Oracle® Retail Store Inventory Management

Implementation Guide, Volume 4 - Extension Solutions
Release 14.1
E53315-01

December 2014

ORACLE

Oracle® Retail Store Inventory Management Implementation Guide, Volume 4 - Extension Solutions, Release 14.1

E53315-01

Primary Author: Kris Lange

Value-Added Reseller (VAR) Language

Oracle Retail VAR Applications

The following restrictions and provisions only apply to the programs referred to in this section and licensed to you. You acknowledge that the programs may contain third party software (VAR applications) licensed to Oracle. Depending upon your product and its version number, the VAR applications may include:

(i) the **MicroStrategy** Components developed and licensed by MicroStrategy Services Corporation (MicroStrategy) of McLean, Virginia to Oracle and imbedded in the MicroStrategy for Oracle Retail Data Warehouse and MicroStrategy for Oracle Retail Planning & Optimization applications.

(ii) the **Wavelink** component developed and licensed by Wavelink Corporation (Wavelink) of Kirkland, Washington, to Oracle and imbedded in Oracle Retail Mobile Store Inventory Management.

(iii) the software component known as **Access Via™** licensed by Access Via of Seattle, Washington, and imbedded in Oracle Retail Signs and Oracle Retail Labels and Tags.

(iv) the software component known as **Adobe Flex™** licensed by Adobe Systems Incorporated of San Jose, California, and imbedded in Oracle Retail Promotion Planning & Optimization application.

You acknowledge and confirm that Oracle grants you use of only the object code of the VAR Applications. Oracle will not deliver source code to the VAR Applications to you. Notwithstanding any other term or condition of the agreement and this ordering document, you shall not cause or permit alteration of any VAR Applications. For purposes of this section, "alteration" refers to all alterations, translations, upgrades, enhancements, customizations or modifications of all or any portion of the VAR Applications including all reconfigurations, reassembly or reverse assembly, re-engineering or reverse engineering and recompilations or reverse compilations of the VAR Applications or any derivatives of the VAR Applications. You acknowledge that it shall be a breach of the agreement to utilize the relationship, and/or confidential information of the VAR Applications for purposes of competitive discovery.

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your beta trial agreement only. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Master Agreement, Oracle License and Services Agreement, Oracle PartnerNetwork Agreement, Oracle distribution agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

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

Contents

Send Us Your Comments

Oracle Retail Store Inventory Management, Implementation Guide, Volume 4 - Extension Solutions, Release 14.1

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

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

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

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

> **Note:** Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the Online Documentation available on the Oracle Technology Network Web site. It contains the most current Documentation Library plus all documents revised or released recently.

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

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

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

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

Preface

The *Oracle Retail Store Inventory Management Implementation Guide – Volume 4, Extension Solutions* guide contains the requirements and procedures that are necessary for the retailer to extend and customize the Store Inventory Management application.

Audience

This document is intended for the Oracle Retail Store Inventory Management application integrators and implementation staff, as well as the retailer's IT personnel.

Related Documents

For more information, see the following documents in the Oracle Retail Store Inventory Management Release 14.1 documentation set:

- *Oracle Retail Store Inventory Management Implementation Guide, Volume 1 – Configuration*
- *Oracle Retail Store Inventory Management Implementation Guide, Volume 3 – Mobile Store Inventory Management*
- *Oracle Retail Store Inventory Management Implementation Guide, Volume 2 – Integration with Oracle Retail Applications*
- *Oracle Retail Store Inventory Management Installation Guide*
- *Oracle Retail Store Inventory Management Operations Guide*
- *Oracle Retail Store Inventory Management Data Model*
- *Oracle Retail Store Inventory Management Release Notes*
- *Oracle Retail Store Inventory Management User Guide*

Customer Support

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

https://support.oracle.com

When contacting Customer Support, please provide the following:

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

Review Patch Documentation

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

Documentation Accessibility

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

Access to Oracle Support

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

The following documents are available through My Oracle Support. Access My Oracle Support at the following URL:

http://support.oracle.com/

Enterprise Integration Guide (Located in the Oracle Retail Integration Suite Library on the Oracle Technology Network)

The Enterprise Integration Guide is an HTML document that summarizes Oracle Retail integration. This version of the Integration Guide is concerned with the two integration styles that implement messaging patterns: Asynchronous JMS Pub/Sub Fire-and-Forget and Web Service Request Response. The Enterprise Integration Guide addresses the Oracle Retail Integration Bus (RIB), a fully distributed integration infrastructure that uses Message Oriented Middleware (MOM) to integrate applications, and the Oracle Retail Service Backbone (RSB), a productization of a set of Web Services, ESBs and Security tools that standardize the deployment.

Supplemental Documentation on My Oracle Support

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

Improved Process for Oracle Retail Documentation Corrections

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

This process will prevent delays in making critical corrections available to customers. For the customer, it means that before you begin installation, you must verify that you have the most recent version of the Oracle Retail documentation set. Oracle Retail documentation is available on the Oracle Technology Network at the following URL:
http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html

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

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

Oracle Retail Documentation on the Oracle Technology Network

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

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

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

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

Conventions

The following text conventions are used in this document:

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

1

Customization

Retailers often modify retail software either in-house or have it modified through third-party system integrators. If the customization efforts are not done properly, the deployed product might not operate correctly. A poorly customized product is difficult to upgrade and deploy. This situation causes serious issues for the retailer, Oracle Retail, and any system integrators involved. This chapter aims to mitigate that risk by providing guidance on how to customize safely and effectively.

Build/Packaging/Deployment Related

To build and deploy customizations correctly, make a separate project for custom-written code. Build a custom JAR with the newly created code. This custom JAR should be placed first in the execution classpath so that classes found within the JAR are chosen by the JVM rather than the base classes. This requires un-signing and re-signing all the JARs in the Oracle Retail Store Inventory Management (SIM)-client application, since all JARs must be signed with the same signature for Web Start to work correctly. Consult the jarsigner documentation from Sun for further information on the JAR un-signing/signing process.

Architecture

Do not connect directly to the Oracle Retail Management System (RMS) or SIM database using JDBC or other database connection software from any client layer, whether that is the PC client, Web Service, or Wavelink server layer. This will lead to being unable to guarantee data integrity and thus SIM can no longer be guaranteed to work.

For more information regarding how to gain the skills and knowledge to implement SIM successfully, see Oracle Retail's Specialization program on the Oracle Partner Network (OPN) located here:

http://www.oracle.com/partners/en/partner-with-oracle/training/index.html#cw01step-2.htm

Process Overview

In order to outline recommended approaches to adding new functionality, this document contains three examples of extensions to the base source code:

- Adding new attributes to pre-existing data and workflows. This includes how to update the database, business and client layers.
- Adding completely new data and workflows to the application including modifying all layers.
- Modifying incoming and outgoing data transport to external systems.

The Factory Plan

SIM provides numerous points in our architecture wherein the factory pattern is used to allow customers to add functionality to the system. The general idea of a factory pattern is that when a new Java object needs to be created to supply some piece of functionality, it is requested through a factory that returns a configured version of the object.

FactoryInterface

It starts by defining a factory interface to define the objects to be created. Example interface methods might be createObjectA() and createObjectB(). The factory interface does not need to be altered by the customer.

Factory

A static factory class will implement this interface with statically implemented methods that get a factory implementation from a configuration file and then use the configured implementation to create the actual object. The factory is not alterned by the customer.

```
public static ObjectA createObjectA() {
    return getFactoryImplementation().createObjjectA();
}
```

FactoryImplementation

This class contains the actual implementation of the original interface that is executed in product. Since this implementation is assigned to the factory in a configuration file, it can be swapped without changing any code. Customers can implement their own factory implementations to override and provide jave objects that have been extended.

```
SomeConfig.xml
Factory=FactoryImplementation
```

The above configuration can be altered to:

```
SomeConfig.xml
Factory=MyFactoryImplementation
```

And code can be written such as:

```
MyFactoryImplementation
public ObjectA createObjectA() {
    return new MyObjectA();
}
```

Pre or Post Processing Pattern

In all the various areas of the system where the factory pattern is used to allow customization of the data or processes, SIM designed such areas with the notion that customers would add “pre” or “post” processing and there would be no modification of the critical business code and value that SIM supplies.

For example, if the factory pattern is used to override the Inventory Adjustment EJB Services with a custom implementation, the following couple of extensions are allowed.

Pre-Processing

Pre-Processing is used to add addition logic prior to executing the standard SIM logic. This could include additional validations against data not normally performed or adding additional logic to modify the data object prior to being received by SIM code. So the customer implementation might look something like this:

```
public void updateInventoryAdjustment(InventoryAdjustment adjustment) {
        // Perform additional code prior to SIM execution
        super.updateInventoryAdjustment(adjustment);
 }
```

Since all customized objects should be sub-classes of the original class, it will still have access to all the original functionality.

Post-Processing

Post-Processing is used to add addition logic after executing the standard SIM logic. This could include updating additional data elements in the database or sending notification or other tasks that should occur after the data has already been used or persisted by SIM. So the customer implementation might look something like this:

```
public void updateInventoryAdjustment(InventoryAdjustment adjustment) {
        super.updateInventoryAdjustment(adjustment);
        // Perform additional code after SIM execution
}
```

Adding Attribute to Pre-Existing Data & Workflows

To add the attribute **Season** to the current Item object and keep the system open to upgrade without difficulty, do the following:

1. Add the New Attribute to the Database
2. Update Business Object or Value Object
3. Update the Services
4. Update the PC Screen
5. Update the HH Forms

Add the New Attribute to the Database

The following information describes adding new attributes in a database.

Create New Attribute Column

Add the appropriate attribute to SIM's current database table or make a new table to store the attribute. Do not alter or remove any columns as this will render SIM unable to access the table.

SIM's item master is ITEM, so add a column to the ITEM table that is named GRI_SEASON. Naming should include a unique tag from the company customizing the code in order to easily tell the difference between base and customization for upgrade purposes. If it is decided to add a new item attributes table where SEASON was just one of the attributes, then the name of the table might be GRI_ITEM_ATTRIBUTE. If the entire table is custom, it is then appropriate to simply use the column name SEASON.

Update DAO (Database Access Objects)

The following update injectors information describes updating Database Access Objects.

Create DAO Code to Access Information

SIM will not automatically access this new data in the database. Custom DAO (Data Access Object) code must be written to retrieve this information. It is advisable to always create completely new DAO source code for this customization.

You may choose to write data access layer code to retrieve these properties in whatever manner you choose to. However, if you want to use SIM's frameworks for data retrieval and database connection pooling, declare the new data access layer class and have it extend BaseOracleDao:

```
public class ItemAttributeOracleDao extends BaseOracleDao {
```

BaseOracleDao supplies several helpful methods to perform a lot of the major duties of data access. It enables the user to execute() statements, batch statements, and stored procedures as well as perform several different types of queries. Becoming familiar with the Application Programming Interfaces (API)s in this class will help in the process of developing database layer code.

Performing Simple Insert

To perform a simple insert, call the execute() method with a ParametricStatement that contains your insert SQL.

```
String insertSQLString = "insert into GRI_ITEM_ATTRIBUTE (ITEM_ID, SEASON) values (?, ?)";

List<Object> insertSQLParams = new ArrayList<>();
insertSqlParams.add(itemId);
insertSqlParams.add(season);

execute(new ParametricStatement(insertSQLString, insertSQLParams));
```

A ParametricStatement is a useful object within the SIM code. It is constructed with the insert SQL string and a list of objects to substitute in the SQL string where "?" appears. Performing a simple update is accomplished as in the previous example.

Performing Simple Query

To perform a simple query, call the query() method with a ParametricStatement that contains your select SQL:

```
String querySql = "select ITEM_ID, SEASON from GRI_ITEM_ATTRIBUTE where ITEM_ID = ?";

List<Object> querySqlParams = new ArrayList<>();
querySqlParams.add(itemId);

List<GriItemAttributeDataBean> beans
      = query(new GriItemAttributeDataBean(),querySql, querySqlParams);
```

Using the query helper method of BaseOracleDao will require creating a data bean. A data bean is a simple class that contains the attributes of the table in question. The data bean implements some required API in order to let the framework execute the SQL and then populate the beans with the correct information.

Creating a Data Bean

To create a data bean, create a class that extends BaseDataBean. Implement both the methods defined by the superclass interface:

```
public class GriItemAttributeDataBean extends BaseDataBean<GriItemAttributeDataBean> {
    private String itemId;
    private String season;
    public String getSelectSql() {
        return "select ITEM_ID, SEASON from GRI_ITEM_ATTRIBUTE where ITEM_ID = ?"
    }
    public GriItemAttributeDataBean read(ResultSet resultSet) throws SQLException {
       GriItemAttributeDataBean bean = new GriItemAttributeDataBean ();
       bean.itemId = getStringFromResultSet(resultSet, "ITEM_ID");
       bean.season = getStringFromResultSet(resultSet, "SEASON");
       return bean;
    }
}
```

Update Business Object or Value Object

To customize the attributes on a business/value object within the system appropriately, subclass the business/value object in question and add the attribute to the sub-class only:

```
public class GriItem extends Item {
    private String season;
    public GriItem(String itemId) {
        super(itemId);
    }
    public String getSeason() {
        return season;
    }
    public void setSeason(String season) {
         this.season = season;
    }
}
```

Next, subclass the BOFactoryImpl to override the item creation method and replace it with your own. Once configuration is altered to use the custom factory, everywhere in the code where an Item would have been created/instantiated, a GriItem is instantiated instead:

```
public class GriBOFactoryImpl extends BOFactoryImpl {

    public Item createItem(String itemId) {
        return new GriItem(itemId);
    }
}
```

To configure to use the new customization, simply modify the server.cfg file with the classpath of your new code:

```
BO_FACTORY_IMPL=gri.custom.code.GriBOFactoryImpl
```

Update the Services

The same process used to customize the objects within the system is used to override services at the service layer. A subclass of the service is made, followed by pointing the correct configuration file at the subclass instead of the original SIM service:

```
public class GriItemServerServices extends ItemServerServices {
  public StockItem readStockItem(String itemId, Long storeId) throws Exception {
     StockItem stockItem = super.readStockItem(itemId, storeId);
     GriItem griItem = (GriItem) stockItem.getItem();
     griItem.setSeason(new GriItemAttributeDao().selectSeason(itemId));
     return stockItem;
```

```
  }
}
```

When overriding an implementation method in the service, it is advisable to call the superclass for standard SIM processing and then perform customized processing afterwards. This ensures continued functionality with future releases.

After creating the customized service, create a customized factory as with the business objects (extending SIM's ServerServiceFactoryImpl), and then configure the common.cfg file to use your customized factory implementation:

```
public class GriServerServiceFactoryImpl extends ServerServiceFactoryImpl {
    public ItemServices createItemServices() {
        try {
            return new GriItemServerServices();
        } catch (Throwable t) {
            LogService.error(this, "Could not create GriItemServerServices", t);
            return null;
        }
    }
}
```

Example: *common.cfg*

```
SERVER_SERVICE_FACTORY_IMPL=gri.custom.code.GriServerServiceFactoryImpl
```

Service Index

The following is a list of services within SIM:

- ActivityHistoryServices
- ActivityLockServices
- BarcodeServices
- BatchServices
- ConfigServices
- CustomThemeServices
- DirectDeliveryServices
- FulfillmentOrderDeliveryServices
- FulfillmentOrderPickServices
- FulfillmentOrderReversePickServices
- FulfillmentOrderServices
- InventoryAdjustmentServices
- ItemBasketServices
- ItemPriceServices
- ItemRequestServices
- ItemServices
- ItemTicketServices
- MdseHierarchyServices
- MpsServices
- NoteServices
- PosTransactionServices
- PrintFormatServices
- ProductGroupServices
- ProductGroupScheduleServices

- ReportingServices
- ReturnServices
- SecurityServices
- ShelfAdjustmentServices
- ShelfReplenishmentServices
- ShipmentServices
- SourceServices
- StockCountServices
- StockCountChildServices
- StockCountLineItemServices
- StoreOrderServices
- StoreSequenceServices
- StoreServices
- ToleranceAdminServices
- TransactionHistoryservices
- TransferServices
- TranslationServices
- UDAServices
- UINServices
- UOMServices
- WarehouseDeliveryServices

Updating Commands

Most service implementations are provided by commands. Commands are small blocks of self-contained code that allow for better organization and structure. Each EJB service implementation in SIM usually directly accesses the DAO layer or instead, executes a command. The command has an execute() method that calls doExecute() which can be overwritten by sub-classes.

For the purposes of customization, it is easier and more concise to customize the actual EJB service as previously described rather than try to customize a command. In the case that a customer wishes to customize a particular command, these are the steps to do so.

