DRW_WS (Workstation Drawer)
- AS_ITM_UNK (Unknown Item)
- AS_LY (Layaway)
- AS_TL (Till)
- AS_WS (Workstation
- CA_DY_BSN (Business Day)
- CA_PRD_RP (Reporting Period
- CO_MDFR_CMN (Commission Modifier)
- CO_MDFR_RTL_PRC Retail Price Modifier)
- CO_MDFR_SLS_RTN_TX (Sale Return Tax Modifier)
- CO_MDFR_TX_EXM Tax Exemption Modifier)
- DO_CNT_PHY (Physical Count Document)
- DO_CR_STR (Store Credit)
- DO_CRD_GF (Gift Card)
- LE_HST_STR (Store History)
- LE_HST_STR_SF_TND (Store Safe Tender History)
- LE_HST_STR_TND (Store Safe Tender)
- LE_HST_TL (Till History)
- LE_HST_TL_TND (Till Tender History)
- LE_HST_WS (Workstation History)
- LE_HST_WS_TND (Workstation Tender History)
- LE_LTM_MD_TND (Tender Media Line Item)
- LO_ADS (Address)
- LO_EML_ADS (E-mail Address)
- OR_LTM (Order Line Item)
- OR_LTM_MDFR_RPRC (Order Line Item Retail Price Modifier)
- OR_ORD (Order)
- ORGN_CT (Business Customer)
- PA_CNCT (Contact)
- PA_CT (Customer)
- PA_ID_PRTY_GEN (Party ID Generation)
- PA_PHN (Phone Number)
- PA_PRTY (Party)
- TR_ADS_SLS_RTN (Sale Return Line Item Address)
- TR_CNT_INV (Inventory Count Transaction)
- TR_CTL (Control Transaction)
- TR_FN_ACNT (Financial Accounting Transaction)
- TR_ITM_CPN_TND (Coupon upon Tender Line Item)
- TR_LON_TND (Tender Lone Transaction)
- TR_LTM_ALTR (Alteration Line Item)
- TR_LTM_CHK_TND (Check Tender Line Item)
- TR_LTM_CR_STR_TND (Store Credit Line Item)
- TR_LTM_CRDB_CRD_TN (Credit Debit Tender Line Item)
- TR_LTM_DSC (Discount Line Item)
- TR_LTM_GF_CF_TND (Gift Certification Tender Line Item)
- TR_LTM_GF_CRD_TND (Gift Card Tender Line Item)
- TR_LTM_PHY_CNT Physical Count Line Item)
- TR_LTM_PRCH_ORD_TND (Purchase Order Tender Line Item)
- TR_LTM_PYAN (Payment On Account Line Item)
- TR_LTM_RTL_TRN Retail Transaction Line Item)
- TR_LTM_SLS_RTN (Sale Return Line Item)
- TR_LTM_SLS_RTN_TX (Sale Return Tax Line Item)
- TR_LTM_SND_CHK_TND (Send Check Tender Line Item)
- TR_LTM_TND (Tender Line Item)
- TR_LTM_TND_CHG Tender Change Line Item)
- TR_LTM_TRV_CHK_TND (Travelers Check Tender Line Item)
- TR_LTM_TX (Tax Line Item)
- TR_PKP_TND (Tender Pickup Transaction)
- TR_RCV_FND (Funds Receipt Transaction)
- TR_RTL (Retail Transaction)
- TR_SLS_PS_NO (POS No Sale Transaction)
- TR_STR_OPN_CL (Store Open Close Transaction)
- TR_TL_OPN_CL (Till Open Close Transaction)
- TR_TRN (Transaction)
- TR_VD_PST (Post Void Transaction)
- TR_WS_OPN_CL (Workstation Open Close Transaction)

Interfaces

The functions of the POSlog Import Service are encapsulated within the Transaction Service; if you need to call for a POSlog Import, do it through the Transaction Service.

Extending This Service

You could extend this service to capture additional custom POSLog elements which are not part of the base ARTS IXRetail XML standard.

Dependencies

None.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Post-Processor Service

The Post-Processor Service provides a service interface for processing transactional data after it is received and storing the information in summary tables. Post-processing serves as a performance enhancement for reports.

Database Tables Used

- LE_SMY_CT (CustomerSummary)
- LE_SMY_CT_DMOG (CustomerDemographicSummary)
- LE_SMY_MRHRC_EM (EmployeeMerchandiseSummary)
- LE_SMY_PS_DPT_EM (EmployeePosDepartmentSummary)
- LE_SMY_ITM_SLS (ItemSalesSummary)
- LE_SMY_MRHRC_SLS (MerchandiseHierarchySalesSummary)
- LE_SMY_OPR (OperatorSummary)
- LE_SMY_PS_DPT (POSDepartmentSummary)
- LE_SMY_PRDV (PriceDerivationRuleSummary)
- LE_SMY_OPR_TMACV (OperatorTimeActivitySummary)
- LE_SMY_WS_TMACV (WorkstationTimeActivitySummary)
- LE_SMY_WS (WorkstationSummary)

Interfaces

This service offers an extremely simple interface: there is only one method, `processTransactions()`. Access this service through `PostProcessorServiceIfc.java`:

Code Sample 5-10 PostProcessorServiceIfc.java: Some Methods

```
public interface PostProcessorServiceIfc
{
    void processTransactions() throws RemoteException;

}
```

Extending This Service

This service is designed to support a variety of post-processors, which can be created as separate objects. You can extend this service by adding additional post-processors.

Dependencies

- Calendar Service
- Financial Totals Service
- Transaction Service

Tier Relationships

This functionality is present in both Back Office and Central Office. However, because there are currently no reports in Central Office which rely on the Post Processors, this service is disabled in Central Office.

Pricing Service

The Pricing service offers functions for requesting pricing rules, modifying them, and creating new ones.

Database Tables Used

- MA_PRC_ITM (ItemPriceMaintenance)
- MA_ITM_PRN_PRC_ITM (PermanentPriceChangeItem)
- TR_CHN_PRN_PRC (PermanentPriceChange)
- MA_ITM_TMP_PRC_CHN (TemporaryPriceChangeItem)
- TR_CHN_TMP_PRC (TemporaryPriceChange)
- CO_EL_PRDV_DPT (DepartmentPriceDerivationRuleEligibility)

- CO_EL_PRDV_ITM (ItemPriceDerivationRuleEligibility)
- CO_PRDV_ITM (ItemPriceDerivation)
- CO_EL_MRST_PRDV (MerchandiseStructurePriceDerivationRuleEligibility)
- TR_ITM_MXMH_PRDV (MixAndMatchPriceDerivationItem)
- RU_PRDV (PriceDerivationRule)
- RU_TY_PRDV (PriceDerivationRuleType)
- CO_EV (Event)
- CO_MNT_ITM (ItemMaintenanceEvent)
- CO_EV_MNT (MaintenanceEvent)
- AS_ITM_RTL_STR (Retail Store Item)

Interfaces

Access this interface through `PricingServiceIfc.java`. The following code sample shows a few of the available methods:

Code Sample 5-11 PricingServiceIfc.java: Some Methods

```
{
    void importPricing(String content) throws RemoteException;

    AdvancedPricingRuleDTO createAdvancedPricingRule(String storeID,
                                                                    int
priceDerivationRuleTypeId, String name,
                                                                            Date effectiveDate,
Date expirationDate,
                                                                            ComparisonBasis
sourceBasis, ComparisonBasis targetBasis) throws RemoteException, AdvancedPricingRuleException;

    AdvancedPricingRuleDTO getAdvancedPricingRule(String store, int id) throws RemoteException,
AdvancedPricingRuleException;

    void removeAdvancedPricingRule(String store, int id) throws RemoteException,
AdvancedPricingRuleException;

    Collection getAllPricingRuleTypesForStore(String storeID) throws RemoteException;

    Collection findAdvancedPricingRules(AdvancedPricingRuleSearchCriteria criteria) throws
RemoteException, AdvancedPricingRuleSearchException;

    void endAdvancedPricingRule(String storeId, int id) throws RemoteException,
AdvancedPricingRuleException;

    Collection findPricingPromotions(PricingPromotionSearchCriteria pricingPromotionSearchCriteria)
throws RemoteException, PricingPromotionNotFoundException;
    PricingChangeDTO updateTemporaryPriceChange(PricingChangeDTO pricingChangeDTO) throws
RemoteException,PricingChangeException;
    java.util.HashMap findPricingPromotion(String promotionId, String storeId) throws RemoteException,
PricingPromotionNotFoundException;

    PricingPromotionSearchCriteria getItemDetails(String eventId, String itemId, String storeId) throws
RemoteException,ItemNotFoundException, ItemIneligibleException;

    Collection findPricingChanges(PricingChangeSearchCriteria pricingChangeSearchCriteria) throws
RemoteException, PricingChangeSearchException, PricingChangeException;
```

```
    void addSourceToAdvancedPricingRule(int pricingRuleID, String storeID, String sourceId, BigDecimal
comparisonValue) throws RemoteException, AdvancedPricingRuleException;

    void addTargetToAdvancedPricingRule(int pricingRuleID, String storeID, String targetId, BigDecimal
reduction, int limitCount) throws RemoteException, AdvancedPricingRuleException;

    void removeSourceTargetFromAdvancedPricingRule(int pricingRuleID, String storeID, String[]
sourceIds, String[] targetIds) throws RemoteException, AdvancedPricingRuleException;

    void removeTargetFromAdvancedPricingRule(int pricingRuleID, String storeID, String targetId) throws
RemoteException, AdvancedPricingRuleException;

    void removeSourceFromAdvancedPricingRule(int pricingRuleID, String storeID, String sourceId) throws
RemoteException, AdvancedPricingRuleException;
```

Extending This Service

You could extend this service to add additional pricing functions or to draw data from a different source, such as a marketing database that tracks upcoming price promotions.