A subclass of the command is made, followed by pointing the correct configuration file at the subclass instead of the original SIM command:

```
public class MyAdjustmentUpdateCommand extends InventoryAdjustmentUpdateCommand {
  protected void doExecute() throws Exception {
      // Add Pre Processing Here
     super.doExecute();
      // Add Post Processing Here
  }
}
```

After creating the customized command, create a customized factory as with the business objects (extending SIM's CommandFactoryImpl), and then configure the common.cfg file to use your customized factory implementation:

```
public class MyCommandFactoryImpl extends CommandFactoryImpl {
    public InventoryAdjustmentUpdateCommand()
           createInventoryAdjustmentUpdateCommand() {
       return new InventoryAdjustmentUpdateCommand();
    }
}
```

Example: *server.cfg*

```
COMMAND_FACTORY_IMPL=gri.custom.code.MyCommandFactorympl
```

Update the PC Screen

In order to view or enter additional information on the PC, a new UI space must be created instead of modifying current screens. This is done by adding an additional button to the navigation of the application. In order to add a new button, a new row must be put into the PC_MENU_ITEM table.

Table: *PC_MENU_ITEM*

Name	Description
MENU	This contains the fully qualified classpath to the screen that contains the menu.
NAME	This is the name of the button to be displayed on the menu. This value is used as a translation key by the client to get the displayable text.
PERMISSION	This is the permission key associated to the button. If the user does not have the assigned permission, the button does not display. Null is valid.
MENU_ITEM_ACTION	This can be one of three values: NONE, BACK or another menu.
MENU_ITEM_ORDER	Buttons are displayed for the MENU in the sequence of lowest to highest MENU_ITEM_ORDER value.
IS_DEFAULT	Y if the button should be the default button for the window. N if not the default button for the window.

If **NONE** is entered in MENU_ITEM_ACTION, then when the navigation button is pressed, no navigation takes place. If **BACK** is entered, then when the navigation button is pressed, the screen navigates to the previous screen in the workflow. If a value matching a MENU column in the table is entered (another menu/screen), then when the navigation button is pressed, the application navigates to the specified screen. In all three of these scenarios, the button action is first sent to the actionPerformed() method of the screen before any navigation takes place.

The following table is an example record to add a button to the item detail screen:

Table: *PC_MENU_ITEM, Item Detail*

Column	Value
MENU	oracle.retail.sim.shared.swing.item.ItemDetailScreen
NAME	Additional Details
PERMISSION	null
MENU_ITEM_ACTION	gri.retail.sim.custom.AdditionalItemDetailScreen
MENU_ITEM_ORDER	999
IS_DEFAULT	N

The following table is an example record to create a menu for the new custom screen:

Table: *PC_MENU_ITEM Table, Custom Screen*

Column	Value
MENU	gri.retail.sim.custom.AdditionalItemDetailScreen
NAME	Cancel
PERMISSION	null
MENU_ITEM_ACTION	BACK
MENU_ITEM_ORDER	1
IS_DEFAULT	N

Next, create the AdditionalItemDetailScreen class which must extend SimScreen and implement all the required methods. After that, design and build the screen (see Adding New Workflows).

Update the HH Forms

Customizing the HH application is more difficult than other areas of the application. Wavelink does not supply simple or straightforward ways to customize their forms. In order to add additional item information to the HH item lookup workflow, the handheld SIM code source files need to be modified directly.

It is strongly suggested that all new features, information or processes be added with completely new handheld forms. For how to build forms in the Wavelink server, see Wavelink documentation. Once the new forms are built, a single form may be customized to contain a menu option or command to navigate to the new workflow. This will minimize the amount of rework that needs to be done with each release update of the handheld source code.

Adding New Workflows

Rather than just adding functionality to an existing workflow, this section describes how to add a new workflow to SIM. It focuses on how to design and implement the new workflow to minimize the impact of upgrades to the customization. The example workflow added to the system is the ability to create new serial numbers in an In Stock status without moving them using a system transaction, that is, entering serial numbers into the UIN_DETAIL table without receiving them or creating an inventory adjustment. This type of functionality is used primarily to data seed serial numbers that should have already been in the system.

The following topics will be covered:

- Building Business Objects
- Building Enterprise Java Bean (EJB) Services
- Building Data Access
- Building PC Screens
- Building Wireless Forms
- Exceptions

Example Code

This section references example code that is packaged independent of this document. For more information about this example code, see Appendix: Code Examples.

The following table lists the various files and their general description.

Table: *Example Code Files*

Name	Description
CustomSerialNumber	Business object representing the new serial number.
CustomUINBean	The Enterprise Java Bean (EJB) that implements the CustomUINInterface.
CustomUINCountTableEditor	Table editor used in the UI CustomUINCreateScreen.
CustomUINCreateDialog	Pop-up dialog that enables users to enter the new UINs.
CustomUINCreateDialogModel	Model for the pop-up dialog.
CustomUINCreateDialogTableEditor	Table editor used in CustomUINCreateDialog to enter individual UINs.
CustomUINCreateModel	Model for the create screen (track data, talk to server).
CustomUINCreatePanel	Panel for the create screen (contains UI widgets).
CustomUINCreateScreen	Screen for the create screen (interfaces with navigation system).
CustomUINCreateWrapper	A client-side wrapper object for the custom serial number.
CustomUINEJBServices	Client-side EJB Service that looks up and accesses the EJB bean.
CustomUINInterface	Defines the API for Custom UIN EJB.

Name	Description
CustomUINOracleDAO	Database Access Object for custom UIN create logic.
CustomUINServerServices	Implementation of CustomUINServices.
CustomUINServices	Defines the API for Custom UIN Services.
CustomUINDataBean	A data bean for a new theoretical CUSTOM_UIN table.

Building Business Objects

In the SIM architecture, business objects represent basic concepts within the Store Inventory Management domain. Some examples of significant objects within SIM are Item, Store, StockCount, Transfer, DirectDelivery and Receipt. Most business objects contain very little business logic. Rather, a business object contains the data associated with the domain concept. The majority of code within a business object concerns itself with getting and setting data values on the business object. The business logic associated with the concept is contained within the ServerServices (or service layer) that uses the business object. Because they contain all the data, business objects flow across all three layers of the SIM architecture.

These business data objects can be designed in any manner as long as they implement Serializable and Cloneable so that they can be used as parameters to server APIs. If the business object extends SIM's BusinessObject class, this is automatically done as well as gaining other useful methods to use.

For more information, see Example: *CustomSerialNumber* in Appendix: Code Examples.

The Business Object Class

All SIM business objects extend from the single class BusinessObject. Many classes can be in a hierarchy chain, but BusinessObject should always be the top level object. Refer to the Javadoc for further information. Features provided by the class include:

Table: *BusinessObject Features*

Name	Description
isAttributesEqual()	Return **true** if two values are equal.
isAttributesNotEqual()	Return **true** if two values are not equal.
isPropertyModifiable()	Return **true** if the property specified is modifiable; **false** otherwise. This method makes an executeRule() call, but wraps the call in a try/catch block. If the rule returns an exception, the block catches the exception and returns **false**, otherwise it returns **true**. The implementation of the rule is just like that of any other business rule. The rule makes whatever checks are necessary to determine if the attribute is modifiable. It returns a RulesInfo object containing a text message if it is not modifiable, otherwise is returns an empty RulesInfo object. To alter this method, either override the method in business object sub-class (for example, CustomSerialNumber.Java) or configure a rule for the object to execute.

isCoherent()	Returns **true** if the object is currently coherent. This method makes an executeRule() call. It throws a business exception if the object is not coherent and returns **true** if it is.
executeRule()	Methods that access the rules engine. This allows business rules to be externally defined against business objects in a configuration file. When a method is called, the rule engine might dynamically load and execute business rules defined for a method.

Table: *BusinessObject Features (con't.)*

Name	Description
checkForNullParameter()	Method to determine if a parameter is null and throws a business style exception if it is.
clone()	A default implementation of clone() is provided. This implementation provides a deep copy of the object. All objects referenced by the business object are also fully copied. Often, this might not be what the developer wants.
cloneShallow()	Clone method that is the same as the standard Java object implementation of clone().
toString()	Uses reflection to convert all attributes of the BusinessObject to a single string.

There are two more types of data objects used within SIM to communicate between the client and server: the query filter and value object.

Query Filter

Business objects represent a single concept of the domain. Sometimes the objects must be selected in groups, often by some criteria, such as finding all employees whose last name begins with "**T"**. For those services that require filter criteria, a QueryFilter business object is created. These filters are used in the DAO layer to construct WHERE clauses when selecting and populating the business objects. There is no difference between creating a business object and creating a query filter object other than adding implement QueryFilter to its declaration.

```
public class MyQueryFilter extends BusinessObject implements QueryFilter
```

Value Object

A value object (VO) is a trimmed-down version of a business object that contains only the attributes needed for a very specific piece of the application. These objects are created to reduce data flow in areas that do not require that the attributes be updated or any business logic:

```
public class MyVO implements Serializable
```

There is no specific benefit or advantages to creating a value object or query filter other than consistency with naming patterns in SIM while doing customizations.

Customization Using the Rules Engine

The set() methods on business objects and query filters (but not value objects) always call executeRule(). When a method is called, the rule engine might dynamically load and execute business rules defined for a method. This enables business rules to be externally defined against business objects in a configuration file.

Rules are simple classes that validate business logic upon various objects within the system. They are executed primarily when attributes are set() on business objects, but there are also rules accessed when isPropertyModifiable() or isCoherent() are executed. There are already many rules defined in the system. Before creating a new rule, look through existing rules to see if the one you need already exists.

Creating a New Business Rule

To create a new business rule, create a new Java class that extends SimRule. This enables the rule to interact with SIM's rule framework. QuantityCannotBeNegativeRule is being used as an example of writing a business rule. This rule passes if the quantity is positive and fails if it is not.

```
public final class QuantityCannotBeNegativeRule extends SimRule
```

Override the execute() method. This first method performs some brief validation of the args parameters, but the standard coding practice is to break out the args array and cast to specific types to be passed to a second execute() method that performs the logic of the business rule. In the example below, the object parameter passed in by the rule engine is not needed. The first parameter of the array is the Quantity that needs to be validated:

Example*: Overriding the Execute() Method*

```
public RulesInfo execute(Object object, Object[] args) {
     return execute((Quantity) args[0]);
}
```

Implement the typed execute() method with the actual logic necessary to perform the desired validation:

Example*: Implementing the Typed Execute() Method*

```
private static final RulesInfo RULE_FAILED
             = new RulesInfo(ErrorKey.QUANTITY_INVALID_NEGATIVE);
private RulesInfo execute(Quantity quantity) {
    if ((quantity != null) && (quantity.doubleValue() < 0)) {
        return RULE_FAILED;
    }
    return RULE_PASSED;
}
```

In the example above, RULE_PASSED is returned at the end of the validation to indicate that no failure took place. Note that RULE_PASSED is actually an empty RulesInfo object declared in the SimRule super-class that should be used in all subclasses at the appropriate spots.

A RulesInfo wrapped around an error string indicates that the rule failed. The framework handles this failed rule information and converts it into the appropriate exception.

Logic in rules is intended strictly for validation checking. The logic should never update or modify the actual object that it is validating. Doing this would violate the contract of the rules engine in SIM and likely leave the business object in a non-coherent state.

Once the rule is developed, the rules configuration file must be updated. Update all occurrences of the rules_sim.xml file:

Example: *rules_sim.xml*

```
<object className="oracle.retail.sim.closed.invadjustment.InventoryAdjustment">
    <property id="setPackSize">
        <rule_class className=
           "oracle.retail.sim.closed.rules.common.PackSizeMustBePositiveRule"/>
    </property>
    <property id="setQuantity">
        <rule_class className=
           "oracle.retail.sim.closed.rules.common.QuantityCannotBeNegativeRule"/>
    </property>
    <property id="setComments" propertyNames="setComments">
        <rule_class className=
           "oracle.retail.sim.closed.rules.common.MaxCommentSizeRule"/>
    </property>
</object>
```

This xml file contains a list of classes and rules to execute when certain properties are modified. At the top level, an object is defined with a className containing the complete path to the object on which the validation should be done:

```
<object className="oracle.retail.sim.closed.invadjustment.InventoryAdjustment">
</object>
```

Within the class definition is a property definition where the ID is the method signature on which the validation should be performed:

```
    <property id="setQuantity">
    </property>
```

Within the property definition is the rules class definition where className is assigned the fully qualified path to the rule:

```
<rule_class className=
           "oracle.retail.sim.closed.rules.common.QuantityCannotBeNegativeRule"/>
```

Building Enterprise Java Bean (EJB) Services

The next step in creating the workflow is defining the server application programming interface (API) that is accessed to accomplish the work. This means designing an EJB to deploy in the server:

1. Define the Service API (see the Example: *CustomUINServices* in Appendix: Code Examples).

   Define the exact API to call on the server to perform some logic. These services files might have numerous APIs associated with them.

2. Define the API in an EJB interface (see the Example: *CustomUINInterface* in Appendix: Code Examples).

   The parameters and return value must be defined as CompressedObject if the API wants to take advantage of SIM's compression during communication between the client and server. A CompressedObject return value should be defined even if the return value is **null** or was defined in step 1 as **null**.

3. Develop the EJB Bean (see the Example: *CustomUINBean* in Appendix: Code Examples).

   When the EJB bean is developed, extending AbstractSimServiceBean enables quick access to helper methods such as handleException(), getSessionContext() and completeServiceContext().

   The EJB Bean must implement the interface defined in Step 2. In addition, note that to use the compression built in to SIM for quicker performance, even if the service is defined as null, a CompressedObject must be returned. Note that using the try/catch/finally allows the smooth handling of all exceptions and the completion of the service context even if the EJB failed with an exception.

4. Develop EJB Services (see the Example: *CustomUINEJBServices* in Appendix: Code Examples).

   The EJB Services class is designed to be the client-side of a service call. It handles the task of looking up the EJB Bean, instantiating the compressed objects around the arguments and making the call to the EJB Bean. The EJB Services use the EJBServicesManager to do cached lookups of the EJB Beans for improved performances. Following the pattern in the example is the easiest way to create an EJB Service.

5. Develop EJB Server Service (see the Example: *CustomUINServerServices* in Appendix: Code Examples).

   The EJB Server Services class contains the actual server side code that performs the business logic of the API. In this particular example, the code only needs to save the new serial numbers, so a Data Access Object (DAO) is instantiated and its API accessed.

6. Deploy EJB Services.

   The new EJB Bean needs to be deployed to the server. A system administrator or lead developer should be able to handle this process. The steps are normal steps for deploying an EJB into a server:

   a. The EJB must be packaged in a standard ejb style .jar.
   b. A reference to that .jar must be included in the META-INF/application.xml within the sim-server.ear.

Building Data Access

The next step is to write the DAO that communicates with the database. Using some of SIM's pre-existing tools can simplify this task.

Database Layer Development Tips

- Removing a column from the database breaks the code that attempts to read that table, so do not remove columns.
- New tables should always begin with a custom client-created prefix to keep it easily distinguishable from basic SIM tables. If the company name is Business Company Inc., then a table containing additional item information might be called BCI_ITEM_INFO.
- New columns added to an existing database table should begin with a custom prefix. If a company named Business Company Inc. wanted to add a new column to captured information about a stock count to the stock count header table, it might look like STOCK_COUNT.BCI_EXTRA_DATA.
- All new columns added to the existing tables must be nullable.
- If there is a foreign key reference in newly added tables, then keep the same column name as the table column name it references.
- Do not modify any of SIM's DAOs. Either sub-class and configure the DAO or create a completely new custom DAO.
- Ensure the DAO sub-classes BaseOralceDao so that connection pooling is handled properly.
- Ensure that all data beans sub-class FullDataBean or BaseDataBean.

Creating a DAO

Refer to Update DAO (Database Access Objects) for information about developing new DAOs. Also, refer to Example: *CustomUINOracleDAO* in Appendix: Code Examples for a code example.

The remainder of this section contains suggestions for using general SIM DAO classes. Reading all Javadoc on the following classes will help with understanding all the available APIs available.

Table: *DAO Layer Framework Classes*

Name	Description
BaseOracleDao	Abstract base class for all Oracle data access objects. Every DAO implementation must be a sub-class of this object. This class provides a range of generic functionality as well as access to the database connection factory.
BaseDataBean	Abstract base class for all data access beans. Every database bean should have this in its hierarchy.
BatchParametricStatement	Encapsulates SQL and the parameters to be used during its execution. It creates a batch of statements and executes them at the end. This performs much better under the scenario where the same SQL statement is executed multiple times.
FullDataBean	Abstract base class for all data beans that contain select, insert, update and delete functionality.

DAO Methods

Almost all DAO methods consist of creating a ParametricStatement or BatchParametricStatement and then calling the appropriate execute() method in the BaseOracleDao. In the saveSerialNumbers() method of the custom UIN example, a custom batch SQL statement is built using the UINDetailDataBean class attributes. Regular inserts and updates of a table are much simpler. For example, if a table called CUSTOM_UIN was created to temporarily hold our input, some simpler examples of how the DAO layer works (see the Example: *CustomUINBean* in Appendix: Code Examples) might be available.

The following would be the save method in the dao layer using the new data bean:

```
public void saveSerialNumbers(Long storeId, List<CustomSerialNumber> serialNumbers)
                                                                    throws SimServerException {
    String sql = CustomUinDataBean.UPDATE_SQL
                    + where(CustomUinDataBean.COL_ID, CustomUinDataBean.COL_STATUS);
    BatchParametricStatement batchStatement = new BatchParametricStatement(sql);
    CustomUinDataBean bean = new CustomUinDataBean();

    for (CustomSerialNumber serialNumber : serialNumbers) {
        bean.setId(serialNumber.getId());
        bean.setStatus(UINStatus.IN_STOCK.getCode());
        bean.setStoreId(storeId);

        List<Object> params = bean.toList(false);
        params.add(serialNumber.getId());
        params.add(serialNumber.getStatus().getCode());

        batchStatement.addParams(params);
    }
    executeBatch(batchStatement);
}
```

Building PC Screens

The SIM PC client is a Swing application. It is launched using WebStart and a browser. It communicates with the SIM server through EJBs. The application does not support offline functionality. If communication with the server is lost, an error message is displayed and the client is returned to its login screen.

> **Note:** Because the EJBs are services that run on the server, any client may be written against these remote services. A thin-client application could be written to take advantage of these services if the client wanted some functionality available through Web pages.

Customization Guidelines for the PC

- Always access the EJB services through the ClientServiceFactory class.
- Do not modify any framework-related code.
- To avoid conflicts with updates, only add functionality to client through adding navigation buttons that navigate to custom screens or launch custom dialogs.

PC Client Architecture

The client architecture is broken into five layers, only three of which are worked on to create new or customized PC functionality:

- Application Framework
- <name>Screen
- <name>Panel
- <name>Model
- EJB Service

Application Framework

There is an extensive framework of Swing-related classes used to launch and control the application as well as tools to make working within the Swing client easier. This framework is located in oracle.retail.sim.closed.swing.* packages. There is a wide area of functionality covered in this framework from layout tools, to customization tools, to internationalization hooks, to a navigation engine, to advanced tables and advanced widgets. These framework classes should not be modified.

<name>Screen

A screen is the top level of a single PC screen display. Its responsibility is to interact with the navigation system and delegate the actions of its menu items to the panel (see the Example: *CustomUINCreatePanel* in Appendix: Code Examples).

<name>Panel

The panel is where all the visual elements of a single PC screen are located. All the Swing widgets are declared here as well as any functional logic that touches those widgets (getting and setting the widget properties and values). All logic not associated directly with the widgets is delegated to the model (see the Example: *CustomUINCreatePanel* in Appendix: Code Examples).

<name>Model

The model is where all access to the service layer takes place as well as where any complex logic that needs to take place on the client is written (see the Example: *CustomUINCreateModel* in Appendix: Code Examples).

EJB Service

An EJB service is accessed through the ClientServiceFactory (or custom written EJB service) and should only be done from the application framework or from the model layer.

Navigation

There is both configurable navigation and hard-coded navigation within the SIM application.

External Configurable Navigation

The primary means of navigation in the PC application is through pressing the menu buttons displayed at the top of the screen. These menu buttons are determined using the information in PC_MENU_ITEM in the database. Adding, editing, and removing buttons for customization must be done by adding or altering information in this database table. See Update the PC Screen for more information.

Hard-Coded Navigation

The ability to navigate from within the <name>Screen or <name>Panel class is supplied in the framework by the navigate() and navigateLater() methods found in SimScreen and ScreenPanel classes. There are two different ways that hard-coded navigation takes place. The DirectDeliveryDetailScreen/Panel is an excellent example of this. Because it can be reached through screens that should not be returned to using the standard framework, very specific navigation is coded for the screen.

For example, in the handleDelete() method, the application return to the previous screen only if the direct delivery is successfully cancelled. This is done using the SimNavigation.BACK value as the navigate parameter. The remainder of the values in SimNavigation are the identities of the buttons used within the application:

```
public void handleDelete() throws Exception {
    lineItemTable.stopEditing();
    // Rest of method code removed
    if (model.isDirectDeliveryEmpty() && cancelEmptyDelivery()) {
         navigate(SimNavigation.BACK);
    }
}
```

NavigationEvents

The buttons on the top of each screen, located within the menu bar, are each attached to a navigation event. When the button is pressed, the navigationEvent() method is executed within the screen and the details of the button event are passed in as a parameter. SIM provides a way to intercept the events that take place upon a button press prior to each reaching a code screen. This allows for customization of additional menu buttons with only the most minor of SIM changes.

SimScreenNavigationListener

This is an abstact class that defines a single API: processEvent(). To customize, subclass this listener and implement the processEvent() method.

This is an abstact class that defines a single API: processEvent(). To customize, subclass this listener and implement the processEvent() method.

SimScreenNavigationListenerInterface

This is an interface implemented by the SimScreenNavigationListenerFactory. The interface defines a method that takes a screen Class as input and returns the correct navigation listener for that screen.

SimScreenNavigationListenerFactory

A factory that is called to get the appropriate listener for a screen. The implementation of the factory is configured in client.cfg file. Simply create your own implementation of this factory and configure it to be used.

The NavigationEvent class has a consume() method. If this method is called during the implementation of processEvent(), the event will become inactive and will no longer be recognized by the Swing framework. If consume() is not called, after the customized processing is finished, the event will be passed back to SIM and trigger any functionality associated with it.

Screens

A screen is the top level of a single PC screen display. The primary responsibility of screens is to interact with the navigation framework. The following is a break-down of the structure of screens. Use these coding guidelines when creating a customized screen. The example is the new CustomUINCreateScreen built to add the customized functionality of creating UINs without going through a transaction.

1. The UI screen must extend SimScreen. Usually, the only declaration is the panel that the screen delegates to. In some rare cases, variables may need to be declared to track some value at the screen level. Declare the constructor, which never contains parameters and simply adds the panel using the add() method available on the superclass.

   Implement the getScreenName() method. This should return the title of the screen to be displayed.

```
public class CustomUINCreateScreen extends SimScreen {
      private static final long serialVersionUID = 487712289865375658L;
      private CustomUINCreatePanel panel = new CustomUINCreatePanel();
      public CustomUINCreateScreen() {
          add(panel);
     }
      public String getScreenName() {
          return "UIN Create";
     }
 }
```

2. Implement a start() method. This method is called whenever the screen is navigated to. The start() method is called as the navigation takes place. Of interest are the two lines of code that are found in nearly every start() method. showMenu() causes the menu of the screen to be displayed. Passing in a parameter assigns a default button for the screen. The call to panel.start() triggers the startup of the visual panel. There can be a wide variety of functionality necessary in the startup of a screen, but it is important to remember that all functionality should be delegated to the panel (in the case of new screens) except for what is related to navigation. The start() method may throw an exception which will halt the navigation to the screen.

```
public void start() throws Throwable {
        showMenu();
        panel.start();
}
```

3. Implement the stop() method. This method is called whenever the screen is exited. Place code in here to clean up the state of the screen. This method can never throw an exception.

```
public void stop() {
        panel.stop();
}
```

4. Implement the resume() method if necessary. Resume is executed when the screen is navigated away from, but has not disappeared from the chain of screens. When the screen is returned to (such as from a sub-screen, this method is called to resume the screen). The resume() method should always call showMenu() to reset which buttons should be visible. The super-class SimScreen has a default implementation:

```
public void resume() {
    showMenu();
}
```

5. There are two new methods:

   - isStartable()
   - isStoppable()

   They are checked before start() and stop() are called respectively when screen navigation is taking place. Screen state validation code can be placed in these

methods. If isStartable() returns **false**, the user cannot navigate away from the screen. If isStoppable() returns **false**, the screen will not allow itself to be navigated away from:

```
public boolean isStartable() {
        return panel.isStartable();
}
```

6. Override the navigationEvent() method from the superclass. All menu buttons create NavigationEvent objects and pass them to the screen through this method. Each event has a command value that matches the button that pressed it. The implementation of this method should determine which button is pressed and delegate to the appropriate panel method:

```
public void navigationEvent(NavigationEvent event) {
       String command = event.getCommand();
       try {
            if (command.equals(SimNavigation.ADD_ITEM)) {
                panel.doAddItem();
            } else if (command.equals(SimNavigation.DELETE_ITEM)) {
                panel.doDeleteItem();
            } else if (command.equals(SimNavigation.DONE)) {
                panel.doHandleDone();
            }
       } catch (Throwable exception) {
            displayException(panel, event, exception);
       }
}
```

Features of the Screen

The SimScreen superclass contains several features that can be used by all subclasses. A brief listing is supplied in the table below.

Table: *Features of the Screen*

Feature	Description
assignFocusInScreen()	Assigns focus to the first button on the navigation menu.
navigate()	Navigates to the specified screen.
navigateLater()	Navigates to the specified screen on a separate thread.
removeFromScreenHistory()	Removes the screen from the navigational history.
removeNavButton()	Removes a navigational button from the menu.
displayException()	Displays the exception in the error dialog.

Extending Screens

To extend a screen, first create a class MyNewScreen that extends the intended screen.

For example:

```
public class MyNewScreen extends SimOriginalScreen
```

Next, find every occurrence of SimOriginalScreen in PC_MENU_ITEM.MENU column and PC_MENU_ITEM.MENU_ITEM_ACTION column within the database and replace it with MyNewScreen. Make sure you replace the entire classpath. After this is done, navigation will execute MyNewScreen instead of SimOriginalScreen, but all of the original functionality is still intact.

SimScreen

The SimScreen class has a new key method added to its API interface:

```
public ScreenPanel getScreenPanel();
```

This method returns the screen panel associated to the screen. This allows the customizing developer access to many of the methods of ScreenPanel in order to assist customization of the screen functionality.

Screen Panels

The panel is where all the visual elements of a single PC screen are located. All the Swing widgets are declared here as well as any functional logic that touches those widgets (getting and setting the widget properties and values). All logic not associated directly with the widgets is delegated to the model. Screen Panels are NOT customizable so all the below guidelines are intended for the scenario wherein a new workflow and screen is being created. The following shows some of the basic principles of designing a panel (see the Example: *CustomUINCreatePanel* in Appendix: Code Examples):

1. Declare the panel. The new panel should extend the ScreenPanel class.

   ```
   public class CustomUINCreatePanel extends ScreenPanel implements REventListener {
       private CustomUINCreateModel model = new CustomUINCreateModel();
   ```

2. Declare Editors and Tables. Declare the editors and tables that will be used within the panel. There should be an editor for each style of data currently used in SIM. See **Editors** for further detail. SimTable is the most common type of table used in SIM. SimTable should be placed within a SIMTablePane.

   ```
   private StockItemTableEditor stockItemTableEditor = new StockItemTableEditor();
   private SimTable stockItemTable = null;
   private SimTablePane stockItemPane = null;
   ```

3. Declare Constructor. The constructor of a panel is expected to not take any parameters and not throw any exceptions:

   ```
   public CustomUINCreatePanel() {
       initializePanel();
       initializeTable();
       layoutPanel();
   }
   ```

4. Initialize the Panel. The following is an example of initializing the property settings on editors and tables:

```
private void initializePanel() {
     stockItemTableEditor.addTableEditorListener(buildStockItemListener());
}
private void layoutPanel() {
    setContentPane(stockItemPane);
}
```

5. Initialize the Table. A s focus.

```
public void initializeTable () throws Throwable {
     stockItemTable.addRow(new CustomUINCreateWrapper());
     stockItemTable.setColumnSize("serialNumberCount", SimTable.LABEL_WIDTH);

}
```

6. Start the Panel. A start() method should be created so that the screen can call it to load the panel with information. Note that if a default focus editor is desired, the assignFocusInScreen() method should be called at the end of start passing in the component to receive focus.

```
public void start() throws Throwable {
     stockItemTable.addRow(new CustomUINCreateWrapper());
}
```

ScreenPanel

The ScreenPanel class return by the API of SimScreen is the superclass of all screen panels within SIM. This means that after subclassing the SimScreen class, the customization developer may get the ScreenPanel and use all the methods provided by that class.

There are two significant APIs that have been added:

```
public ScreenModel getScreenModel();
```

This method returns the screen model associated to the screen, which will have numerous other methods that can now be accessed directly by the screen subclass.

```
public SimTable getScreenTable();
```

If a detail line item table is present within the screen, this method will return that table. If no table is present, it will return null. Once the table is retrieved, the customization developer will then have access to all the methods of the table.

Editors

The table below is a brief list of common UI data editors used in SIM and what they do.

Table: *Editors*

Editors	Description
RCheckBoxEditor	A check box that can be selected or de-selected.
RComboBoxEditor	A combo box list of selections.
RDateFieldEditor	A single date field that can be edited by hand or through a calendar.
RDateRangeEditor	Two date fields that represent a start date and end date.
RDecimalFieldEditor	A text field that only allows decimal numbers to be entered.
RDisplayLabelEditor	An object displayer that does not allow editing.
RIntegerFieldEditor	A text field that only allows integer numbers to be entered.
RListEditor	A scrollable list of selections.
RLongFieldEditor	A text field that only allows long numbers to be entered.
RLongTextFieldEditor	A text field with an expansion dialog associated to it, often used for very long text fields that must take up limited screen space.
RMoneyFieldEditor	A text field that only allows money values to be entered.
RNumericIdEditor	A text field that edits numeric-only identifier strings.
RPasswordFieldEditor	A text field that allows the hidden entry of a password.
RPercentFieldEditor	A text field that only allows percent values to be entered.
RQuantityEditor	A text field that only edits Quantity values.
RRadioButtonEditor	An editor that allows a set of option buttons to be displayed and selected.
RSearchComboEditor	An editor that present a combo box of values but allows the user to trigger a search for additional values.
RSearchFieldEditor	A text field that allows the user to enter a value or search for one.
RTextAreaEditor	A large text area where large quantities of text can be edited.

Table: *Editors (cont.)*

Editors	Description
RTextFieldEditor	A single line text field where small quantities of text can be edited.

Initializing Editors

The editors are customized to contain numerous properties beyond those of the normal Swing widgets. Many of the customized properties are just short-cuts to other work. Here is a sample of some initialization in the InventoryAdjustmentFilterDialog. This includes setting displayers (which are responsible for formatting data), setting general size properties for the editors, and assigning an identifier to a text entry field (without an identifier, the editor will not activate).

Example: *InventoryAdjustmentFilterDialog*

```
private void initContent() {
   adjustmentEditor.setIdentifier(SimName.INVENTORY_ADJUSTMENT_NUMBER);
   adjustmentEditor.setEntryAlignmentLeft();
   adjustmentEditor.setSizeType(EditorConstants.MEDIUM);
   statusEditor.setDisplayer(new TranslatedObjectDisplayer());
   statusEditor.setSizeType(EditorConstants.LARGE);
   statusEditor.setEmptyType(RComboBoxEmptyType.ALL);
}
```

Editor Layout

To lay out editors in a simple grid, use an REditorPanel. Declare the rows and columns when instantiating and add editors to the panel. The REditorPanel handles all the layout details:

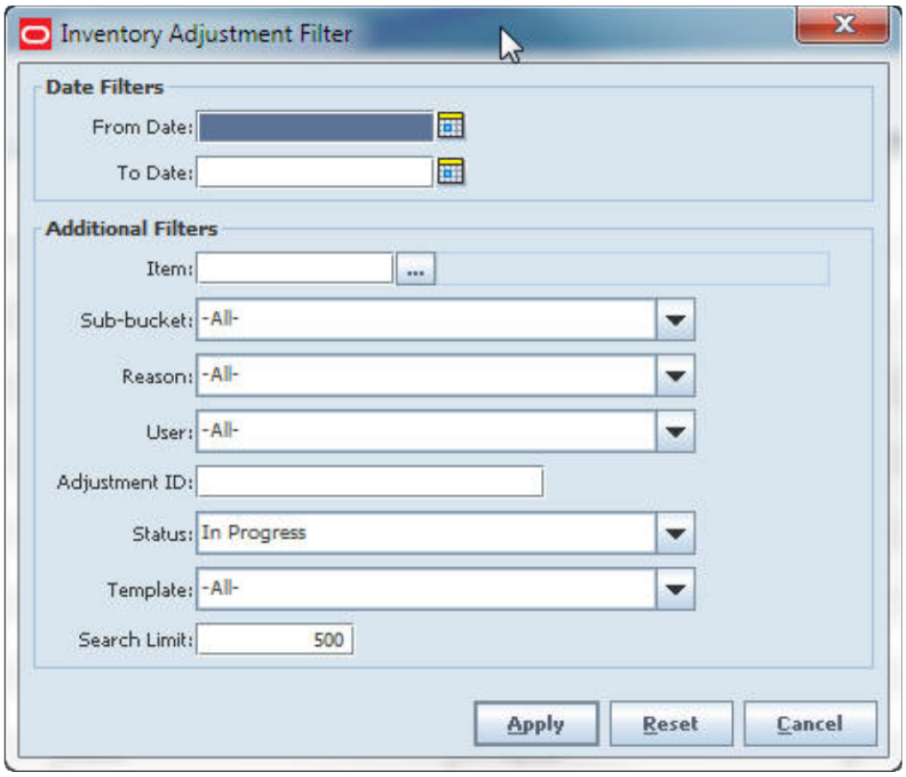
```
private void layoutContent() {
    REditorPanel miscFilterPanel = new REditorPanel(6);
    miscFilterPanel.setTitleBorder("Additional Filters");
    miscFilterPanel.add(itemEditor);
    miscFilterPanel.add(reasonEditor);
    miscFilterPanel.add(userEditor);
    miscFilterPanel.add(adjustmentEditor);
    miscFilterPanel.add(statusEditor);
    miscFilterPanel.add(searchLimitEditor);
}
```

Search Editor

A search editor is an editor that enables a user to enter data or find data. It consists of an editable text field that allows the user to enter an ID, a button that allows the user to search, and a non-editable text field that displays the description. The InventoryAdjustmentFilterDialog has a search editor for its item value that will be used as an example.