Dependencies

- Item Service
- Workflow Service

Tier Relationships

When used in Back Office, all of the Pricing functionality is available. When used in Central Office, Import pricing is not available.

Reporting Service

The Reporting Service is a framework for creating and exporting reports, managing users' favorite reports, and maintaining collections for scheduling. The service supports XSL and Crystal reports. The service uses the i-net Crystal Clear EJB to process report definitions created with Seagate Software's Crystal Reports.

Export formats include HTML, CSV, PDF and TXT.

Database Tables Used

- EXECUTED_REPORT (Executed Report)
- FAVORITE_REPORT (Favorite Report)
- REPORT_CONFIG (Report Configuration)
- REPORT_CONFIG_PARAMETER (Report Configuration Parameters)

- REPORT_CRITERIA (Report Criteria)
- REPORT_GROUP (Report Group)
- REPORT_RECIPIENT (Report Recipient)

Interfaces

The Reporting Service includes methods for report creation and report type. It is contained within `ReportingServiceIfc.java`:

Code Sample 5-12 ReportingServiceIfc.java: Methods

```
String createReport(Handle handle, ReportCriteriaIfc reportCriteria);
String getType();
```

Extending This Service

The Reporting Service can be easily extended to support new XSL and Crystal Reports. Report definitions are database driven. The report definitions contain a name, report implementation (Java class or Crystal Report template), report parameters, and report types.

Dependencies

- Workflow/Scheduling Service
- Store Service
- Store Ops Service
- Calendar Service

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Store Directory Service

This service provides access to the directory of stores in the enterprise.

Database Tables Used

- CO_STRGP_FNC (RetailStoreGroupFunction)
- ST_ASCTN_STRGP_STR (AssociatedRetailStoreStoreGroup)
- CO_STRGP_LV (RetailStoreGroupLevel)
- CO_STRGP (RetailStoreGroup)

- ST_ASCTN_STRGP (AssociatedRetailStoreGroup)

Interfaces

Use `StoreDirectoryIfc.java`, which offers more than 20 methods. These include methods for getting paths to stores, the current store's node in the hierarchy, or a set of stores based on some set of search criteria.

Code Sample 5-13 StoreDirectoryIfc.java: Some Methods

```
    /**
     * Get all of the store hierarchies on the system.  This is just the store hierarchy alone, no
     * individual store(s) underneath of group node.
     * @return HierarchyNodeIfc if any exists, else null
     * @throws RemoteException
     */
    HierarchyNodeIfc getStoreHierarchies() throws RemoteException;


    /**
     * Return the hierarchy node that the store belongs to.  This hierarchy node will also have a list
of
     * ancestors of the store represented by storeId.
     * @param storeId
     * @return HierarchyNodeIfc or null if the store does not belong to any store hierarchy
     * @throws RemoteException
     * @throws FinderException
     */
    HierarchyNodeIfc getStoreItsStoreHierarchy(String storeId) throws RemoteException, FinderException;


    StoreDTO getStore(String storeID) throws RemoteException, FinderException;

    /**
     * gets all the stores for a given selection criteria
     *
     * @return a ArrayList of string store ids
     */
    Collection getStores(StoreSelectionCriteria criteria) throws RemoteException;

    Collection getStores(StoreSelectionCriteria criteria, boolean returnEmpty) throws RemoteException;

    String getGroupName(HierarchyNodeKey key) throws RemoteException;

    /**
     *
```

Extending This Service

You could replace this service with a connection to an existing database of stores, if your enterprise already maintains this information in another form.

Dependencies

Parameter Service.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Store Service

Look up and maintain store attributes, store hierarchy information and store history.

Database Tables Used

- CA_DY_BSN (Business Day)
- CA_PRD_RP (Reporting Period)
- CO_STRGP (Store Group)
- CO_STRGP_FNC (Store Group Function)
- CO_STRGP_LV (Store Group Level)
- EMPLOYEE_HIERARCHY_ASSN (Employee Hierarchy Association)
- LE_HST_STR (Store History)
- LE_HST_STR_SF_TND (Store Safe Tender History)
- LE_TND_STR_SF (Store Safe Tender)
- PA_STR_RT (Retail Store)
- ST_ASCTN_STRGP (Associated Retail Store Group)
- ST_ASCTN_STRGP_STR (Associated Retail Store Group Store)

Interfaces

Use `StoreServiceIfc.java`, which provides one method:

Code Sample 5-14 StoreServiceIfc.java

```
public interface StoreServiceIfc
{
    /**
     * Returns list of WorkstationDTOs
     * @param storeId
     * @return
     * @throws RemoteException
     */
    List getAllWorkstations(String storeId) throws RemoteException;
}
```

Extending This Service

If your enterprise needs additional store information not carried by the default service, you could extend this service to include the new data.

Dependencies

Parameter Service.

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office.

Store Ops Service

This service provides functions for opening and closing the store, as well as other store operations.

Database Tables Used

The service depends on other services for its data and does not access persistent storage directly.

Interfaces

Use `StoreOpsServiceIfc.java`, which includes methods for opening and closing the store plus querying whether the store is currently open, opening and closing a specific workstation, and handling tills:

Code Sample 5-15 StoreOpsServiceIfc.java: Some Methods

```
public interface StoreOpsServiceIfc
{
    /**
     * @param storeId the store id
     * @return a store status dto
     * @throws StoreStatusNotFoundException
     * @throws RemoteException
     */
    public StoreStatusDTO getStoreStatus(String storeId) throws StoreStatusNotFoundException,
RemoteException;

    /**
     * This method encapsulates all of the business logic for opening a
     * workstation, a.k.a register. Here is the list of the operations that take
     * place to open a workstation:
     *  1.update the workstation status
     *  2.create a Register Open transaction
     *  3.create a workstation history record
         *
     * @param storeID
     * @param workstationID
     * @throws RemoteException
```

```
     */
    public void openWorkstation(String storeID, String workstationID, Date businessDay) throws
RemoteException;

    /**
     * This method encapsulates all of the business logic for closing a
     * workstation, a.k.a register. Here is the list of the operations that take
     * place to open a workstation:
     *  1.update the workstation status
     *  2.create a Register Close transaction
     *  3.create a workstation history record
         *
     * @param storeID
     * @param workstationID
     * @throws RemoteException
     */
    public Hashtable closeWorkstation(String storeID, String workstationID, Date businessDay) throws
RemoteException;


    /**
     * @param storeID
     * @param businessDay
     * @param openOperatingFundsBalance
     * @throws CurrencyCreationException
     * @throws CurrencyTypeNotFoundException
     * @throws RemoteException
     */
    public void openStore(String storeID, Date date, BigDecimal openOperatingFundBalance)
               throws CurrencyCreationException, CurrencyTypeNotFoundException, RemoteException;

    /**
     * @param storeID
     * @param businessDay
     * @param openOperatingFundsBalance
     * @throws CurrencyCreationException
     * @throws CurrencyTypeNotFoundException
     * @throws RemoteException
     */
    public Hashtable closeStore(String storeID, Date date, BigDecimal closeOperatingFundBalance)
               throws CurrencyCreationException, CurrencyTypeNotFoundException, RemoteException;


    /**
     * This is the service method to open a store. Specifically, the following steps
     * happen in opening a store:
     * 1. create a store open transaction.
     * 2. update the safe
     * 3. create tender media lineitem
     * 4. update store history
     *
     * @param storeID
     * @param businessDay
     * @param openOperatingFundsBalance
     * @throws CurrencyCreationException
     * @throws CurrencyTypeNotFoundException
     * @throws RemoteException
     */
       public void openStore(String storeID, Date businessDay, CurrencyDTO openOperatingFundsBalance)
        throws RemoteException;

    /**
     * This is the service method to close a store.
     *
```

```
     * @param storeID
     * @param businessDay
     * @param closeOperatingFundsBalance
     * @throws CurrencyCreationException
     * @throws CurrencyTypeNotFoundException
     * @throws RemoteException
     */
       public Hashtable closeStore(String storeID, Date businessDay, CurrencyDTO
closeOperatingFundsBalance)
        throws RemoteException;
```

Extending This Service

You could extend or modify this service to change how stores are opened, closed, or reconciled.

Dependencies

Parameter Service.

Tier Relationships

This service is used only in Back Office.

Tax Service

The Tax service allows you to import tax information from a tax file.