If an ID is entered, the editor automatically searches for the whole object. When the button is clicked, the screen navigates to a dialog where an object is chosen. Upon return, the data of the object is displayed:

Figure: ***Inventory Adjustment Filter Dialog***



Using a Search Editor

Do the following to declare and implement a search editor. Examples are from InventoryAdjustmentFilterDialog.

1. Declare the search editor.

```
private RSearchFieldEditor itemEditor =
SimEditorFactory.createItemVOSearchFieldEditor(false);
```

2. Set all the properties of the search editor. The following basic properties are required for a search editor to function:

Table: *Search Editor Properties*

Property	Description
identifier	The identifier is a component's name and can be used to reference the component uniquely.
search processor	The search processor is a class that implements the SearchProcessor interface. When the ID is altered in the entry field, this processor will be triggered to attempt to find the information.
search listener	The search listener is a class that implements the SearchListener interface. This class will be called when the **Search** button is pressed.

Example of setting up the properties of the search editor:

```
itemEditor.setIdentifier(SimName.ITEM_ID);
itemEditor.setSearchProcessor(new ItemVOSearchProcessor(allowNonInventoryItems));
itemEditor.setSearchListener(buildItemSearchListener());
```

3. Build a search listener. A search listener needs to be declared independently for each search editor that is used. This class must implement the SearchListener interface. Basically, it contains the method that is called once the data is found.

```
private ItemSearchListener buildItemSearchListener() {
    return new ItemSearchListener() {
        public void assignItem(ItemVO itemVO) {
            if (itemVO != null) {
                itemEditor.setData(itemVO);
            }
        }
    };
}
```

4. Build a search processor. A new search processor must implement the SearchProcessor interface. It must return a display for the entry area of the widget, a displayer for the value area of the widget, it must implement the searchById() method to find the appropriate object can put in a method that validates the data.

```
public class UserSearchProcessor implements SearchProcessor {
  private AttributeDisplayer entryDisplayer = new AttributeDisplayer("userName");
  private DualAttributeDisplayer valueDisplayer
              = new DualAttributeDisplayer("firstName", "lastName", StringConstants.SPACE);

public Object searchById(String userName) throws Exception {
    User user = ClientServiceFactory.getSecurityServices().readUser(userName);
    if (user == null) {
        UIException exception = new UIException(ErrorKey.USER_NOT_FOUND);
        exception.addValue(userName);
        throw exception;
    }
    return user;
  }
  public BasicDisplayer getEntryDisplayer() {
```

```
        return entryDisplayer;
    }
    public BasicDisplayer getValueDisplayer() {
        return valueDisplayer;
    }
    public Object validateData(Object data) {
        return data;
    }
}
```

SimTable

SimTable is an extension of JTable that allows the display and editing of cells and data. It contains advanced options not available with the standard table. It is also specifically designed to handle common design patterns in SIM. See the Javadoc of SimTable for a complete listing of available APIs.

Using a SimTable

To use a table within the screen panel, declare the SimTable variable globally within the panel class. Also declare a SimTablePane within the same panel because the SimTable will eventually need to be placed in the table pane in order to supply scrollbars. In the constructor of the panel, or some method that executes when the panel if first created, call a method to initialize your table and pane.

Example: *CustomUINCreatePanel*

```
private SimTable stockItemTable = null;
private SimTablePane stockItemPane = null;

public CustomUINCreatePanel() {
    initializeTables();
    initializePanel();
    layoutPanel();
}
```

When initializing the screen panel, initialize the table and table pane. This consists of instantiating a SimTableDef, which requires as a constructor parameter the class that represents each row in the table. Each attribute of the declared class is then independently added to the table definition. The below example defines a table where each row data object is a CustomUINCreateWrapper and it displays columns for item, item description and uin quantity.

Read the API carefully for SimTableDef and other classes used within SimTableDef. The first property of SimTableAttribute() is a display label, but the second is a column identifier. The identifier must match an attribute on the Class file passed into the constructor. For example, if the identifier is ABC, then the class must contain getABC() and setABC().

Example: *CustomUINCreatePanel*

```
private void initializeTables() {
      SimTableDef tableDef = createTableDef
(CustomUINCreateWrapper.class);
      tableDef.addNotEditableAttribute(
CustomUINCreateWrapper.UIN_COUNT_PROPERTY);

      tableDef.addTableAttribute(new SimTableAttribute("Item",
CustomUINCreateWrapper.STOCK_ITEM_PROPERTY,
new AttributeDisplayer("id"), stockItemTableEditor));
      tableDef.addTableAttribute(new SimTableAttribute(
"Item Description",
CustomUINCreateWrapper.DESCRIPTION_PROPERTY));
```

```
tableDef.addTableAttribute(new SimTableAttribute("UIN Qty",
CustomUINCreateWrapper.UIN_COUNT_PROPERTY,
new IntegerDisplayer(), new CustomUINCountTableEditor(
new UINPopupListener())));

      stockItemTable =
ClientTableFactory.createCustomUinTable(tableDef);
      stockItemTable.setColumnSize("serialNumberCount",
SimTable.LABEL_WIDTH);

      stockItemPane = new SimTablePane(stockItemTable);
}
```

The developer may then alter or set desired properties on the table. In the example workflow code above, we limit the size of the serial number count column to the width of its label. In the ItemLookupPanel code below, we show an example of other simple features of the table being applied. All the available options can be discovered by reading the SimTable API.

Example: *ItemLookupPanel*

```
itemTable.setTableEditable(false);
itemTable.setSingleRowSelectionMode();
itemTable.registerDoubleClickAction(this, ITEM_SELECTED);
```

Assigning data to the table is simply a matter of passing a collection of objects to be displayed. The setRows() method automatically replaces all the content of the table. The objects must all be of the class type declared in the table definition. There are other common methods for dealing with data such as addRow(), insertRow(), updateRow() and removeRow() that simply take the data object to be displayed in the row.

Example: *ItemLookupPanel*

```
itemTable.setRows(model.findItemVOs(searchFilter));
```

Example: *CustomUINCreatePanel*

```
stockItemTable.addRow(new CustomUINCreateWrapper());
```

Retrieving information from the table is as simple as using the various available get() methods such as getSelectedRow(), getSelectedRowData(), getAllSelectedRowData(), and getAllRowData(). Methods ending in “Data” retrieve the object represented by the row, so getSelectedRow() returns the row index, whereas getSelectedRowData() returns the row object. For example, see the doDeleteItem() and doHandleDone() methods in CustomUINCreatePanel in Appendix: Code Examples.

SimTableDefinition

A table definition is the core of how a table works. To create a table definition, simply declare a new SimTableDef object and specify what data object it will use in the data. CustomUINCreatePanel as an example of this class. In Table: *SimTableDefinition APIs* is a quick summary of APIs on SimTableDefinition class.

Table: *SimTableDefinition APIs*

API	Description
getDataClass()	The Class object of the data that each row in the table represents. This operates as the model for a row.
getIdentifier()	The unique identifier of the table. This API is used to track configuration information. The default implement is declaringClassName.className.
getTableAttributes()	A list of SIM table attributes that represent the attributes of each column within the table. For example, if there are 5 table attributes, there will be 5 columns.
add/removeTableAttribute()	Methods to add or remove a table attribute from the list of table attributes.
getSortAttributes()	Retrieves the list of default sort attributes. If the table configuration does not have any sort configuration settings, these sort attributes will be used to generate the initial sort configuration.
add/removeSortAttribute()	Methods to add or remove a sort attribute from the list of sort attributes.
getNonEditableAttributes()	Retrieves a list of attributes that do not require having a setter() method on the model in order to be considered editable in a table.
add/removeNonEditableAttribute()	Methods to add or remove a non-editable attribute from the list of table attributes.

In some cases, a single row of the table actually represents data that is originally from two different business objects. This is solved by using a wrapper class that can wrap the data objects and have one consistent API for the table row. Indeed, it is best that a wrapper be used in all circumstances in order to separate the table row API from the underlying object API.

SimTableAttribute Properties

A SimTableAttribute represents exactly one column within the SimTableDefinition. There are many different properties that can be assigned to the object. See the Example: *CustomUINCreatePanel* in Appendix: Code Examples for an example of using SimTableAttribute.

Table: *SimTableAttribute APIs*

Attribute	Description
Title	The title is displayed as the column header. This text label is actually a key into the translation tables. This key is also used as a unique identifier for the column.
Attribute	The second parameter is the attribute of the column. The value represents a getter and setter on the class type for retrieving the data. For example, if shortDescription is the attribute, then getShortDescription() and setShortDescription() should exist on the class type defined in the getDataClass() method. The attribute should always begin with a lower case letter. If a period appears within the attribute, then multiple levels of method calls with take place. For example, the attribute one.two.three would be converted into getOne().getTwo().getThree() when attempting to retrieve the column information from the data class. However, layered method calls like this are a good indicator that a wrapper or value object needs to be created.
Displayer	Each attribute value is examined by the table during instantiation and default displayers are assigned for the data type belonging to the attribute. If the attribute is a Quantity, then a QuantityDisplayer is assigned; if the attribute is a Boolean, then a BooleanDisplayer is assigned, and so on. If the attribute requires specialized formatting, then a displayer should be assigned to the attribute manually. For example, an AttributeDisplayer is assigned to the stockItem attribute and a UOMModeDisplayer is assigned to the unitOfMeasureMode attribute.
TableEditor	Each attribute value is examined by the table during instantiation and default table editors are assigned for the data type belonging to the attribute. If the attribute is a Quantity, then a QuantityTableEditor is assigned; if the attribute is a Boolean, then a BooleanTableEditor is assigned, and so on. If the attribute requires specialized editing, then a table editor should be assigned to the attribute manually. For example, a StockItemTableEditor is assigned to the stockItem attribute. The StockItemTableEditor is declared at the class level so it can modified by class code.
Editable	Should be assigned true if the column allows editing, false otherwise.

Displayers

Displayers are a hierarchy of classes responsible for formatting the information of an object into a displayable string. Each displayer must implement the method getDisplayText(). The package oracle.retail.sim.closed.swing.displayer in the client project contains numerous useful generic displayers.

Creating a New Displayer

Use the following procedure to create a new displayer as needed:

1. Check all previously existing displayers. Many of the generic displayers can handle formatting different objects (such as AttributeDisplayer, DualAttributeDisplayer and DefaultDisplayer).
2. Design the new displayer and extend the appropriate superclass (often AbstractDisplayer).
3. Implement the getDisplayText() methods.

Example: *StoreDisplayer*

```
public class StoreDisplayer extends AbstractDisplayer {
  private String separator = " - ";
  public String getDisplayText(Object object) {
      if (object == null) {
          return StringConstants.EMPTY;
      }
      if (object instanceof BuddyStore) {
          BuddyStore store = (BuddyStore) object;
          return store.getBuddyId() + separator + store.getBuddyName();
      }
      if (object instanceof Store) {
          Store store = (Store) object;
          return store.getId() + separator + store.getName();
      }
      if (object instanceof SimStore) {
          SimStore store = (SimStore) object;
          return store.getId() + separator + store.getName();
      }
      return object.toString();
  }
}
```

The Displayable Interface

There is an interface named Displayable with a single method toDisplayString(). This is intended to be implemented by wrappers, business objects and any other type of object that might need to have a display value, but which the toString() method is primarily reserved for debugging. Many classes in SIM already implement this interface. The UI framework is already set up to recognize and use Displayable. For example, if you drop several Displayable objects into an RComboBoxEditor, it automatically calls toDisplayString() to determine what to display.

Example: *StoreDisplayer*

```
public class Store extends BusinessObject implements Displayable {
    public String toDisplayString() {
        return id + " - " + name;
    }
    public String toString() {
        StringBuilder buffer = new StringBuilder();
        buffer.append("Store: [");
        buffer.append("Id: ").append(getId());
        buffer.append(";  Name: ").append(getName());
```

```
        buffer.append(";  Language: ").append(getLanguage());
        buffer.append(";  Country: ").append(getCountry());
        buffer.append(";  Currency: ").append(getCurrencyCode());
        buffer.append(";  Sim flag: ").append(getSimFlag());
        if (timezone != null) {
            buffer.append(";  Timezone: ");
            buffer.append(timezone.getDisplayName());
        }
        buffer.append("]");
        return buffer.toString();
    }
}
```

TableEditors

Like the editors designed to be used on screens, table editors are advanced widgets designed for use within table cells. The SimTable uses the SimTableEditor interface to control editing within SimTable cells.

- CustomUINCreateDialogTableEditor is included in the example code set as an implementation of a normal table editor.
- CustomUINCountTableEditor is included in the example code set as an implementation of a table editor that opens a dialog box.

There are many generic table editors located within the client project in oracle.retail.sim.closed.swing.tableeditor package. If these do not meet requirements, there are SIM-specific table editors within the client project in the package oracle.retail.sim.close.swingclient.tableeditor. If these do not meet requirements, a custom table editor must be written.

Note

No specific documentation on creating table editors is planned for this document. Table editors are complicated objects that require very precise design. Design and code review for any new table editors should be performed by a senior developer.

Wrappers

When creating a table definition, the getDataClass() is used to specify what object is being displayed. Often, the definition of a row does not exactly match a business object. Sometimes a row might require two business objects. Sometimes the API of the business object does not match what is desired for display. Design of the business layer and its objects should be done strictly with the idea of capturing functional requirements, relationships and behavior, and not necessarily with how the business layer and its objects are gathered and displayed on the screen.

Wrappers are PC client layer objects that wrap one or more business objects into a single API that the table can easily access and modify. This creates an isolated layer between the table and business object where the UI client code can live that does not belong to the behavior of the business object.

The Example: DirectDeliveryDetailPanel is a poor example, as it directly uses the business object ReceiptLineItem:

Example: *DirectDeliveryDetailPanel*

```
SimTableDef tableDef = createTableDef(ReceiptLineItem.class);
```

The Example: DirectDeliveryDetailPanel is a good example, as it uses a wrapper for the direct delivery line item that represents an interface to the table:

Example: *DirectDeliveryDetailPanel*

```
SimTableDef tableDef =
       createTableDef(DirectDeliveryLineItemWrapper.class);
```

A wrapper simply wraps another object or objects and has methods that are designed for the table and its column. The wrapper determines what to do with the code and passes it on to the business objects. Note how the wrapper contains both a stock item and the serial number values to add. This makes it convenient to create one table row API that accesses both objects:

Example: *CustomUINCreateWrapper*

```
public class CustomUINCreateWrapper {
    public static final String STOCK_ITEM_PROPERTY = "stockItem";
    public static final String DESCRIPTION_PROPERTY = "description";
    public static final String UIN_COUNT_PROPERTY = "serialNumberCount";
    private StockItem stockItem;
    private List<SerialNumberValue> serialNumbers = CollectionUtil.newArrayList();
    public StockItem getStockItem() {
        return stockItem;
    }
```

To customize the attributes of a wrapper within the system appropriately, subclass the wrapper object in question and add an attribute to the sub-class only:

```
public class MyCustomUinCreateWrapper extends CustomUINCreateWrapper {
    private String season;
    public String getSeason() {
        return season;
    }
    public void setSeason(String season) {
         this.season = season;
    }
}
```

Next, subclass the ClientWrapperFactoryImpl to override the creation method and replace it with your own. Once configuration is altered to use the custom factory, everywhere in the code where a CustomUINCreateWrapper would have been created/instantiated, a MyCustomUinCreateWrapper is instantiated instead:

```
public class MyClientWrapperFactoryImpl extends ClientWrapperFactoryImpl {

    public CustomUINCreateWrapper createCustomUINCreateWrapper() {
        return new MyCustomUinCreateWrapper();
    }
}
```

To configure to use the new customization, simply modify the client.cfg file with the classpath of your new code:

```
GUI.WRAPPER_FACTORY=oracle.retail.sim.client.core.MyClientWrapperFactoryImpl
```

Value Objects

Value Objects are end-to-end (db to client) objects that represents a summary or partial view of one or more objects. Value objects should be designed as clean representations of the data matching exactly where they are going to be used. Value objects do not execute rules or contain setters. They contain only getter methods and doSet() methods that should only be used by the DAO layer. They also tend to declare only basic data they need and not contain complete objects. The purpose of the value object is to transmit less data between the server and clients when it is known that no editing of the data will be needed. A value object can be used within the code of a table replacing the wrapper. However the rowsof the table must not be editable as the value object is considered unchangeable. The ItemVO is a good example of this pattern. This value object is designed to primarily be used on the ItemLookupPanel (which does not edit any information) and so its information matches the screen.