Database Tables Used

- CO_GP_TX_ITM (TaxableGroup)
- PA_ATHY_TX_PSTL (TaxAuthorityPostalCode(Deprecate))
- PA_ATHY_TX (TaxAuthority)
- RU_TX_GP (TaxGroupRule)
- RU_TX_RT (TaxGroupRule)

Interfaces

Access this interface through `TaxServiceIfc.java`. There is only one method, `importTaxFile()`:

Code Sample 5-16 Ifc.java: Some Methods

```
public interface TaxServiceIfc {
    public void importTaxFile(String content) throws RemoteException, TaxAuthorityException,
TaxableGroupException, TaxRuleException;
}
```

Extending This Service

You could replace this service with one to import tax information from a different source. If you want this service to perform other functions when importing a tax file, you could wrap this service with one of your own creation, calling this service to perform the tax file import.

Dependencies

Party Service.

Tier Relationships

This service is used only in Back Office.

Time Maintenance Service

The Time Maintenance Service provides an interface to functions which manage employee work time data. This includes clock in/out, editing, creating, and confirming employee time.

Database Tables Used

- ADT_LOG (AuditLog)
- CO_CONF_EM_TM_ENR (EmployeeConfirmedTimeEntry)
- CO_EM_TM_ENR (EmployeeTimeEntry)
- CA_WRK_WK (WorkWeekConfirm)

Interfaces

Access this interface through `TimeMaintenanceServiceIfc.java`. The following code sample shows a few of the available methods:

Code Sample 5-17 TimeMaintenanceServiceIfc.java: Some Methods

```
public interface TimeMaintenanceServiceIfc
{
    /**
     * Returns an EmployeeTimeEntryDTO for the passed in user that represents the last time entry
     * the employeeID made.
     *
     * @param employeeID
     * @return
     * @throws RemoteException
     * @throws CreateException
     */
    EmployeeTimeEntryDTO getLastEmployeeTimeEntryForEmployee(String employeeID) throws RemoteException,
FinderException;
```

```
    /**
     * Adds either an IN or OUT entry associated with a timestamp for the current user.
     *
     * @param employeeTimeEntryDTO
     * @return
     * @throws RemoteException
     * @throws CreateException
     */
    void addTimeEntry(EmployeeTimeEntryDTO employeeTimeEntryDTO) throws RemoteException,
CreateException;

    /**
     * Check whether the TimeEntries for the week for the period of startOfWeek - endOfWeek are
complete.
     * Returns true if there are no unmatched entries for the week.
     *
     * @param retailStoreId
     * @param startOfWeek
     * @param endOfWeek
     * @return
     * @throws RemoteException
     * @throws TimeMaintenanceException
     */
    Boolean checkComplete(String retailStoreId, Week week) throws RemoteException,
TimeMaintenanceException;

    /**
     * Check whether the time entries for the week have been confirmed.
     * Return true if the time entries for the week have been confirmed.
     *
     * @param retailStoreId
     * @param startOfWeek
     * @param endOfWeek
     * @return
     * @throws RemoteException
     * @throws TimeMaintenanceException
     */
    Boolean checkConfirmed(String retailStoreId, Week week) throws RemoteException,
TimeMaintenanceException;

    /**
     * Perform a Confirm of the week's time maintenance.
     *
     * @param retailStoreId
     * @param week
     * @throws RemoteException
     * @throws TimeMaintenanceException
     * @throws ConfirmException
     */
    void  confirm(String retailStoreId, Week week) throws RemoteException, TimeMaintenanceException,
ConfirmException;

    /**
     * Perform validation of whether time maintenance can be confirmed on the given week for the store.
     * Prior to validation matchTimeEntries() is called to sweep in any new time entries. Then
validation is performed
     * and if any confirm rules are violated a corresponding exception is thrown.
     *
     * @param retailStoreId
     * @param week
     * @throws RemoteException
     * @throws TimeMaintenanceException
     * @throws ConfirmException
     */
```

```
    void validateConfirm(String retailStoreId, Week week) throws RemoteException,
TimeMaintenanceException, ConfirmException;

    /**
     * Retrieve an EmployeeTimeSummaryDTO for an employee and a date range.
     *
     * @param employeeId
     * @param retailStoreId
     * @param dateRange
     * @return
     * @throws RemoteException
     * @throws TimeMaintenanceException
     */
    EmployeeTimeSummaryDTO getEmployeeTimeSummary(String employeeId, String retailStoreId, DayRange
dateRange) throws RemoteException, TimeMaintenanceException;

    /**
     * Retrieve the EmployeeTimeSummaryDTO array for mulptiple employee and a date range.
     *
     * @param employeeIds
     * @param retailStoreId
     * @param dateRange
     * @return
     * @throws RemoteException
     * @throws TimeMaintenanceException
     */
    EmployeeTimeSummaryDTO[] getEmployeeTimeSummaries(String[] employeeIds, String retailStoreId,
DayRange dateRange) throws RemoteException, TimeMaintenanceException;

    /**
     * Make edits to employeeTimeEntries based on the EmployeeDailyHoursDTO parameter.
     * This may include add, mark deleted or edit
     *
     * @param hours
     * @param retailStoreId
     * @param week
     * @throws RemoteException
     * @throws TimeMaintenanceException
     */
    void editEmployeeHours(EmployeeDailyHoursDTO hours, String retailStoreId, Week week) throws
RemoteException, TimeMaintenanceException;

    /**
     * Return the next sequential id to use for creation of EmployeeConfirmedTimeEntry
     * @return
     * @throws RemoteException
     */
    String getNextEmployeeConfirmedTimeEntryId() throws RemoteException;
}
```

Extending This Service

You could extend this service to add additional time maintenance functions, or to connect to a different application (other than Back Office) to supply time tracking information.

Dependencies

Calendar Service.

Tier Relationships

This service is expected to be used only with Back Office, although its functionality would work from either Back Office or Central Office.

Transaction Service

Searches for transactions and E-journals based on transaction, customer or item information.

Database Tables Used

- AS_ITM (Item)
- AS_ITM_STK (Stock Item)
- CO_MDFR_CMN (Commission Modifier)
- CO_MDFR_RTL_PRC (Retail Price Modifier)
- ID_IDN_PS (POS Identity)
- JL_ENR (Journal Entry)
- LE_LTM_MD_TND (Tender Media Line Item)
- LO_ADS (Address)
- PA_CNCT (Contact)
- PA_CT (Customer)
- TR_CTL (Control Transaction)
- TR_LON_TND (Tender Lone Transaction)
- TR_LTM_CHK_TND (Check Tender Line Item)
- TR_LTM_CRDB_CRD_TN (Credit Debit Tender Line Item)
- TR_LTM_RTL_TRN (Retail Transaction Line Item)
- TR_LTM_SLS_RTN (Sale Return Line Item)
- TR_LTM_TND (Tender Line Item)
- TR_PKP_TND (Tender Pickup Transaction)
- TR_RTL (Retail Transaction)
- TR_SLS_PS_NO (POS No Sale Transaction)
- TR_STR_OPN_CL (Store Open Close Transaction)
- TR_TL_OPN_CL (Till Open Close Transaction)
- TR_TRN (Transaction)
- TR_VD_PST (Post Void Transaction)

- TR_WS_OPN_CL (Workstation Open Close Transaction)

Interfaces

The file `TransactionServiceIfc.java` contains a number of methods for accessing the service. These include transaction creation, search result, and others.

Code Sample 5-18 TransactionServiceIfc.java: Some Sample Methods

```
/** Retrieves a transaction's type.
 *
 * @param transactionKey
 * @return TransactionType
 */
TransactionType getType(TransactionKey transactionKey)
        throws RemoteException, InvalidTypeException,
        ObjectNotFoundException;

/** Retrieves a transaction data transfer object given a TransactionKey as input.
 *
 * @param transactionKey
 * @return RetailTransactionDTO
 */
TransactionDTO retrieveTransaction(TransactionKey transactionKey)
        throws RemoteException, ObjectNotFoundException,
        InvalidTypeException;

/** Retrieves a set of ejournal information given ejournal search criteria as input.
 *
 * @param storeSelectionCriteria
 * @param ejournalCriteria
 * @param startIndex
 * @return returnLimit
 */
EJournalSearchResultDTO getEJournals(
        StoreSelectionCriteria storeSelectionCriteria,
        EJournalCriteria ejournalCriteria, int startIndex, int returnLimit)
        throws RemoteException, SearchResultSizeExceededException;
```

Extending This Service

You would need to extend this service if you wanted to add new transaction types or new ways of searching for transactions.

Dependencies

- Parameter Service
- Customer Service
- Item Service
- Store Service

Tier Relationships

The functionality of this service is the same whether it is used within Central Office or Back Office. Although full functionality is available to both applications, Central Office tends to import transactions while Back Office tends to export them, due to the intended use of the applications.

Workflow/Scheduling Service

Create and edit tasks, schedule tasks, track task approval. Even for tasks which should be scheduled immediately, the Workflow/Scheduling Service provides task tracking features.

Database Tables Used

- CO_EVT_MSG (Job Event Messages)
- FILE_SET (File Set)
- FILE_SET_ITEM (File Set Item)
- SCHEDULE (Schedule)
- TASK (Task)
- TASK_DESTINATION_STATUS (Task Destination Status)
- TASK_HISTORY (Task History)
- TASK_NOTIFICATION_RECIPIENT (Task Notification Recipient)
- TASK_REVIEW (Task Review)
- WORKFLOW_CONFIGURATION (Workflow Configuration)

Interfaces

The Workflow Service's methods are defined in `WorkflowServiceIfc.java`. They include methods for task creation, notification, task destination, and more.

Extending This Service

Extend this service by adding new task types. You would need to add the new task type to the workflow configuration table, add a map to execute the task, and add a user interface to allow the task type to be created.

Dependencies

- Parameter Service
- Store Service

Tier Relationships

This service is the same whether it is used within Central Office or Back Office; either application calls the service to schedule tasks.

CHAPTER 6

STORE DATABASE

Overview

360Store Back Office uses an ARTS-compliant database. Data is stored and retrieved by entity beans in a bean-managed persistence pattern, so the system makes database calls from the entity bean code.

A single entity bean exists for each database table, and handles reads and writes for that table. Each entity bean contains the necessary methods to create, load, store, and remove its object type.

The Back Office application writes data to the Store database, a repository for transaction information for a single store.

Related Documentation

The following related sources provide specific information about the database for your use when developing code.

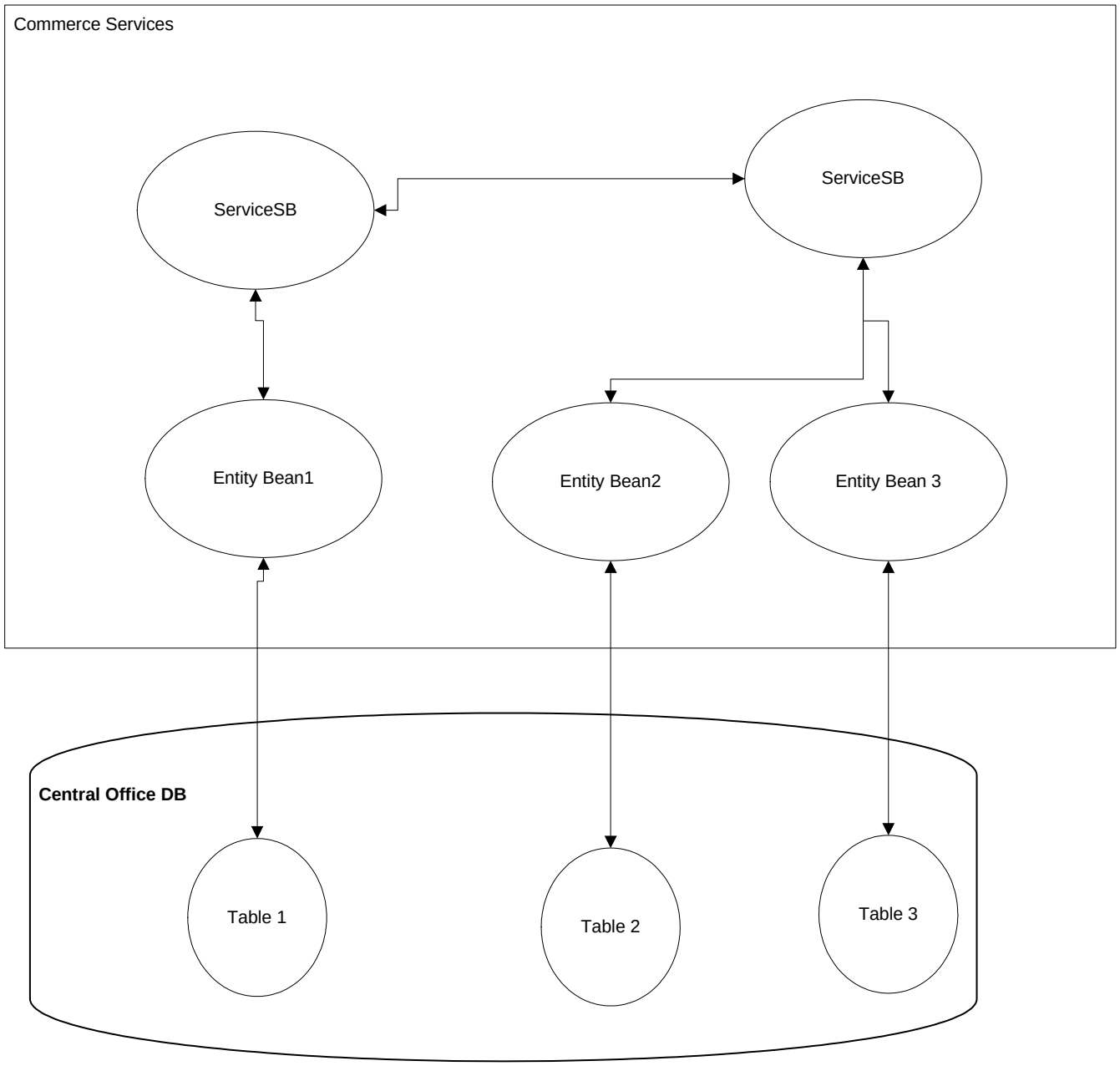
Table 6-1 Related Documentation

Source	Description
ARTS Database Standard	See http://www.nrf-arts.org/ for a description of the ARTS database standard.
Data Dictionary	Contains table and column definitions for the database used to store Back Office data. See the `_resources` directory provided with your Back Office documentation.
Database Diagrams	See the `_resources` directory for diagrams which show the relationships between various tables in the database schema.

Database/System Interface

As described in Chapter 1, “Architecture,” a persistence layer of entity beans represents the database tables to the rest of the system. One bean represents each table. Figure 6-1 illustrates these relationships.

Figure 6-1 Commerce Services, Entity Beans, and Database Tables



Each commerce service communicates with one or more entity beans, and each entity bean communicates with one database table. Although there are exceptions, in general only one commerce

service communicates with an entity bean; other services request the information from the relevant service rather than talking directly to the entity bean. For example, if the Customer Service needs information provided by the Item Bean, it makes a request to the Item Service.

ARTS Compliance

When new code is added or features are added, modified, or extended, database plans should be evaluated to ensure that new data items fit the ARTS schema. Complying with the standards increases the likelihood that extensions can migrate into the product codebase and improves code reuse and interoperability with other applications.

Note: Because the ARTS standard continues to evolve, older code may contain deviations from the standard or may be compliant only with an earlier version of the standard. 360Commerce continues to evaluate ARTS compliance with each release of its software.

Bean-managed Persistence in the Database

In general, the system uses standard J2EE bean-managed persistence techniques to persist data to the 360Store database. Each of the entity beans that stores data requires JDBC code in standard ejbLoad, ejbStore, ejbCreate, and ejbRemove classes. However, there are some differences worth noting:

- All SQL references are handled as constant fields in an interface.
- Session and entity beans extend an EnterpriseBeanAdapter class. Special extensions for session and entity beans exist. These contain common code for logging and a reference to the 360Commerce DBUtils class (which provides facilities for opening and closing data source connections, among other resources).

Code Sample 6-1 ItemPriceDerivationBean.java: ejbStore Method

```
public void ejbStore() throws EJBException
    {
        ItemPriceDerivationPK key = (ItemPriceDerivationPK) getEntityContext().getPrimaryKey();
        getLogger().debug("store");
        PreparedStatement ps = null;
        Connection conn = null;
        if (isModified())
        {
            getLogger().debug("isModified");
            try
            {
                conn = getDBUtils().getConnection();
                ps = conn.prepareStatement(ItemPriceDerivationSQLIfc.STORE_SQL);
                int n = 1;
                ps.setBigDecimal(n++,getReductionAmount().toBigDecimal());
                ps.setBigDecimal(n++,getDiscountPricePoint().toBigDecimal());
                getDBUtils().preparedStatementSetDate(ps, n++, getRecordCreationTimestamp());
                ps.setBigDecimal(n++,getReductionPercent().toBigDecimal());
                getDBUtils().preparedStatementSetDate(ps, n++, getRecordLastModifiedTimestamp());
                ps.setInt(n++, key.getPriceDerivationRuleID());
```

```
                ps.setString(n++, key.getStoreID());
                if (ps.executeUpdate() != 1)
                {
                    throw new EJBException("Error storing (" + getEntityContext().getPrimaryKey() +
")");
                }
                setModified(false);
            }
            catch (SQLException ex)
            {
                getLogger().error(ex);
                throw new EJBException(ex);
            }
            catch (Exception ex)
            {
                getLogger().error(ex);
                throw new EJBException(ex);
            }
            finally
            {
                getDBUtils().close(conn, ps, null);
            }
        }
    }
```

CHAPTER 7

EXTENSION GUIDELINES

Overview

This document 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—a Struts/Tiles implementation utilizing Actions for processing UI requests and JSP pages with Tiles providing layout.
- Application Manager—a session facade for the UI (or external system) that models application business methods. May or may not be reusable between applications. 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 consumed by the service. The entities are local to the service that controls them.