Example: *ItemVO*

```
public class ItemVO implements Serializable, ItemDescription {
    private String id = "";
    private String shortDescription = "";
    private String longDescription = "";
    private SupplierVO supplierVO = null;
    private String departmentName = "";
    private String className = "";
    private String subclassName = "";
    private boolean isRanged = true;
    private ItemType itemType = ItemType.ITEM;
    public ItemVO(String id) {
        this.id = id;
    }
    public String getId() {
        return id;
    }
    public String getShortDescription() {
        return shortDescription;
    }
// Remainder of code…}
```

In this example, note how the ItemVO contains departmentName instead of an ID or the full merchandise hierarchy node. This is because the name is the only item required where the VO is used. The same with adding the flag isRanged(), which a normal item does not contain. This makes the item much smaller and easier to transmit across the service call.

Triggering User Interface Events

Sometimes, a user might want to perform an action and execute some logic when an event occurs within the components on the screen. Under these circumstances, there is a specific framework in place to accomplish this. Whenever possible, avoid using standard Swing listeners to accomplish these kinds of tasks, instead using REventListeners and RActionEvents:

Regular Editor Actions

Do the following to receive and process actions. Almost all actions of this nature take place within the Panel code.

1. Register an action on an editor.

   Every editor within the system implements RetailEditor that contains the method registerAction(). There are two parameters: the listener and the command. When the contents of the editor change, the listener is notified with the command. Actions are normally registered within the initializePanel() method of the Panel.

   Example: *DirectDeliveryCreatePanel*

```
private RSearchFieldEditor itemEditor =
                SimEditorFactory.createNonRangingItemSearchFieldEditor("Item");
private RSearchFieldEditor supplierEditor =
SimEditorFactory.createSupplierSearchFieldEditor();
private static final String ITEM_MODIFIED = "Item.modified";
private static final String SUPPLIER_MODIFIED = "Supplier.modified";

private void initializePanel() {
    // Additional Code Here
    itemEditor.registerAction(this, ITEM_MODIFIED);
    supplierEditor.registerAction(this, SUPPLIER_MODIFIED);
    // Additional Code Here}
```

2. Receive and parse the action.

   All RActionEvents that are generated are sent to the method performActionEvent(). The method should parse out which action took place in the standard pattern and delegate to a method to execute the logic. This pattern is preferable to creating inner class listeners on the fly because there is one centralized place within the panel to find where all actions are being delivered. This pattern also aids debugging as well.

   Example: *DirectDeliveryCreatePanel*

```
public void performActionEvent(RActionEvent event) {
    String command = event.getEventCommand();
    try {
        if (command.equals(TYPE_SELECTED)) {
            doRadioSelectionModified();
        } else if (command.equals(ITEM_MODIFIED)) {
            doItemModified();
        } else if (command.equals(ITEM_SUPPLIER_MODIFIED)) {
            doItemSupplierModified();
        } else if (command.equals(SUPPLIER_MODIFIED)) {
            doSupplierModified();
        } else if (command.equals(PO_MODIFIED)) {
            doPurchaseOrderModified();
        } else if (command.equals(PACK_ITEM_MODIFIED)) {
            doPackItemModified();
        } else if (command.equals(PACK_MODIFIED)) {
            doPackModified();
        } else if (command.equals(PACK_SUPPLIER_MODIFIED)) {
            doPackSupplierModified();
        }
    }
 catch (Throwable e) {
        displayException(e);
    }
}
```

3. Implement the action logic.

   Implement the private method that executes the logic associated with the editor.

Table Editor Actions

Editors within tables can be assigned listeners in order to receive events, but they are handled in an entirely different manner.

Use the addTableEditorListener() method to add a listener to the table editor. A table editor can either be declared as a class variable so it is handy or can be retrieved from the table itself based on the type of property that is being modified.

Build a listener. Use a build<name>Listener method to get the code separated into a convenient method rather than declaring inline where it might obfuscate what is taking place. In the following example, every time the stock item value changes within the table, the validateEnabledState() method is called within the panel:

Example: *ReturnDetailPanel*

```
private StockItemTableEditor stockItemTableEditor = new StockItemTableEditor();
private void initializePanel() {
   stockItemTableEditor.addTableEditorListener(buildStockItemListener());
}
private SimTableEditorListener buildStockItemListener() {
    return new SimTableEditorListener() {
        public void performTableEditorEvent(SimTableEditorEvent event) {
            validateEnabledState();
        }
    };
}
```

Sub-Class a Screen

Follow the guidelines for **Extending Screens**.

Override the start() Method

Every screen has a start() method, simply override the method of the parent class. Call super.start() to guarantee the screen is started correctly, and then add the method to remove the column.

Example: *Override the start() Method*

```
private void removeColumn() {

  SimTable table = getScreenPanel().getScreenTable();

  int index = table.findColumnIndex(DirectDeliveryProperty.QUANTITY_EXPECTED_UOM);

  table.removeColumn(table.getColumnModel().getColumn(index));

}
```

Screen Models

Screen models hold onto the state of data or information and interact with the business layer of the system for the panel of UI widgets. Screen Models cannot be subclassed or customized. This section is meant to inform how to properly create a screen model for new workflows. The following is an example of building a screen model (using the CustomUINCreateModel.java found in the example code):

1. Declare the model. The model must extend SimScreenModel. Declare any variables, usually the data the model represents. No constructor is necessary for a model as there are never any parameters passed into a model constructor.

```
public class CustomUINCreateModel extends SimScreenModel {
    private String CUSTOM_UIN_SERVICES_KEY = "CUSTOM_UIN_SERVICES_KEY";
```

2. Add all the functional methods required by the panel to communicate with the server or manipulate data on a logic level. Here is an example of common methods found in a model.

```
public void saveSerialNumbers(List<CustomUINCreateWrapper> wrappers) throws Exception {
 List<CustomSerialNumber> serialNumbers = CollectionUtil.newArrayList();
 for (CustomUINCreateWrapper wrapper : wrappers) {
     for (SerialNumberValue value : wrapper.getSerialNumbers()) {
         CustomSerialNumber serialNumber = new CustomSerialNumber();
         serialNumber.setId(value.getUINDetailId());
         serialNumber.setStatus(value.getStatus());
         serialNumbers.add(serialNumber);
     }
 }
 getCustomUINService().moveSerialNumbersToInStock(serialNumbers);
}
```

Features of the Model

The SimScreenModel superclass contains several features that can be used by all subclasses. A brief listing is supplied in **Table:** Model Features.

Table: Model Features

Feature	Description
getUser()	Retrieves the current user logged into the application.
getStore()	Retrieves the current store the user is logged into.
getAllowedStores()	Retrieves a list of stores the user is allowed to access.
getTimeZone()	Retrieves the time zone of the current store.
hasPermission()	Returns true if the user has the specified permission.
hasDataPermission()	Returns true if the user has the specified permission for a particular piece of data.
releaseLock()	Release an activity lock on the transaction being used.
confirmLock()	Confirms the current user still holds the activity lock on the transaction being used.
obtainLock()	Obtains an activity lock for a user on a transaction.

Dialog Windows

There are a few different usages of dialog boxes in SIM. The most common usage is to display errors, messages and confirmations during the workflow process. Other usages include pop-up search dialogs (item & supplier) and entering data.

Error & Message Dialogs

The developer never instantiates or creates an error or message dialog. These types of dialogs are handled entirely by the framework. Models cannot display errors because they only represent logic, so all exceptions must be propagated to the Panel or Screen level where helper methods exist to display the exceptions. The displayMessage() and displayWarning() methods take a string and display a message window. These last two methods are only available at the panel level as the screen level is thin enough to not need warnings or messages.

The Example: *ItemTicketListPanel* in ItemTicketListPanel uses several different helpers all in one method:

Example: *ItemTicketListPanel*

```
private void printTickets() throws Exception {
       try {
           List<ItemTicketWrapper> wrappers
                    = itemTicketTable.getAllSelectedRowData();
           ReportResponse response = model.printTickets(wrappers);
           if (response != null) {
               displayError(response.getMessage(), response.getMessageValue());
           } else {
               displayMessage(ReportMessageText.ITEM_TICKET_PRINTED);
            }
        } catch (NoPrinterDefinedException noPrinterException) {
               displayError(ReportMessageText.NO_STORE_PRINTERS);
         } catch (BusinessException exception) {
               displayException(exception);
          } finally {
               itemTicketTable.setRows(model.findItemTickets());
          }
  }
```

Confirmation

Confirmation dialogs are Ok/Cancel or Yes/No dialogs that allow the user to make decisions. Confirmation dialogs are not instantiated. A utility class (RConfirmUtility) is supplied to make the display of confirmation dialogs easier. The confirm() method returns **true** if the option was confirmed; **false** if it was not. The Example: *ReturnDetailPanel* and the Example: *StoreSequenceListPanel* show some different usages of the confirmation utility:

Example: *ReturnDetailPanel*

```
protected boolean cancelReturn() throws Exception {
    if (RConfirmUtility.confirmWithOkCancelType("Delete Return Confirmation",
               SimClientErrorKey.RETURN_MISSING_QUANTITY)) {
        model.cancelReturn();
        return true;
    }
    return false;
}
```

Example: *StoreSequenceListPanel*

```
public void handleApplyClassList() throws Exception {
    if (model.isSequenceCoherent()) {
        if (RConfirmUtility.confirm("Confirmation",
```

```
                                    SimClientErrorKey.SEQUENCE_GENERATE_LOCATION_CONFIRM)) {
                LocationArea locationArea = LocationArea.BACKROOM;
                if (RConfirmUtility.confirm("Select Area",
                        SimClientErrorKey.SEQUENCE_SHOPFLOOR_OR_BACKROOM, "Shopfloor", "Backroom")) {
                    locationArea = LocationArea.SHOPFLOOR;
                }
                try {
                    model.applyClassList(locationArea);
                } catch (BusinessException exception) {
                    displayException(exception);
                }
                locationTable.setRows(model.findLocations());
            }
        }
    }
```

Pop-up Windows

All pop-up windows have RDialog as a super-class. If the customization includes created a new dialog to enter data, then by using RDialog as the super-class, the customized code can access several additional useful methods connected to SIM's framework. Existing dialogs cannot be customized.

Building Wireless Forms

The SIM Wireless handheld client is a Wavelink application. All wireless clients (users in the store) use handheld physical devices to talk to a wireless container (sometimes called the wireless server). Most of the SIM application can be accessed through these handheld devices. In certain areas such as stock counts and product groups, some administrative tasks must be done on the PC as these tasks cannot be performed by the handheld device.

Wireless Application Architecture

The client architecture is broken into five layers, only three of which are worked on during application development:

Wireless Framework

This is the wireless framework application consisting of the Wavelink code that starts and executes as a server and handles the actual communication protocol back and forth to the handheld devices.

Form_<name>

The framework loads and displays forms on the handheld device. The framework then receives actions from these forms back at the server – similar to a Web page. These forms are not coded, but are generated from xml files (also similar to Web pages). A form's xml file will be named **screen_<name>** and located in its own directory in the wireless project with the path `generator.screens`.

EventHandler_<name>

AbstractEventHandlers are generated along with the forms to receive actions. The EventHandler_<name> class extends the abstract class and actually implements the actions that can be received from the form. The EventHandler classes are coded by the developer to handle the logic for the handheld device.

<functional area>Utility

Often, an EventHandler communicates with the rest of the SIM system by using logic available through a utility that covers common logic within a functional area.

EJB Service

The EJBs to the regular SIM services are accessed from EventHandlers or Utilities by using the client-side EJB services (either by specifically instantiating or by using ClientServiceFactory).

Forms

Forms contain the page information that is sent back and forth to the actual device.

Event Handler

When a form is created, an AbstractEventHandler_<name>.java file must exist as well. The developer must then code an EventHandler_<name>.java file that matches the abstract handler. This section will outline how an EventHandler relates to a form and some of the general methods that are available to an EventHandler. The Example: *Screen_InventoryAdjustmentNormalItem* is an example of a form designed in an .xml file for entering an item on an inventory adjustment:

Example: *Screen_InventoryAdjustmentNormalItem*

```
<Screen name="InventoryAdjustmentNormalItem">
  <LogicalScreen>
    <field name="itemId" type="string" length="21" />
    <field name="itemDesc" type="string" length="42" />
    <field name="reason" type="string" length="21" />
    <field name="unavailableLabel" type="string" length="10" />
    <field name="unavailableQty" type="string" length="5" />
    <field name="qtyLabel" type="string" length="15" />
    <field name="qty" type="string" length="5">1</field>
    <field name="uom" type="string" length="5" />
    <field name="packSize" type="string" length="5" />
  </LogicalScreen>
<PhysicalScreens deviceclass="dnw">
    <PhysicalScreen seq="0">
      <label y="0" x="0" height="1" width="21"
            style=".heading1">$[wireless.inventoryAdjustment]</label>
      <label y="2"  x="0"  height="1" width="21" name="itemId" field="itemId"></label>
      <label y="3"  x="0"  height="2" width="21" name="itemDesc" field="itemDesc"></label>
      <label y="5"  x="0"  height="1" width="21" name="reason" field="reason"></label>

      <label y="6"  x="0"  height="1" width="10" name="unavailableLabel"
             field="unavailableLabel"></label>
      <label y="6"  x="10"  height="1" width="6" name="unavailableQty"
             field="unavailableQty"></label>
      <label y="6"  x="16"  height="1" width="5" name="unavailableUnits" />

      <label y="8"  x="0" height="1" width="9" name="qtyLabel" field="qtyLabel"></label>
      <input y="8"  x="10" height="1" width="6"  name="qty" field="qty" seq="0"
             acceptScan="false" validateKeyEventOnScan="false"/>
      <label y="8"  x="16" height="1" width="5"  name="uom" field="uom" />

      <label y="9"  x="0"  height="1" width="10" name="packSizeLabel" />
      <input y="9"  x="10" height="1" width="6" name="packSize" field="packSize"
             seq="1" acceptScan="false" validateKeyEventOnScan="false"/>

      <scan/>

      <cmdkey y="0" x="0" width="0" height="0" key="&toggle;" name="toggle"
```

```
            action="callMethod" target="doToggle"/>
      <cmdkey y="0" x="0" width="0" height="0" key="&exit;" name="Exit"
             action="callMethod" target="doExit" />
  </PhysicalScreen>
</PhysicalScreens>
</Screen>
```

The API of the AbstractEventHandler that was created to match the form is shown in the example below:

Example: *AbstractEventHandler_InventoryAdjustmentNormalItem*

```
abstract public class AbstractEventHandler_InventoryAdjustmentNormalItem extends SimEventHandler {
    abstract protected void onFormOpen();
    abstract protected void onFormClose();
    abstract public void onScan(String data);
    abstract public boolean qty_OnChange(String newValue);
    abstract public boolean qty_OnExit(String newValue);
    abstract public boolean packSize_OnChange(String newValue);
    abstract public boolean packSize_OnExit(String newValue);
    abstract public void doToggle();
    abstract public void doExit();
}
```

OnFormOpen() and OnFormClose() must exist for each AbstractEventHandler regardless of the form:

- OnFormOpen() is executed before the form is displayed to the handheld device. The code that populates the form with data should be placed in this method.
- OnFormClose() is executed when the form is removed from the handheld device.

The onScan() method matches the `<scan/>` tag in the xml, which indicated a row on the form into which to scan a value. When a value is scanned by the device, the information is passed to the onScan() method as a data parameter. The developer can implement this method in the EventHandler to process the scanned data (in this case, the barcode of an item).

Both qty_OnChange() and qty_OnExit() were created by the `<input>` tag with the field=qty set within that tag. Since qty was entered as the field, these two methods exist to handle processing when a quantity is entered.

Both packSize_OnChange() and packSize_OnExit() were created by the `<input>` tag with the field =packSize set within that tag. Since packSize was entered as the field, these two methods exist to handle processing when a pack size is entered.