Audience

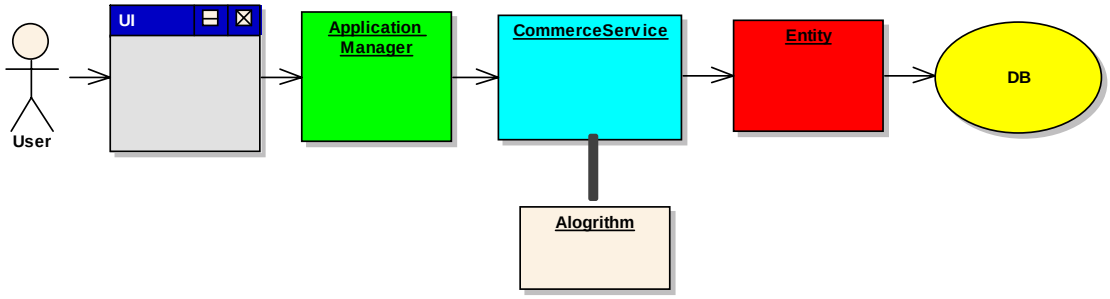
This chapter provides guidelines for extending the 360Commerce Enterprise applications. The guidelines are designed for three audiences:

- Members of customer architecture and design groups can use this document as the basis for their analysis of the overall extension of the systems.
- Members of 360commerce's Technology and Architecture Group can use this document as the basis for analyzing the viability of the overall extension strategy for enterprise applications.
- Developers on the project teams can use this document as a reference for code-level design and extension of the product for the solution that is released.

Application Layers

The following diagram describes the general composition of the enterprise applications. The sections following describe the purpose and responsibility of each layer.

Figure 7-1 Application Layers



UI

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

- Struts configuration
- Tiles definition
- Style sheets (CSS)
- JSP pages
- Resource bundles for 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 (DTOs)

Algorithm

An SPI-like interface defined to allow more fine grained pieces of business functionality to be replaced without impacting overall application logic. For reference, review the various POJO “calculator” classes that are contained in the Financial Totals Commerce Service.

Entity

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

DB

The 360Commerce 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-Sale.

Extension and Customization Scenarios

Style and Appearance Changes

This should only present minor changes to the UI layer of the application. These type 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

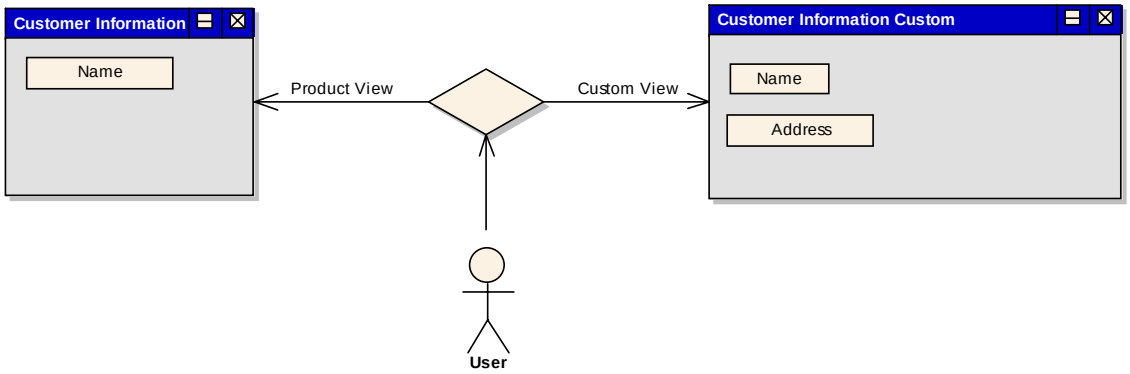
This is one of the more common extensions to the base product: allowing for the full life cycle management of information required by a particular customer that is not represented in the base product.

If the information is simply presented and persisted then we can choose a strategy that simply updates the UI and Persistent layer 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 7-2 Managing Additional Information



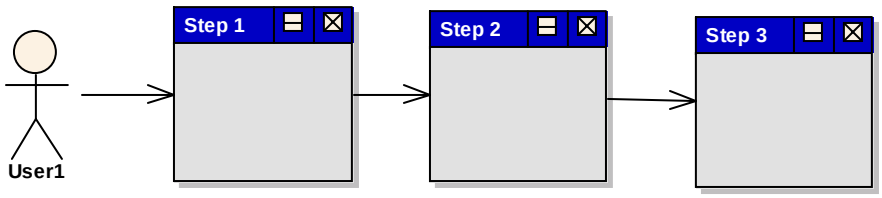
Application impact:

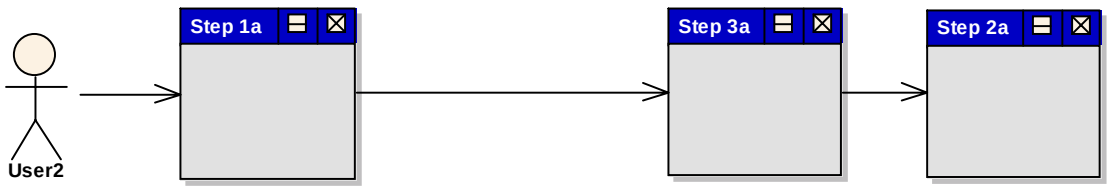
- JSP pages
- View Beans
- Struts configuration
- UI Actions
- UI Forms
- Application Manager
- Commerce Service
- Entity
- Database Schema

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 7-3 Changing Application Flow





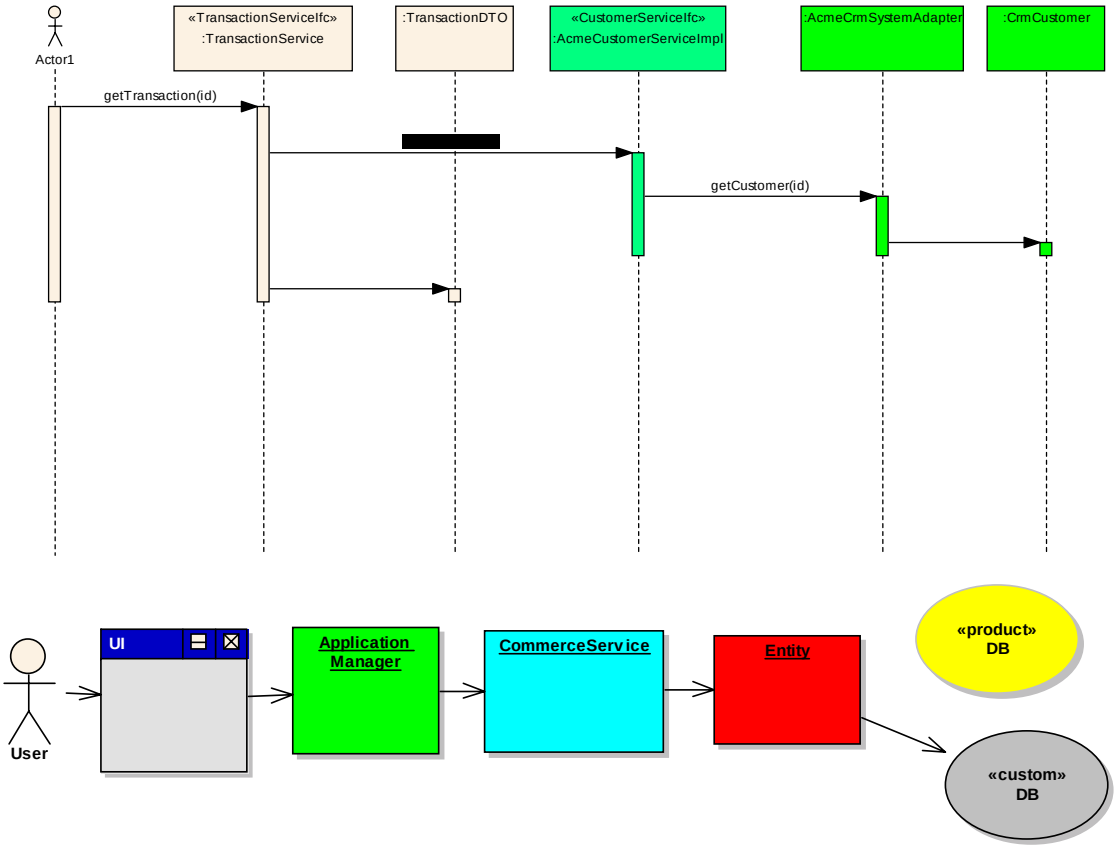
Application impact:

- 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 7-4 Accessing Data from a Different Database



Application impact:

- Entity Beans
- 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 7-5 Accessing Data from an External System

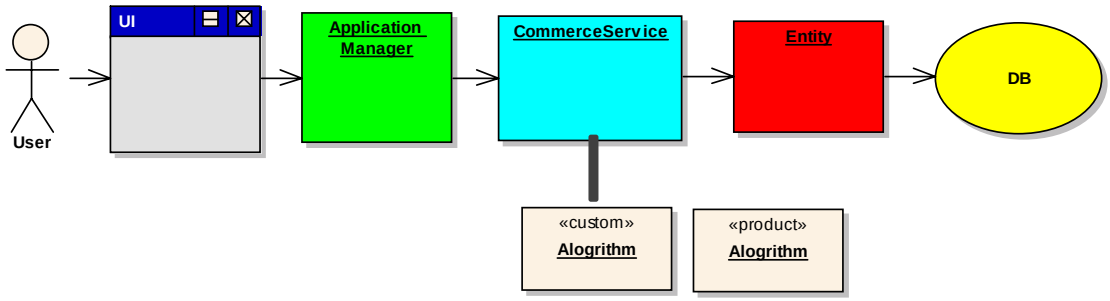
Application impact:

- 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 7-6 Application Layers



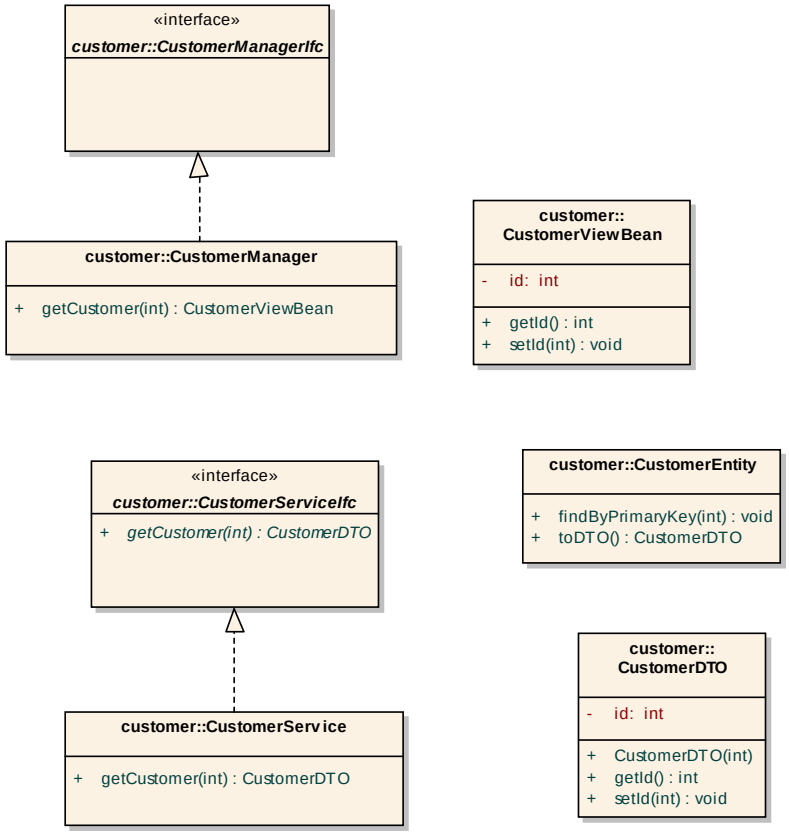
Application impact:

- Algorithm
- Application Configuration

Extension Strategies

Refer to the following diagram as a subset of classes for comparison purposes.

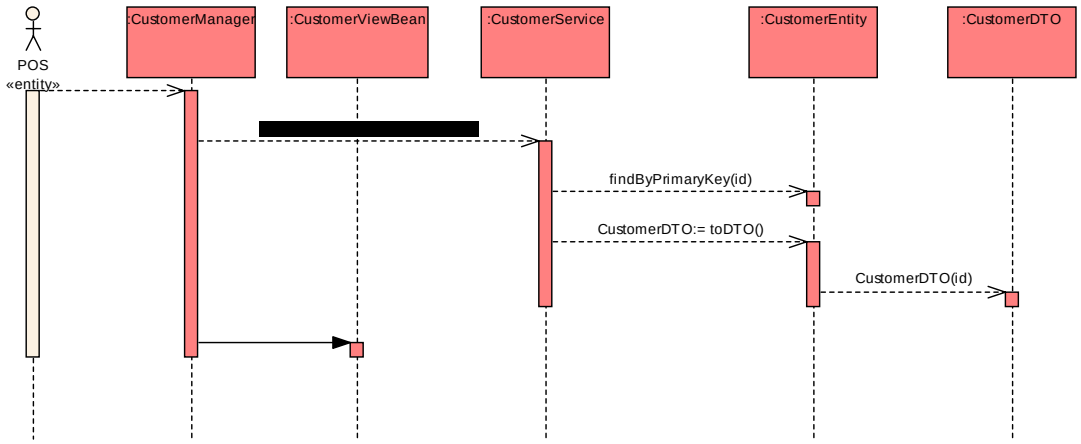
Figure 7-7 Sample Classes for Extension



Extension with Inheritance

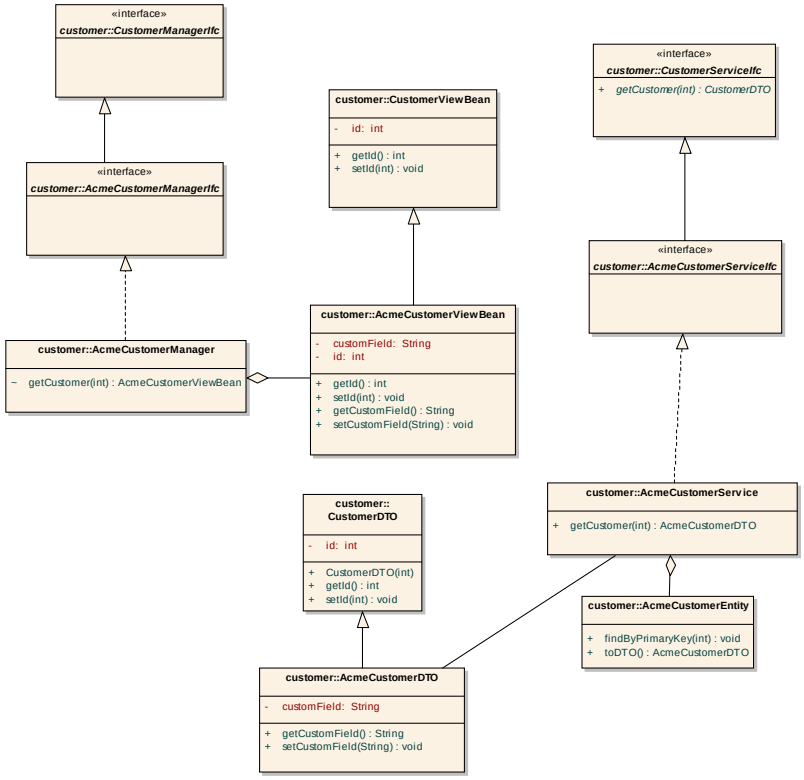
This strategy involves changing the interfaces of the service itself, perhaps to include a new finder strategy or data items unique to a particular implementation. For instance, if the customer information contained in base product does not contain data relevant to the implementation, call it CustomField1.

Figure 7-8 Extension with Inheritance



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 7-9 Extension with Inheritance: Class Diagram



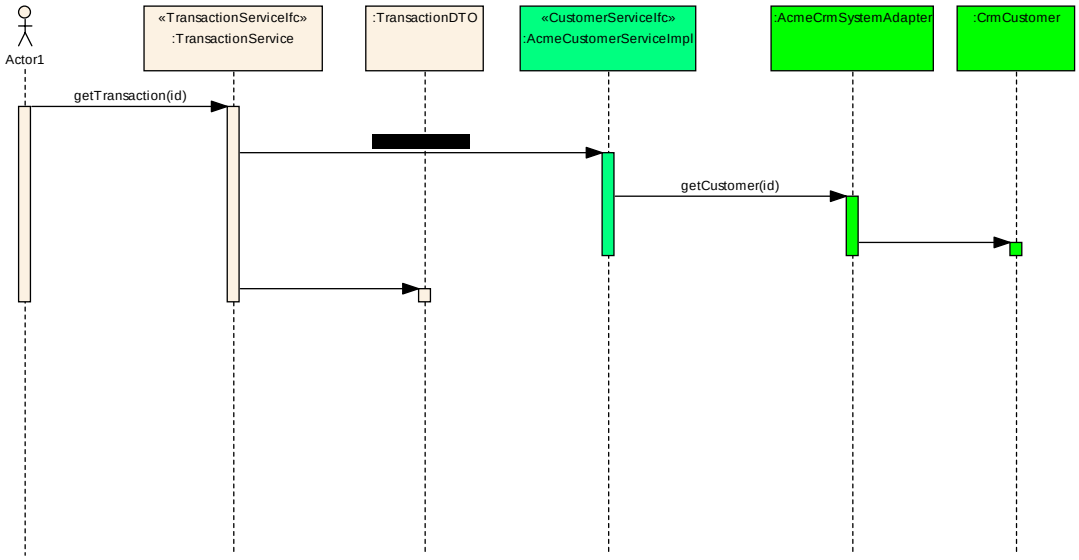
Replacement of Implementation

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.