The doToggle() method was created by the `<cmdkey>` tag based on the target=value section of the tag. Since doToggle was entered as the target value, this method was created. A cmdkey displays as an option on the screen. The method is executed in the EventHandler when the option is chosen on the screen. This is the same with the doExit() based on its `<cmdkey>` tag.

SimEventHandler

Each AbstractEventHandler extends from SimEventHandler, a superclass that contains methods for common tasks. These methods should always be used when these tasks must be performed. There are methods for the following:

- Assigning data to forms
- Reading data from forms
- Displaying alerts
- Displaying exceptions
- Checking locks
- Releasing locks
- Showing specialized screens (barcode, Yes/No choice, text input)
- Navigating to other forms

YesNoEventHandler

YesNoEventHandlers are simple choice windows that display a choice and allow the user to pick **Yes** or **No**.

Example: *RequestCancelYesNoHandler*

```
public class RequestCancelYesNoHandler extends SimYesNoHandler {
  public void performYes(IApplicationForm currentForm) throws Exception {
    try {
      ItemRequest itemRequest = ItemRequestWirelessUtility.getContext().getItemRequest();
      if (itemRequest.getLineItems().size() > 0) {
          ItemRequestWirelessUtility.doSave(itemRequest, false);
      }
      currentForm.gotoForm(ItemRequestWirelessKeys.SCREEN_MENU);
      } catch (BusinessException be) {
          currentForm.gotoForm(ItemRequestWirelessKeys.SCREEN_MENU);
      } catch (Exception e) {
          handleException(e, currentForm);
      }
  }
  public void performNo(IApplicationForm currentForm) throws Exception {
    try {
        currentForm.gotoForm(ItemRequestWirelessKeys.SCREEN_SUMMARY);
    } catch (Exception e) {
        handleException(e, currentForm);
    }
  }
  public String getTitle() throws Exception {
    return getText(ItemRequestWirelessUtility.getTitleKey());
  }
  public String getMessage() throws Exception {
    String id = ItemRequestWirelessUtility.getContext().getItemRequest().getId();
    return getMessage(ItemRequestWirelessKeys.MESSAGE_EXIT_CONFIRM, id);
  }
}
```

The SimYesNoHandler superclass that all YesNoHandlers should extend contains most of the same helper methods as the SimEventHandler class. This means that such functionality as translating text and handling exceptions should always use these helper methods.

The SimYesNoHandler also has the implemented code that returns the Yes and No choice labels for the screen, so the user only has to implement the following four methods for each new YesNoHandler:

- performYes() is executed when the user chooses the **Yes** option.
- performNo() is executed when the user chooses the **No** option.
- getTitle() returns the title to display at the top of the form.
- getMessage() return the query text to display on the form.

Wireless Context

A context represents a repository of data entered or being altered for a particular functional area within the user session. This context is carried in the repository to make it readily accessible between different forms. The InventoryAdjustmentContext is used as an example to trace some of the usages of context. Note that the context object itself simply has a set of data variables.

Example: *InventoryAdjustmentContext*

```
public class InventoryAdjustmentContext {
    private InventoryAdjustment inventoryAdjustment;
    private InventoryAdjustmentReason lastAdjustmentReason = null;
    private String lastScannedUIN = null;
    private Quantity scanEnteredQty = null;
    private Quantity computedQty = null;
    private boolean takeFromUnavailableBucket = false;
    private int currentUINIndex = 0;
    public InventoryAdjustment getInventoryAdjustment() {
        return inventoryAdjustment;
    }
    public void setInventoryAdjustment(InventoryAdjustment inventoryAdjustment)     {
        this.inventoryAdjustment = inventoryAdjustment;
    }
    public boolean isTakeFromUnavailableBucket() {
        return takeFromUnavailableBucket;
    }
    public void setTakeFromUnavailableBucket(boolean takeFromUnavailableBucket)     {
        this.takeFromUnavailableBucket = takeFromUnavailableBucket;
    }
    // Other Getters & Setters
}
```

Access to the context is through the utility for the functional area, which contains a set of static helper methods to create, retrieve and remove the context. Note that the actual context itself is stored within the SimWirelessRepository.

Example: *InventoryAdjustmentWirelessUtility*

```
public static InventoryAdjustmentContext getContext() {
    return (InventoryAdjustmentContext)
       SimWirelessRepository.getValue(InvAdjustmentWirelessKeys.CONTEXT);
}

public static InventoryAdjustmentContext createContext() {
    InventoryAdjustmentContext context = new InventoryAdjustmentContext();
    SimWirelessRepository.setValue(InvAdjustmentWirelessKeys.CONTEXT, context);
    return context;
}
public static void removeContext() {
   SimWirelessRepository.removeValue(InvAdjustmentWirelessKeys.CONTEXT);
}
```

Wireless Utilities

Wireless Utilities are static classes that contain numerous helper methods to execute business logic, such as in the examples for context in which the context is created or retrieved, or in the following example where a new inventory adjustment was created. There is usually only one <name>WirelessUtility for each functional workflow on the handheld device, but some large areas may have more than one.

The Wireless Utility contains public static helper methods to perform business logic for EventHandlers. Wireless Utilities do not contain any data variables at the class level as the utility does not store state of its data. That responsibility is handled by the context. Example: *InventoryAdjustmentWirelessUtility*. The below example code shows a utility method for persisting the inventory adjustment:

Example: *InventoryAdjustmentWirelessUtility*

```
public static void persistInventoryAdjustment(IApplicationForm currentForm) {
    InventoryAdjustment invAdjustment = getContext().getInventoryAdjustment();
    try {
        if (invAdjustment.isCoherent()) {
            ClientServiceFactory.getInventoryAdjustmentServices()
                .processInventoryAdjustment(invAdjustment);
        alert(currentForm, getTitleKey(), InvAdjustmentWirelessKeys.MESSAGE_ADJUSTMENT_COMPLETE,
               ItemWirelessKeys.SCREEN_SCAN_BARCODE, false);
        }
     } catch (BusinessException businessException) {
        String message = getMessage(businessException.getMessage(),
                                    businessException.getParameters());
        alert(currentForm, getTitleKey(), message, getItemScreen(), true);
     } catch (Exception e) {
        handleException(e, currentForm);
     }
}
```

Form Paging

The ScreenPageManager and PageableEventInterface are used to create selection screens that go on for more than one visual screen worth of information on the handheld device. A **next** and **previous** command option exists on the base of the form to scroll through the pages.

The ScreenPageManager is a wireless framework class that handles a lot of the formatting and displaying of the options on a form and controls going forward and backward through the pages. Developers should never modify this class.

Primarily, developers access this information by having an EventHandler for a form implement the PageableEventInterface. The ScreenPageManager uses this defined API to control the paging.

Example: *PageableEventInterface*

```
public abstract Map getMenuItemMap();
public abstract String getScreenName();
public abstract IApplicationForm getCurrentForm();
public abstract String[] getPageFieldNames();
public abstract String[] getSpecialFieldValues();
```

The ItemRequestSelectRequest form displays a list of requests for the user to select from. These requests might not all fit on one form, so this EventHandler implemented PageableEventInterface. In the implementation of getMenuItemMap(), a Map of ItemRequests is returned from the Context to be displayed as the list of options to select. The getScreenName() method returns the title of the screen.

Example: *EventHandler_ItemRequestSelectRequest*

```
public Map getMenuItemMap() {
    return (Map) SimWirelessRepository.getValue(ItemRequestWirelessKeys.ORDER_MAP);
}
public String getScreenName() {
    return ItemRequestWirelessKeys.SCREEN_SELECT_REQUEST;
}
```

The method getCurrentForm() is implemented by SimEventHandler, so all EventHandlers automatically implement that method. The getPageFieldNames() method returns a list of names to assign to the options displayed on the screen. So in the following example, nine options (or item requests) are displayed with **order0** through **order9** assigned as their name:

Example: *EventHandler_ItemRequestSelectRequest*

```
private static final String[] orderFieldNames = { "order0", "order1", "order2",  "order3",
"order4", "order5", "order6", "order7", "order8", "order9" };

 public String[] getPageFieldNames() {
    return orderFieldNames;
}

 public String[] getSpecialFieldValues() {
    return null;
}
```

When the option is selected, the Wavelink framework calls back to a method named after the options names assigned in getPageFieldNames(). In this case, selectOrder0(), selectOrder1(), and so on. This is where the developer can place code to handle the selection. There are no examples of getSpecialFieldValues(). This method returns null in all EventHandlers that implement the interface.

Example: *EventHandler_ItemRequestSelectRequest*

```
public void selectOrder0() {
    selectOrder(0);
}
public void selectOrder1() {
    selectOrder(1);
}
// Rest of orderFieldNames…public void selectOrder9() {
    selectOrder(9);
}
```

Once the methods are defined, starting the page display is easy. In the onFormOpen() method, if there is a list available to display selections for, then retrieve the page manager and call initializeScreen(). This sets up and displays the first page of selection options. It should be the last method executed in onFormOpen():

Example: *EventHandler_ItemRequestSelectRequest*

```
protected void onFormOpen() {
  try {
      LogService.debug(this, getClass().getSimpleName() + ".onFormOpen()");
      setFormText(FIELD_INSTRUCTIONS, getLabel("Select Item Request"));
      if (getMenuItemMap() == null || getPageManager() == null) {
          initializeOnEntry();
      } else {
          getPageManager().initializeScreen();
      }
  } catch (Exception e) {
```

```
      handleException(e);
  }
}
```

Customizing Wireless Forms

This section contains some tips on customizing wireless code. Customizing wireless code requires modifying source code in SIM. This needs to be handled with care.

Coding Guidelines

- The wireless clients should always access services through the ClientServiceFactory class.
- Do not alter existing context classes. Create new ones instead and store in SimWirelessRepository.
- When at all possible, design the modifications to be new forms or replacements to current forms instead of modifying current forms.
- Similar to forms/screens, attempt to write new utility classes rather than modifying existing ones.

By following the guidelines above, the only code that should need to be modified in SIM is the code that directs one form to a newly designed form – keeping the amount of rework at a minimum for future releases.

Creating a New Context

If you need to store additional state data during a session, create an entirely new context to store the information. The newly created context must then be placed in the SimWirelessRepository. Next, create a custom wireless utility to handle interaction with the context, including placing the context in the SimWirelessRepository.

```
public class MyCustomContext() {
    { . . . . } // Context code.
 }
```

Creating a New Utility

If new functionality is desired in the wireless application that needs to access a new context or access new services, then the developer should create an entirely new utility to use from the eventhandlers to execute this code.

If you need ready access to a utility method or need to override the functionality of a utility method, then subclass the utility in question with your new utility as shown in Example: *MyCustomUtility*:

Example: *MyCustomUtility*

```
public class MyCustomStockCountUtility extends StockCountWirelessUtility {
    public static MyCustomContext getContext() {
      return (MyCustomContext) SimWirelessRepository.getValue(MyCustomContextKeys.CONTEXT);
    }
    public static MyCustomContext createContext() {
        MyCustomContext context = new MyCustomContext ();
        SimWirelessRepository.setValue(MyCustomContextKeys.CONTEXT, context);
        return context;
    }
    public static void removeContext() {
       SimWirelessRepository.removeValue(MyCustomContextKeys.CONTEXT);
    }
    [ . . . . ] // New Utility Code
}
```

Altering Code in an EventHandler

Because of the manner that the Wavelink code functions, eventhandlers cannot be easily replaced or sub-classed. Instead, the eventhandler must be removed from the JAR, re-coded, and placed in a custom JAR earlier in the classpath. These modifications are not guaranteed to function correctly with newer releases.

Altering Code in a Form

Altering a Wavelink form is a complicated process:

1. Alter the form_<name> source code.
2. Alter the form_dnw_<name> source code.
3. Alter the AbstractEventHandler_<name> source code.
4. Alter the EventHandler_<name> source code.

All of these files must be placed back in the custom JAR that is first in the classpath.

Significant Classes to Understand

Before attempting to modify or create new handheld Wavelink forms, read all documentation supplied by Wavelink. In addition, the following is a list of critical framework classes specific to SIM:

- CommandListHelper
- CommandListInterface
- InputTextInterface
- LocaleWirelessUtility
- PageableEventInterface
- PrintSelectInterface
- ScreenPageManager
- SimBasicHandler
- SimEventHandler
- SimWirelessRepository
- SimYesNoHandler
- WirelessExceptionManager
- WirelessPrintService
- WirelessUtility
- WirelessValidation
- YesNoEventInterface

Exceptions and Logging

This section covers exceptions and logging in SIM so that if code is customized, the exception handling framework can be utilized.

Exceptions

SimServerException

The SimServerException class represents an exception originating on the server. It contains an ID counter and a timestamp field. This ID counter starts at one (**1**) and increases until the server is restarted or the maximum integer value is reached and then it resets to one (**1**) again. The ID is used to uniquely identify an exception within a server log. This class can be instantiated around another exception, such as a SQLException.

DowntimeException

A DowntimeException occurs on the client when communication to the server fails or a severe problem on the server takes place.

BusinessException

This type of execution is thrown from either the client or server whenever the code encounters an attempt to perform some action that is defined as invalid by the functional business requirements. Rules and business objects primarily throw BusinessExceptions. BusinessExceptions are not considered severe errors and allow the system to continue to operate after the exception takes place.

UIException

A UIException is used strictly on the client and indicates a failure in the GUI framework or a general failure in UI processing.

Exception Handling

This section describes exception handling.

The DAO Layer

All DAO interfaces should throw SimServerException. These exceptions should not be logged in the DAO layer. This is handled by the EJB before propagating the exception to the client. The database framework generates most SimServerExceptions, though sometimes logic requires manually throwing a SimServerException. The Example: *ProductGroupDao* is an example of a DAO interface declaration and a SimServerException thrown manually:

Example: *ProductGroupDao*

```
void insert(ProductGroup productGroup) throws SimServerException;
```

Example: *ProductGroupOracleDao*

```
public void insert(ProductGroup productGroup) throws SimServerException {
  if (productGroup.getId() != null) {
     throw new SimServerException("Unable to insert product group with an existing identifier!");
  }
  productGroup.doSetId(getNextGroupId());
  execute(new ParametricStatement(RkProductGroupDataBean.INSERT_SQL,
            fromObjectToBean(productGroup).toList(true)));
  if (productGroup.isAllItems()) {
      return;
  }
  insertProductGroupDetails(productGroup);
}
```

The Service Layer

The service layer framework is written so that Exception, SimServerException, BusinessException and DowntimeException are all thrown by the service and the EJB, such that the exception is not wrapped when it arrives at the client. Services should all be declared to throw a simple Exception. No logging needs to be handled manually in the code. The framework code handles logging the exceptions before throwing them to the client. The following is an example of a service declaration and a BusinessException thrown manually from within the service layer. DowntimeExceptions are generally only thrown when an unexpected Throwable is caught from the code (such as a NullPointerException) or the EJB stub is no longer communicating.

Example: *ReplenishmentServerServices*

```
public PickList updatePickList(PickList pickList) throws Exception {
    PickListUpdateCommand command = CommandFactory.createPickListUpdateCommand();
    command.setPickList(pickList);
    command.execute();
    return command.getPickList();
}
```

Example: *PickListUpdateCommand*

```
protected void doExecute() throws Exception {
  PickList dbPickList = DAOFactory.getPickListDao().selectPickList(pickList.getId());
  if (dbPickList == null) {
      throw new BusinessException(ErrorKey.PICK_LIST_LIST_MUST_EXIST);
  }
  if (dbPickList.getStatus() == PickListStatus.CANCELED) {
      throw new BusinessException(ErrorKey.PICK_LIST_CANCELLED_ERROR);
  }
  if (dbPickList.getStatus() == PickListStatus.CLOSED) {
      throw new BusinessException(ErrorKey.PICK_LIST_CLOSED_ERROR);
  }
  // Rest Of Method
}
```

The UI Layer

On the client side, it is preferable that a BusinessException be thrown whenever business logic reaches an error state. All exceptions are caught and handled by the framework.

Throwing an Exception

When throwing an exception within the PC application, throw a BusinessException with the appropriate message. The exception should always be propagated to the last place that handled an event in order to cleanly break the execution flow.

Example: *ItemTicketListModel*

```
public void updateStockOnHand(ItemTicket itemTicket) throws Exception {
  Quantity stockOnHand = itemTicket.getItem().getAvailableStockOnHand();
  if (stockOnHand.intValue() < 1) {
      throw new BusinessException(SimClientErrorKey.ITEM_NOT_UPDATED);
  }
  // Remainder of Method
}
```

Catching an Exception

Exceptions should always be propagated and caught at the screen layer if triggered by a menu button, or in the panel layer if triggered by an editor event. The helper method displayException() should always be used to handle the error correctly, whether in a screen or panel. The following example allows each of the panel handle() methods to simply throw an exception and allows the screen to handle it.