The following diagram demonstrates the replacement of the product Customer Service implementation with an adapter that delegates to an existing CRM solution for that is 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 7-10 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 it 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 7-11 Extension with Composition: Class Diagram

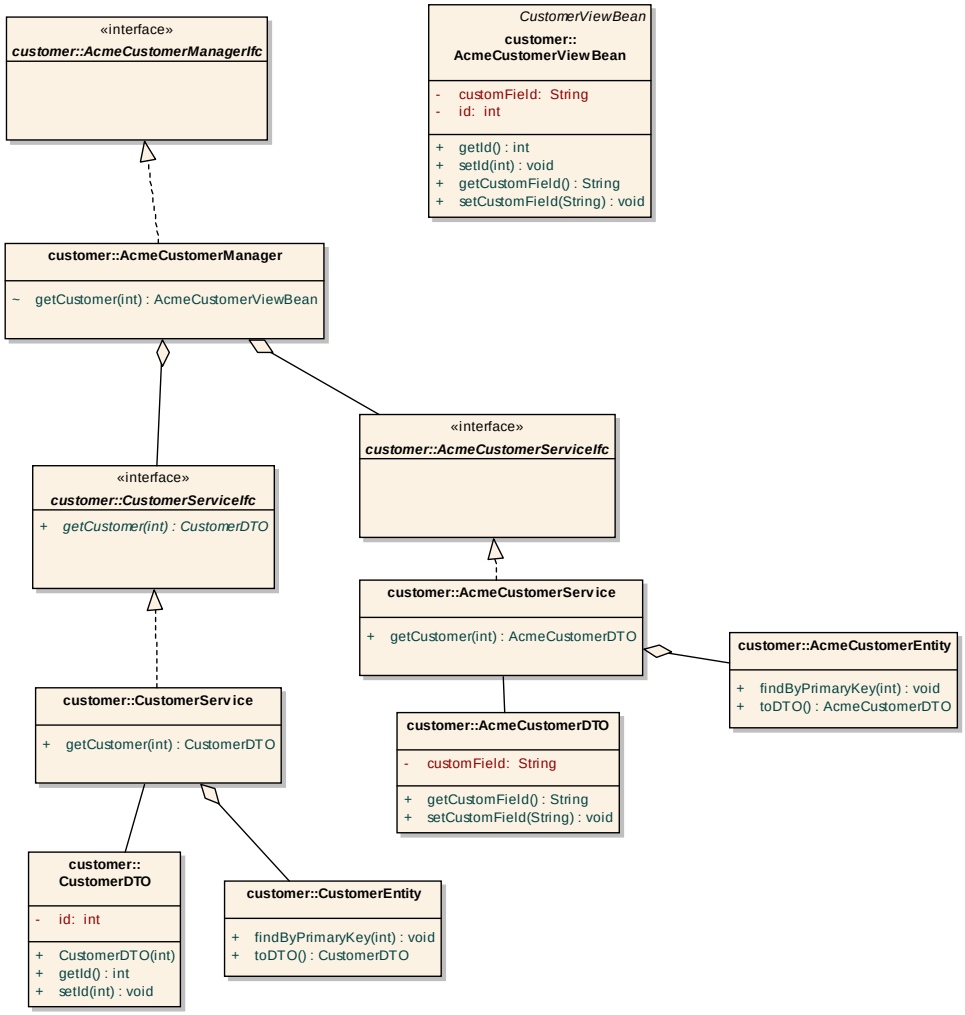
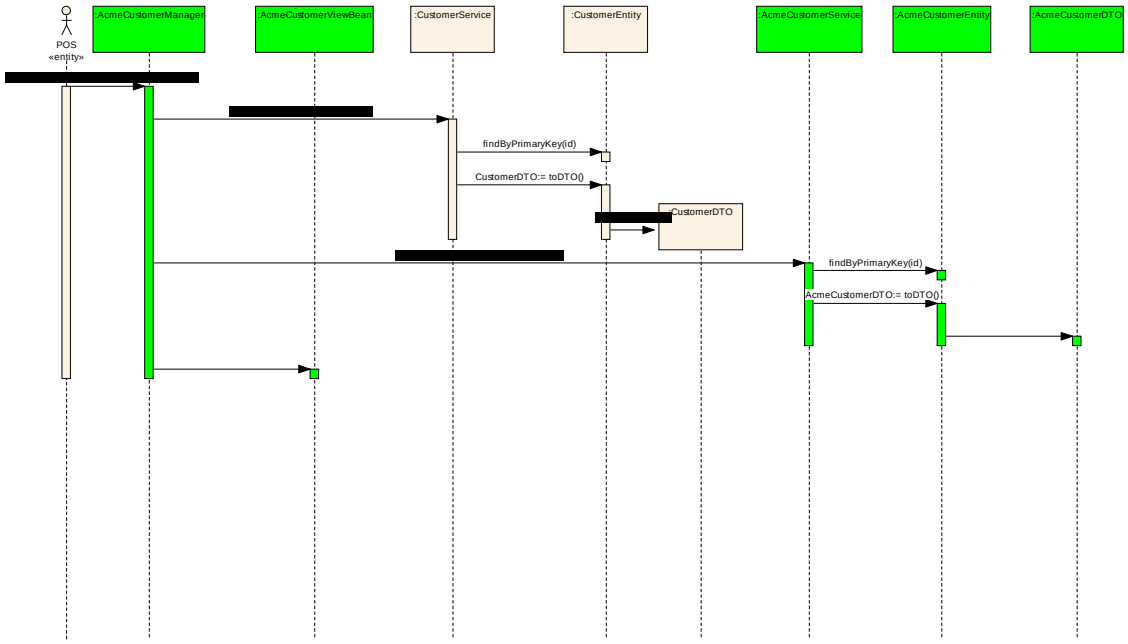


Figure 7-12 Extension with Composition



Data Extension through Composition

This strategy describes having the entity 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.

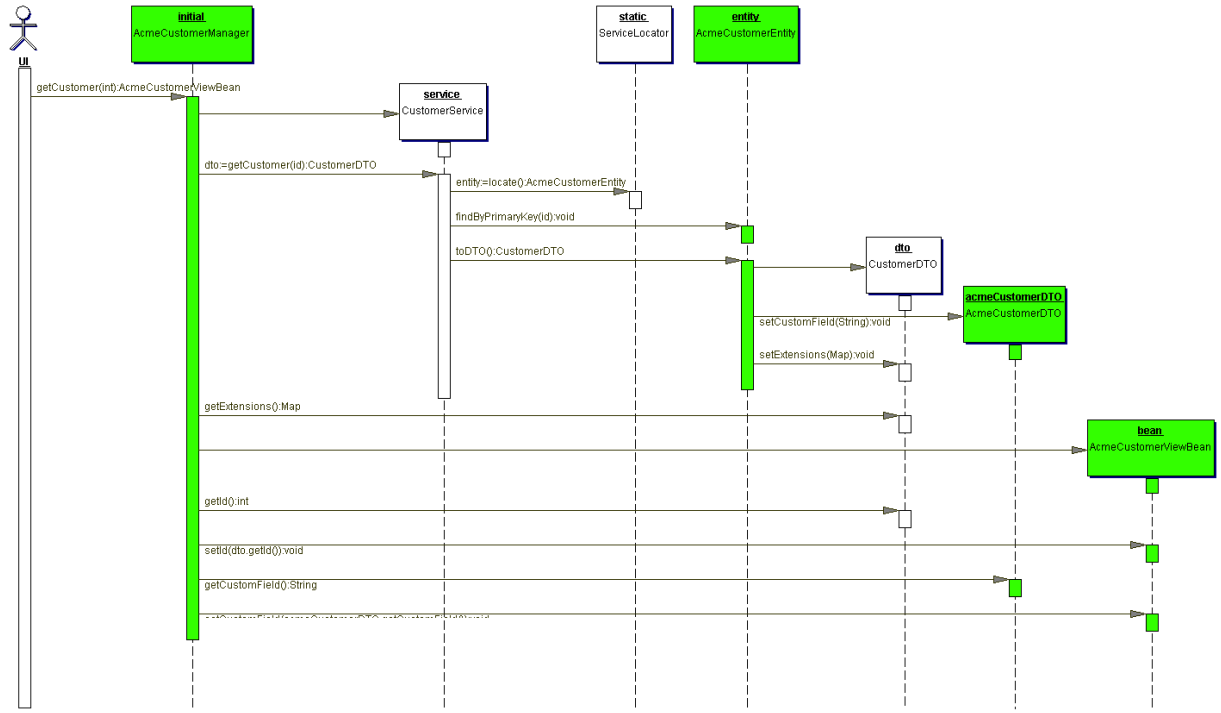
This scenario alters the UI layer (JSP/Action/ViewBean) and add a new ApplicationManager method to call assemble the ViewBean from the extensible DTO provided by the replaced Entity bean. Slight modifications to the Service session bean may be necessary to support the `toDTO()` and `fromDTO` (`ExtensibleDTOIfc dto`) methods on the Entity bean, depending on base product support of extensions on the particular entity bean.

- Create the new ApplicationManager session facade.
- Create the new ViewBeans required of the UI.
- Create new Entity bean that references the original data to construct a base product DTO that additionally contains the custom data using the extensible DTO pattern.
- Create new DTO based on the extensible DTO pattern.
- Create new JSP pages to reference the additional data.
- Change the deployment descriptors that describe which implementation to use for a particular Entity bean.
- Change the new Struts configuration and Action classes that reference the customized Application Manager Session facade.

- If necessary change the Commerce Service Session facade to give control of the `toDTO` and `fromDTO` methods to the Entity bean and not assemble/disassemble the DTO in this layer as it does not give a good plug point for the Extensible DTO's.

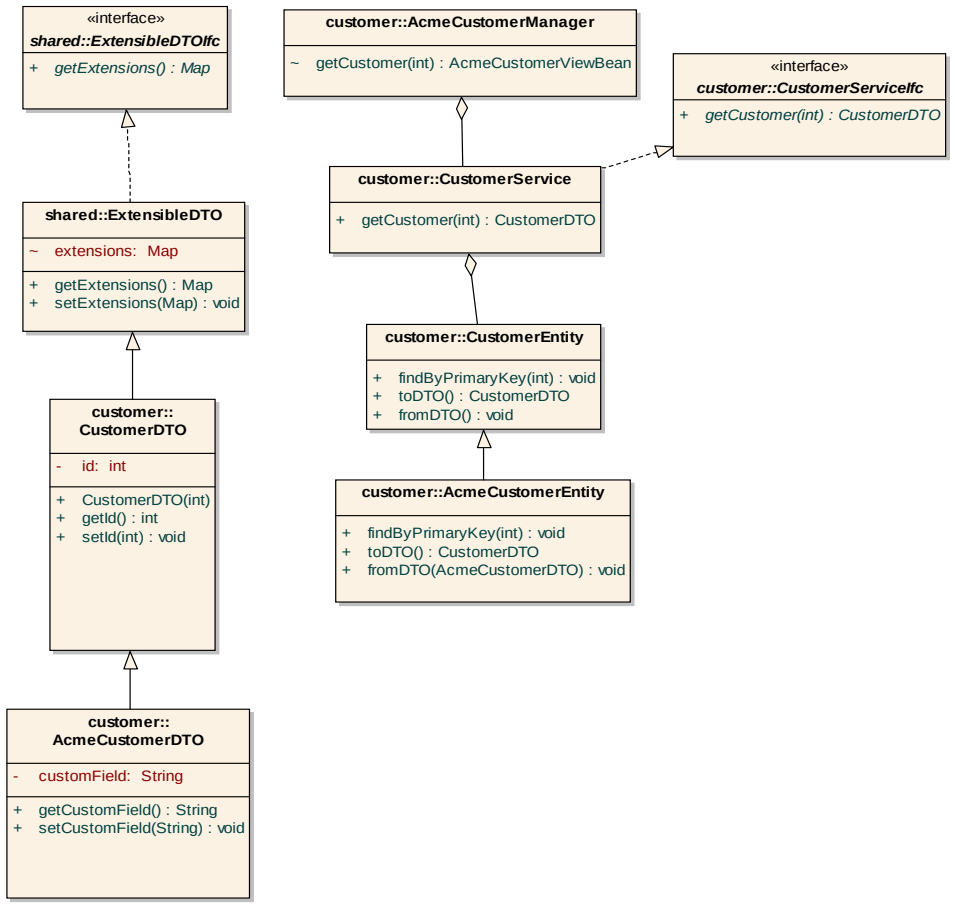
The following diagram describes the life cycle of the data throughout the request.

Figure 7-13 Data Extension Through Composition



The following class diagram describes the various classes created.

Figure 7-14 Data Extension Through Composition: Class Diagram



CHAPTER 8

GENERAL DEVELOPMENT STANDARDS

The following standards have been adopted by 360Commerce 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 360Commerce applications.

Basics

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

Java Dos and Don'ts

The following dos and don'ts 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.
- DON'T have an empty catch block. This destroys an exception from further down the line that might include information necessary for debugging.
- DON'T concatenate strings. 360Commerce products tend to be string-intensive and string concatenation is an expensive operation. Use `StringBuffer` instead.
- DON'T 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.
- DON'T use a static array of strings.
- DON'T use public attributes.
- DON'T 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 Table 8-1.

Table 8-1 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.

Note: A code block is defined as a number of lines proceeded with an opening brace and ending with a closing brace.

- 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, etc.) 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 PVCS tag for revision and log history.

Code Sample 8-1 Header Sample

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

    Copyright (c) 1998-2003 360Commerce, Inc.    All Rights Reserved.

    $Log$

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
package com._360commerce.samples;

// Import only what is used and organize from lowest layer to highest.
import com.ibm.math.BigDecimal;
import com._360commerce.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). Additional naming conventions follow:

Table 8-2 Naming Conventions

Element	Description	Example
Package Names	Package names are entirely lower case and should conform to the documented packaging standards.	`com.extendyourstore.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 Sybase-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.

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.

DB2

Table 8-3 shows examples of potential problems in DB2 SQL code.

Table 8-3 DB2 SQL Code Problems

Problem	Problem Code	Corrected Code

Don't use quoted integers or unquoted char and varchar values; these cause DB2 to produce errors.	`CREATE TABLE BLAH`
	`(`
	`  FIELD1 INTEGER,`
	`  FIELD2 CHAR(4)`
	`);`
	`INSERT INTO BLAH (FIELD1,`
	`FIELD2) VALUES ('5', 1020);`
		`CREATE TABLE BLAH`
		`(`
		`  FIELD1 INTEGER,`
		`  FIELD2 CHAR(4)`
		`);`
		`INSERT INTO BLAH (FIELD1,`
		`FIELD2) VALUES (5, '1020');`

Don't try to declare a field default as NULL.	`CREATE TABLE BLAH`
	`(`
	`  FIELD1 INTEGER NULL,`
	`  FIELD2 CHAR(4) NOT NULL`
	`);`
		`CREATE TABLE BLAH`
		`(`
		`  FIELD1 INTEGER,`
		`  FIELD2 CHAR(4) NOT NULL`
		`);`

MySQL

MySQL does not support sub-selects.

Oracle

Table 8-4 provides some examples of common syntax problems which cause Oracle to produce errors.

Table 8-4 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)`
		`);`

PostgreSQL

PostgreSQL does not currently support the command `ALTER TABLE BLAH ADD PRIMARY KEY`. However, it does support the standard `CREATE TABLE` command with a `PRIMARY KEY` specified. For this reason, the PostgresqlDataFilter converts SQL of the form shown in Code Sample 8-2 into the standard form shown in Code Sample 8-3.

Code Sample 8-2 SQL Code Before PostgresqlDataFilter Conversion

```
CREATE TABLE BLAH
(
 COL1 INTEGER NOT NULL,
 COL2 INTEGER NOT NULL,
 COL3 INTEGER,
);

ALTER TABLE ADD PRIMARY KEY (COL1, COL2)
```

Code Sample 8-3 SQL Code After PostgresqlDataFilter Conversion

```
CREATE TABLE BLAH
(
 COL1 INTEGER NOT NULL,
 COL2 INTEGER NOT NULL,
 COL3 INTEGER,
 PRIMARY KEY (COL1, COL2)
);
```

Note: There must be a new line and "(" after the `CREATE TABLE` command for the PostgresqlDataFilter's conversion to work, properly formatting the SQL.

Sybase

Sybase does not throw errors if a table element is too large; it truncates the value. If using a VARCHAR(40), use less than 40 characters.

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 the Point-of-Sale 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. The following AntiPatterns introduce bugs and reduce the quality of code.

Table 8-5 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.
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.

Table 8-5 Common AntiPatterns

Pattern	Description	Solution
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 Functional Requirements, including details like maximum length of fields and dialog message text.	Read the Functional 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-Sale. Another example is a making change to a field in the 360Store database for a Back Office feature without handling the Point-of-Sale 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. It is important that developers on customer projects whose code may be rolled back into the base product follow these standards as well as the guidelines in Chapter 7, “Extension Guidelines.”

- 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 360Commerce applications.

Internationalization

The following are some general guidelines for maintaining an internationalized code base which can be localized when needed. Refer to other documents for detailed instructions on these issues.

- All displayable text must be referenced from the appropriate resource bundle and properties file, so that the text can be changed when needed.
- Numbers, currency, and amounts must be displayed using Java internationalization conventions, so that appropriate symbols and number dividers can be used for the current locale.
- Formats and conventions related to dates, times and calendars are locale sensitive. All the date, time and calendar related operations must use DateFormat, SimpleDateFormat and Calendar classes, instead of the Date class. Remove hardcoded dates (mm/dd/yyyy, etc). Use the formats available as part of the DateFormat class.
- Properties in the `application.properties` file specify default and supported locales:
  ```
  default_locale=en_US
  supported_locales=en_US,fr_CA,en_CA
  ```
- Help files for new screens must be created in the appropriate locale directory, and pos\config\ui\help\ helpscreens.properties must be updated.
- Display database driven locale sensitive data according to the current locale.

Logging

360Commerce's systems 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.
- Logging uses few CPU cycles, so use debugging statements freely.
- Use the correct logging level:
  - FATAL—crashing exceptions
  - 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, etc.), 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:

Code Sample 8-4 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:

Code Sample 8-5 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:

Code Sample 8-6 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; don't create a custom exception if there is a problem with the formatting of a string (ApplicationFormatttingException) 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):

Code Sample 8-7 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:

Code Sample 8-8 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:

Code Sample 8-9 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:

Code Sample 8-10 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:

Code Sample 8-11 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:

Code Sample 8-12 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:

Code Sample 8-13 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.