Example: *ItemTicketListScreen*

```
public void navigationEvent(NavigationEvent event) {
    String command = event.getCommand();
    try {
        if (command.equals(SimNavigation.PRINT_TICKETS)) {
            panel.handlePrintTickets();
        } else if (command.equals(SimNavigation.CREATE)) {
            panel.handleCreate();
        } else if (command.equals(SimNavigation.UPDATE_SOH)) {
            panel.handleUpdateStockOnHand();
        } else if (command.equals(SimNavigation.DELETE)) {
            panel.handleDelete();
        }
    } catch (Throwable exception) {
        displayException(panel, event, exception);
    }
}
```

An alternate case to handling exceptions at the screen layer occurs when the code must continue to execute normally after the exception is displayed, usually to clean up or reset some information. In Example: *ItemTicketListPanel*, the table must be refreshed whether or not an error occurred:

Example: *ItemTicketListPanel*

```
public void handleUpdateStockOnHand() throws Exception {
    if (itemTicketTable.getSelectedRowCount() == 0) {
        displayError(SimClientErrorKey.NO_ROWS_SELECTED);
        return;
    }
    // Code removed from example
    List<ItemTicketWrapper> wrappers = itemTicketTable.getAllSelectedRowData();
    for (ItemTicketWrapper wrapper : wrappers) {
        try {
            model.updateStockOnHand(wrapper.getItemTicket());
        } catch (BusinessException exception) {
            displayException(exception);
        }
    }
    itemTicketTable.refreshTable();
}
```

Processing an Exception

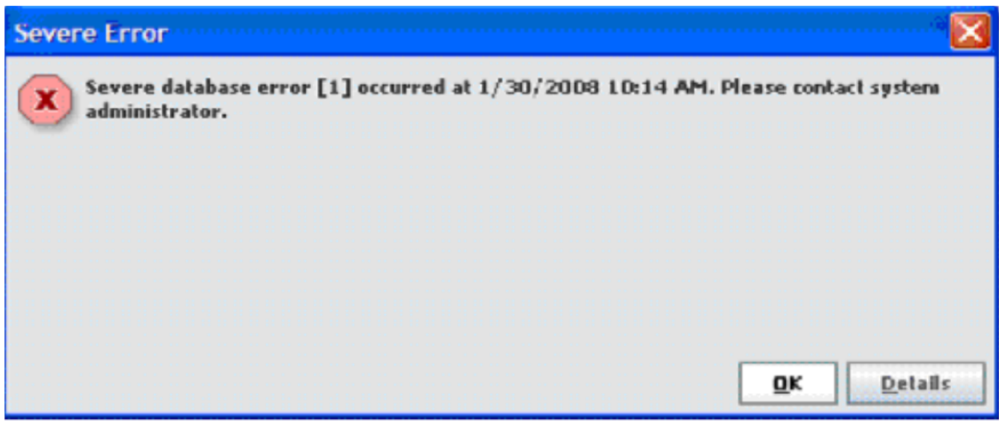
BusinessExceptions and UIExceptions are caught and their message displayed in a pop-up window that locks the application as shown in Figure: *Table Editor Error Screen*. When the window is exited, control should be returned to the screen/area in which the error occurred. If the UIException has an ErrorSeverity of FATAL, then a Fatal Window is displayed and the user is returned to the main login screen of the application.

Figure: *Table Editor Error Screen*

SimServerExceptions are thrown from the server layer and regardless of the details of the exception, only one message displays as shown in the figure below.

It includes an error number and the date and time that the error occurred. This can be used to find the error in the server exception log. Click **Details** to display the contents of the main exception.

Figure: *Severe Error Screen*



Logging

Logging allows the developer to write information about errors or processes to a file. Errors on the server side are automatically logged by the EJB class regardless of whether or not they come from the DAO layer or business layer. Therefore, errors that are logged in local code end up being logged twice if the user logs them.

LogService

If it is necessary to log an error in the business layer or DAO layer, this is done through the LogService class, which provides numerous static methods that hide the implementation away from the end developer. The SimStore business object in the example below logs a message if it fails to load the buddy stores. The LogService should be configured at the time of installation, but can also be configured during runtime to change logging modes.

Example: *SimStore*

```
private void doLoadStoreLists() {
    if (storesNotLoaded) {
        try {
            StoreServices storeServices = ClientServiceFactory.getStoreServices();
```

```
            buddyStores
                = CollectionUtil.newSortedSet(storeServices.findBuddyStores(store.getId()));
            autoReceiveStores
                = CollectionUtil.newSortedSet(storeServices.findAutoReceiveStores(store.getId()));
            storesNotLoaded = false;
        } catch (Exception exception) {
            LogService.error(this,"SimStore could not load store lists", exception);
        }
    }
}
```

Debugging

LogService provides the ability to log messages at the info, debug and warn level as well. All debugging messages should be logged through LogService as well. In the example below, the e-mail dispatcher logs a warning if no permission is associated with the e-mail.

Example: *EmailDispatcher*

```
public static void sendEmailAlert(EmailAlert emailAlert) {
  String fromAddress = RkConfigManager.getString(RkConfigManager.EMAIL_FROM_NAME);
  if (StringHelper.isNullOrEmpty(fromAddress)) {
      LogService.error("EmailDispatcher", "Invalid email from-address");
      return;
  }
  String permissionName = emailAlert.getPermissionName();
  if (StringHelper.isNullOrEmpty(permissionName)) {
      LogService.warn("EmailDispatcher", "Email alert is not associated to a permission.");
      return;
  }
  // Remainder Of The Code
}
```

The business rule executed when the logic attempts to create a new line item for a return will log a debug level message if the item is not supplied by the correct source.

Example: *ReturnCreateLineItemRule*

```
public static void execute(Return stockReturn, StockItem stockItem) throws BusinessException {
  Source source = stockReturn.getDestination();
  if (source == null) {
      throw new BusinessException(ErrorKey.RETURN_NO_DESTINATION);
  }
  if (stockReturn.isVendorReturn()) {
      try {
        ItemServices services = ClientServiceFactory.getItemServices();
        if (services.isItemSuppliedBySupplier(stockItem.getId(), source.getId())) {
          return;
      } catch (Throwable exception) {
          LogService.debug(ReturnCreateLineItemRule.class, "Failed rule.", exception);
      }
      throw new BusinessException(ErrorKey.RETURN_NO_SOURCE);
  }
  // Remainder of rule code
}
```

Logs

One of the first places to look for information concerning a problem in SIM is in the log files. Stack traces and debugging information can be found within the log files. The log files are configured to roll over once they reach a certain size (currently 10 MB). Once a log file reaches the configured size, it is renamed (for example, sim.log will be renamed to **sim.log.1**) and new log messages are written to a new file (for example, sim.log). If there are already rolled-over logs, they are also be renamed (for example, sim.log.1 becomes sim.log.2, sim.log.2 becomes sim.log.3, and so forth). Only ten files are kept. If ten files already exist and the current file rolls over, the oldest log file is deleted.

Client Side Logs

On the client, logs are sent to the console and to the file <location of sim client>\bin\log\sim.log.

Logging is configured in client/resources/log4j.xml. This file defines which kinds of messages are logged and where they are logged.

Server Side Logs

On the server, logs are saved to <location of server>\bin\log\sim.log. Log messages may also displayed in the server console.

Exception Format

The following example demonstrates a formatted and logged service exception. The first line contains the EJB and method name of the exception failure. The second line contains ERROR with an ID (in this case **1**). This ID is displayed on the client side for reference. It is followed by the user ID of the transaction user, a timestamp, the type of exception, the primary message, and the primary message of the root cause. Following that is the stack trace of the exception.

```
ERROR 02:14:05.728 [ejb.ItemEjb] findItemVOs: Exception occurred during service invocation <
ERROR-1 User: 15000 Time: 11/27/06 2:14 PM Type: ApplicationException Message: This is an example
exception. Root Cause: Application is in illegal state.>
oracle.retail.sim.closed.common.ApplicationException: This is an example exception. at
oracle.retail.sim.closed.item.ItemServerServices.findItemVOs(ItemServerServices.ja va:37) at
oracle.retail.sim.closed.item.ItemHelper.findItemVOs(ItemHelper.java:55) at
oracle.retail.sim.closed.item.ejb.ItemEjb.findItemVOs(ItemEjb.java:127) at
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39) at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java: 25) at
java.lang.reflect.Method.invoke(Method.java:585) at
com.evermind.server.ejb.interceptor.joinpoint.EJBJoinPointImpl.invoke(EJBJoinPoint Impl.java:35)
at com.evermind.server.ejb.interceptor.InvocationContextImpl.proceed(InvocationContex
tImpl.java:69) at
com.evermind.server.ejb.interceptor.system.DMSInterceptor.invoke(DMSInterceptor.ja va:52) at
com.evermind.server.ejb.interceptor.InvocationContextImpl.proceed(InvocationContex tImpl.java:69)
at com.evermind.server.ejb.interceptor.system.TxSupportsInterceptor.invoke(TxSupports
Interceptor.java:37) at
com.evermind.server.ejb.interceptor.InvocationContextImpl.proceed(InvocationContex tImpl.java:69)
at com.evermind.server.ejb.interceptor.system.DMSInterceptor.invoke(DMSInterceptor.ja va:52) at
com.evermind.server.ejb.interceptor.InvocationContextImpl.proceed(InvocationContex tImpl.java:69)
at com.evermind.server.ejb.StatelessSessionEJBObject.OC4J_invokeMethod(StatelessSessi
onEJBObject.java:86) at ... 24 more
```

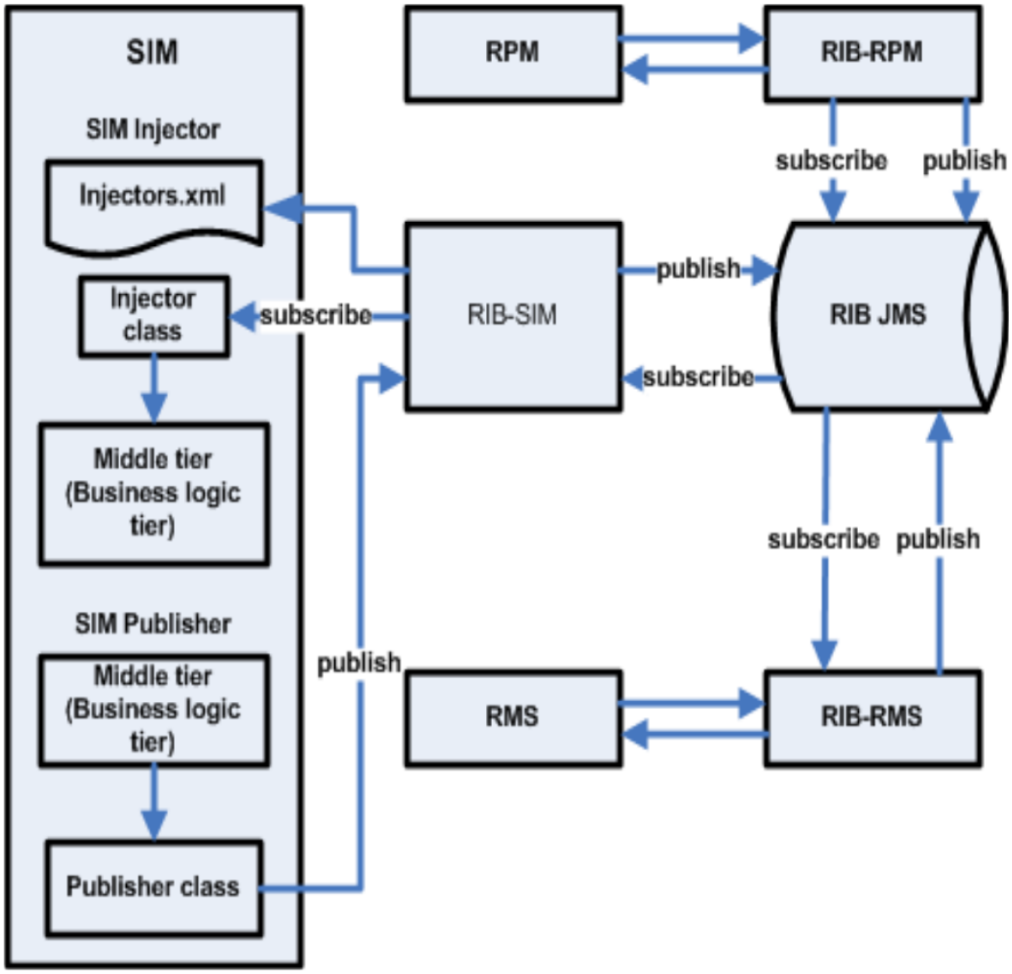
External System Integration

RIB-based Integration

SIM can integrate with other Retail products (such as a merchandising system, a price management system, and a warehouse management system) through Oracle Retail Integration Bus (RIB). RIB utilizes publish and subscribe (pub/sub) messaging paradigm with some guarantee of delivery for a message. In a pub/sub messaging system, an adapter publishes a message to the integration bus that is then forwarded to one or more subscribers. The publishing adapter does not know, nor care, how many subscribers are waiting for the message, what types of adapters the subscribers are, what the subscribers current states are (running/down), or where the subscribers are located. Delivering the message to all subscribing adapters is the responsibility of the integration bus.

See the *Oracle Retail Integration Bus Operations Guide* and other RIB-related documentation for additional information.

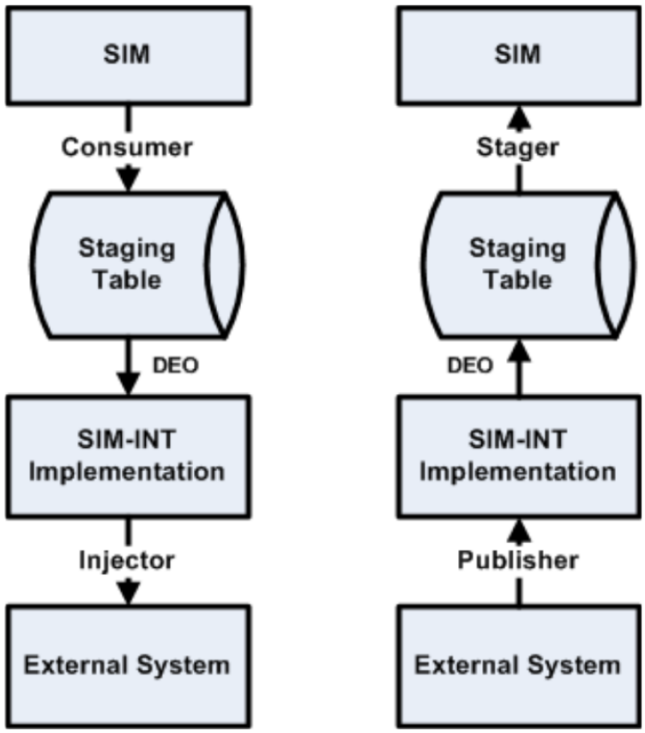
Figure: ***RIB-based Integration***



SIM Standalone Integration

SIM is also capable of integration with external systems without the use of the RIB. This allows SIM to allow integration to legacy and other non-enterprise systems. All messages that come into or out of SIM through an Injector or Publisher are first staged into a staging table. This staging table isolates SIM from the workings of the external system. In order to connect to an external system, simply customize the injector or publishers classes in order to connect to the external system instead of using the RIB versions of these classes. Details about accomplishing this are described below.

Figure: ***SIM Standalone Integration***



Modifying Data Transport to External Systems

To modify data transport to external systems, we can continue the previous example of Generic Retailers Inc, adding the new attribute Season to the items within the system. Besides the SIM side as given in earlier examples, this information could also arrive from an external system and might need to be sent to external systems. So in this case, the following steps are taken to support the new data:

1. Update DEOs
2. Update Injectors
3. Update Consumers
4. Update Stagers
5. Update Publishers

This customization document does not cover any modifications to code or processes external to SIM, or in other words, how to update the external system. As such, this document does not cover updating RIB configurations and payloads should the external system be linked using the RIB.

Process of Receiving an External Message

Do the following to receive an external message:

1. The external message is received by an Injector.
2. The Injector class converts the message to a Data Exchange Object.
3. The Data Exchange Object (DEO) is written into the MPS_STAGED_MESSAGE table.
4. The DEO is read from the MPS_STAGED_MESSAGE table by a polling timer and given to a Consumer object to be processed.
5. The Consumer object uses the DEO to do processing and update SIM information.

Process of Sending an External Message

Do the following to send an external message:

1. Business processes use a Stager class to convert business data into a DEO.
2. The DEO is written into the MPS_STAGED_MESSAGE table.
3. The DEO is read from the MPS_STAGED_MESSAGE table by a polling timer and given to a Publisher object to be processed.
4. The Publisher object converts the DEO to external information and sends the information to an external system.

Update DEOs

The first step of customizing the external message process is to update the Data Exchange Objects (DEOs) to contain the added attributes. ItemHdrDEO (incoming message containing the basic item information) and ReturnToVendorLineItemDEO (outgoing message containing item information on a return) will be used as examples of this process. Data Exchange Objects map very closely to Retail Enterprise payloads. Refer to Enterprise integration documentation for definitions of payloads and payload attributes.

Since this is new information on already existing messages, the original DEOs should be sub-classed and the new attributes added much the same way as was done for business objects and value objects:

```
public class GriItemHdrDEO extends ItemHdrDEO {
    private String season;
    public GriItemHdrDEO() {
        super();
    }
    public String getSeason() {
        return season;
    }
    public void setSeason(String season) {
         this.season = season;
    }
}
```

DEOs are instantiated through a DEOFactory. This factory uses a configured implementation to create the objects. The next step is to subclass the default implementation (DEOFactoryImpl) to override the methods that create the modified DEOs:

```
public class GriDEOFactoryImpl extends DEOFactoryImpl {
  public ItemHdrDEO createItemHdrDEO() {
      return new GriItemHdrDEO();
  }
  public ReturnToVendorLineItemDEO createReturnToVendorLineItemDEO() {
      return new GriReturnToVendorLineItemDEO();
```

```
  }
}
```

To configure the factory, simply modify server.cfg with the classpath of your new code:

```
DEO_FACTORY_IMPL=gri.custom.code.GriDEOFactoryImpl
```

Update Injectors

SIM's injector is generic and does not need to be altered or extended. It simply takes a payload, looks up an appropriate mapper that converts the payload to a DEO, and then stages the DEO. The key to extending the functionality is to alter a DEO (previously cover) and the use a custom mapper.

To add a customer mapper, the standard process of customization and configuration is used. Message mappers are returned from a message mapper factory, so begin with extending this factory:

```
public class GriMessageMapperFactoryImpl extends SimMessageMapperFactoryImpl
```

To configure the factory, simply modify server.cfg with the classpath of your new code:

```
MESSAGE_MAPPER_FACTORY_IMPL=gri.custom.code.GriMessageMapperFactoryImpl
```

Then override the method that retrieves mappers:

```
public <T, U extends DataExchangeObject> DEOMapper<T, U> getMapper(Class<?> sourceClass) {
Class<? extends DEOMapper<T, U>> mapperClass
          = findMyCustomClass();
if (mapperClass != null) {
    return mapperClass;
}
return super.getMapper(sourceClass);
                }
```

Finally, you will have to extend the actual mapper to do your additional work. For example, to extend the functionality of the AsnInDescMapper:

```
public class GrAsnInDescMapper extends ASNInDescMapper {
    public ASNInDEO mapToDEO(ASnInDesc asnInDesc) throws Exception {
            ASNINDEO deo = super.mapToDEO(asnInDesd);
            // Perform customized work here
            return deo;
      }
}
```

Update Consumers

The first step of customizing a consumer is to subclass the consumer and add custom logic. In order to deal with the new season attribute, ItemCreateConsumer is customized as an example. Every consumer must implement consume(), and so this is the method that must be overridden with custom code. Calling super.consume() will guarantee that the normal process of consuming external data will be executed and that future releases will continue to work with no code changes.

```
public class GriItemCreateConsumer extends ItemCreateConsumer {
  public void consume(List<DataExchangeObject> deos) throws Exception {
      super.consume(deos);

      GriItemDao griItemDao = new GriItemDao();
      for (DataExchangeObject deo : deos) {
          ItemHdrDEO itemHdrDEO = ((ItemDEO) deo).getItemHdrDEO();
          griItemDao.updateSeason(itemHdrDEO.getItemId(), itemHdrDEO.getSeason());
```

```
      }
 }
```

The second step is very similar to other customization patterns: the factory implementation is altered to return the custom consumer and then server.cfg is updated to point at the new implementation. The custom consumer factory implementation should extend the regular factory implementation and then override the getConsumer() method. After first returning any customized consumers, it can then call the superclass to retrieve a SIM supplied consumer:

```
public class GriMessageComsumerFactoryImpl extends SimMessageComsumerFactoryImpl {
   public SimMessageConsumer getConsumer(SimMessageType messageType) throws Exception {
       if (messageType == SimMessageType.ITEM_CRE) {
           return new GriItemCreateConsumer();
       }
       return super.getConsumer(messageType);
    }
}
```

To configure the factory, modify server.cfg with the classpath of your new code:

```
MESSAGE_CONSUMER_FACTORY_IMPL=gri.customer.code.GriMessageConsumerFactoryImpl
```

Update Stagers

Stagers and Publishers are the opposite of Consumers and Injectors. They take data from SIM and prepare it to be sent externally.

The first step of customizing a stager is to subclass the stager and add custom logic. In order to deal with the new season attribute, VendorReturnStager is customized as an example. Every stager must implement getDataExchangeObjects(), and so this is the method that must be overridden with custom code. Calling super.getDataExchangeObjects() first will guarantee that the normal process of staging external data will be executed and that future releases will continue to work with no code changes.

```
public class GriVendorReturnStager extends VendorReturnStager {
  public List<DataExchangeObject> getDataExchangeObjects() {
List<DataExchangeObject> deos = super.getDataExchangeObjects();
// Do Custom work
return deos;
  }
}
```

The second step is very similar to other customization patterns: the factory implementation is altered to return the custom stager and then server.cfg is updated to point at the new implementation. The custom stager factory implementation should extend the regular factory implementation and then override the getStager() method. After first returning any customized stagers, it can then call the superclass to retrieve a SIM supplied consumer:

```
public class GriMessageStagerFactoryImpl extends SimMessageStagerFactoryImpl {
    public VendorReturnStager getVendorReturnStager(Return stockReturn) {
      return new GriVendorReturnStager(stockReturn);
   }
}
```

To configure the factory, modify server.cfg with the classpath of your new code:

```
MESSAGE_STAGER_FACTORY_IMPL=gri.custom.code.GriMessageStagerFactoryImpl
```

Update Publishers

SIM's publisher is generic and does not need to be altered or extended. It simply takes a DEO, looks up an appropriate mapper that converts the DEO to a payload, and then publishers the payload to the RIB. For alternate methods of publishing, see Stand Alone Deployment. The key to extending the functionality is to alter a DEO (previously cover) and the use a custom mapper.

To add a customer mapper, the standard process of customization and configuration is used. Message mappers are returned from a message mapper factory, so begin with extending this factory:

```
public class GriMessageMapperFactoryImpl extends SimMessageMapperFactoryImpl
```

To configure the factory, simply modify server.cfg with the classpath of your new code:

```
MESSAGE_MAPPER_FACTORY_IMPL=gri.custom.code.GriMessageMapperFactoryImpl
```

Then override the method that retrieves mappers:

```
public <T, U extends DataExchangeObject> DEOMapper<T, U> getMapper(Class<?> sourceClass) {
Class<? extends DEOMapper<T, U>> mapperClass
          = findMyCustomClass();
if (mapperClass != null) {
    return mapperClass;
}
return super.getMapper(sourceClass);
               }
```

Finally, you will have to extend the actual mapper to do your additional work. For example, to extend the functionality of the AsnInDescMapper:

```
public class GrAsnInDescMapper extends ASNInDescMapper {
     public ASNInDesc mapFromDeo(ASnInDEO asnInDEO) throws Exception {
              ASNInDesc payload = super.mapFromDeo(asnInDEO);
              // Perform customized work here
              return payload;
       }
}
```

Customizing Look and Feel

Customization of the look and feel of the SIM PC application is available directly through the PC GUI and the appropriate privileges. Fonts, colors and icons can be changed to suit the company's desires. Completely new and fresh themes can be created directly through SIM. Refer to the *Oracle Retail Store Inventory Management User Guide* for description of how to use these features.

Visual appearances outside of this spectrum, such as widget shapes or animation, are not open to customization. Note that if completely new workflows are built on the PC, there are no restrictions as to what UI elements are put on the new screens.

The handheld code is not open to look and feel changes. These devices use simple two-tone text-based screens. All customizations are subject to the frameworks provided by Wavelink. See the Wavelink documentation for additional information.

Customizing a Theme Icon

To customize icons on the client:

1. Create a new custom directory for custom images in the client environment default images directory. The client images directory is sim_home\client\resources\images where sim_home is the SIM environment home directory.

   Example: C:\apps\orsim\client\resources\images\customimages
2. Alter icon using PC Technical Maintenance.
   - Log on to PC client.
   - Navigate to Admin > Technical Maintenance > UIConfiguration > Customize Themes.
   - Select theme to customize and navigate to Icons.
   - Select icon to edit and press edit.
   - Edit the icon path to the new icon and click **Apply**.
   - Save the theme.
3. Restart the PC Client.

Customizing Barcodes

Barcodes are handled in SIM by barcode processors. The first step of customization is to understand the BARCODE_PROCESSOR table in the database. This table is read to determine which barcode processors should be used when searching for items based on an input string.

Table: *BARCODE_PROCESSOR*

Column	Comment
NAME	Name of the processor
DESCRIPTION	Description of the processor
FILE_NAME	Java classpath name of the processor
ACTIVE	Barcode active indicator, Y means barcode is active, N means it is not active.
PROCESS_SEQUENCE	Indicates the sequence of processors used to attempt to process barcode.

The NAME and DESCRIPTION rows are not used and are simply there to label the record in the database.

The FILE_NAME row represents the fully qualified classpath to the .java file that implements that particular barcode. For example, the UPC-E barcode processor FILE_NAME value supplied by SIM is oracle.retail.sim.closed.item.barcode.BarcodeProcessorUPCE.

The ACTIVE row must have a **Y** or **N** value to determine whether or not it is used in the system. The first and easiest customization is to activate or deactivate barcode processors based on usage. SIM provides several barcode processors which are all marked as **Inactive** by default. Updating the ACTIVE column to **Y** for those that are used. Note that the more active processors there are, the longer it takes to find items.

To add new barcode processors to the system, first code a new barcode processor Java class and then insert a row into the BARCODE_PROCESSOR table like in Table: *BARCODE_PROCESSOR* This example is the processor for a 24-character barcode. The Java class must be placed in the classpath of the deployed server.

Note

Barcode processor classes are cached, so, altering the table, for example, and making a barcode active, will not have an effect on the system until the server has been restarted.

Note

Barcode processing stops at the first barcode processor that successfully parses the barcode; so if processor #1 successfully parses the barcode, processor #2 and so forth will not be attempted. So, the PROCESS_SEQUENCE should be set to the most commonly used barcodes to least commonly used barcodes. Any barcodes not being used should be deactivated.

Creating a Barcode Processor

Barcode processors are implemented as Java classes, but are dynamically loaded from a database table. Once a new barcode process is created, a record will have to be placed in the BARCODE_PROCESSOR table and the server will need to be restarted. The new barcode processor must implement the BarcodeProcessor interface.

For example:

```
public class BarcodeProcessorABC implements BarcodeProcessor
```

There are two methods in the BarcodeProcessor interface that will need to be implemented.

1. BarcodeRecord parseBarcodeRecord(String barcode):

   The input parameter to this method is the barcode to be parsed. The method should parse out the barcode and return a BarcodeRecord object with all the appropriate values filled in from the barcode. If the barcode does not match the format of the processor or another error should occur, this method should return NULL which indicates that no BarcodeRecord exists.

2. void processBarcodeItem(BarcodeItem barcodeItem, BarcodeRecord barcodeRecord) throws Exception:

   After the parseBarcodeRecord() method is finished, the BarcodeRecord object will be used to lookup items. If items are found, this second method will be called passing the item found and the original BarcodeRecord for additional processing. This method should use information in the BarcodeRecord to lookup and assign appropriate values to the BarcodeItem. This method is responsible for setting price, UIN, and quantity on the BarcodeItem through whatever business rules or algorithms desired. The BarcodeItem object will already contain the associated StockItem.

   Note

   Both BarcodeRecord and BarcodeItem can be extended and customized by using the standard process for customizing business objects.

BarcodeRecord

This object represents the information parsed from a barcode by the processor. It carries some information about the type of barcode as well in case the next step needs the information.

Table: *Barcode Record*

Attribute	Description
itemId	The SKU or UPC number found in the barcode. This is required when parsing the barcode.
serialNumber	A serial number. This is optional and may or may not be found within the barcode.
type2Code	The single letter code indication what format of type 2 the record is. This value is used to find the correct item in the data store, which must contain this format code as part of the item's data. Current values are A->L. This attribute is optional.
type2Prefix	The prefix of the type 2 format. If this is a PLU Type 2 Barcode processor, then this value should be set to 2.
type2Format	True if this is a type 2 barcode, false if another type of barcode.
priceSupported	True if price is supported within the barcode format, false otherwise.
price	The price found in the barcode. This should be the exact value parsed from the barcode without any formatting.
gs1CodeValueMap	A map containing the GS1 information. The key is the GS1 AI code and the value is the information parsed out for that code.

BarcodeItem

This object represents the item found and some basic additional information from the barcode. This value is return by services to the client that supplied the scan.

Table: *Barcode Record*

Attribute	Description
stockItem	Populated by SIM based on the item id returned in the BarcodeRecord.
price	The price found on the barcode. This attribute is a SimMoney object. The processBarcodeItem() method is responsible for converting it from the String price in BarcodeItem. The default value is null.
quantity	The default value is one (1).
uin	The default value is null.

Replacing a Barcode Processor

To replace a barcode processor, simply deactivate the one supplied by base SIM and create a new custom barcode processor for the barcode type and then activate it.

Customizing Reports

Reporting Services

The ReportingServices interface defined three main services for printing reports: printReport() printers a single report, printReports() prints multiple reports at once to a single store printer, and the second printReports() prints multiple reports at multiler printers. All three of these services eventually make calls to an external printing service and all three of these take a ReportRequest as input. There is no need to customize reporting at the service level, but if the customer should choose to do so, it is done in the same manner as customizing any other service.

Report Request

The ReportRequest class itself consists of a key/value string map. Each report has its own sub-class of ReportRequest. These classes will contain getters() and setters() for each of the attributes of the report. These report requests can be customized to have additional attributes in the same standard fashion as any business object previous described in this document. The following is a list of available report requests:

DirectDeliveryReportRequest

FulfillmentOrderBinReportRequest

FulfillmentOrderDeliveryReportREquest

FulfillmentOrderPickReportRequest

FulfillmentOrderReportRequest

FulFillmentOrderReversePickReportRequest

InventoryAdjustmentReportRequest

ItemBasketReportRequest

ItemReportRequest

ItemRequestReportRequest

ItemTicketReportRequest

ReturnReportReuqest

ShelfAdjustmentReportRequest

ShelfReplenishmentReportRequest

StockCountRejectedItemReportRequest

StockCountReportRequest

StoreOrderReportRequest

TransferReportRequest

UinReportRequest

WarehouseDeliveryReportRequest

External Reporting Services

Eventually, all reports go through an external reporting service to connect to the external system that provides this printing. This connection to an external service can be customized following the standard services pattern. This allows customers to connect reporting services to any external printing system there is without the modification of any of SIM's base code. The customer begins by writing a new external service.

```
public class MyReportExternalServiceImpl implements
ReportingScheduleExternalServices {
     public ReportResponse scheduleReport(ReportRequest request,
            Locale locale, StorePrinter printer) {
        // Custom Code goes here
}
}
```

If the customer chooses to customize external printing, then the customized code is responsible for the connection to the external system, the protocols, formatting the data to whatever type is needed by the external system, security, etc, etc.

Acter creating the new external service implementation, the factory must also be updated.

```
public class MyExternalServiceFactoryImpl implements
ExternalServiceFactoryInterface {
```

```
    public  ReportingScheduleExternalServices
getReportingScheduleExternalServices() {
        return getService(MyReportExternalServiceImpl.class);
}
```

Then alter server.cfg to point at the correct factory:

```
EXTERNAL_SERVICE_FACTORY=custom.code.MyExternalServiceFactoryImpl
```

Customizing Item Ticket

Item Ticket Services

The ItemTicketServices interface defines a service for printing tickets: printTickets(). This service takes a store and a list of item tickets and prints them. There is no need to customize item tickets at the item ticket service level, but if the customer should choose to do so, it is done in the same manner as customizing any other service.

External Ticketing Services

Customizing the external item ticket printing service to connect to a desired external printing system is very similar to reports. First, you create a class that implements TicketPrintExternalServices. After creating the new external service implementation, the factory must also be updated.

```
public class MyExternalServiceFactoryImpl implements
ExternalServiceFactoryInterface {
    public TicketPrintingExternalServices getTicketPrintingExternalServices
(boolean printPreview) {
        return getService(MyTicketPrintingService.class);
}
```

The customization outlined above is not necessary if the customer has configured SIM to use external web services for printing item tickets. If so configured, the customer writes a web service provider for that defined API and does whatever they like with the data that is passed to them, printing in any customized fashion to any type of printing system. The specifics of the following web services can be found document with all the other web services.

Web Service	Operation	Input	Output
TicketPrinting	printTickets	ItmTktPrtDesc	
TicketPrinting	previewTickets	ItmTktPrtDesc	ImTktPrvDesc

Customizing Notifications

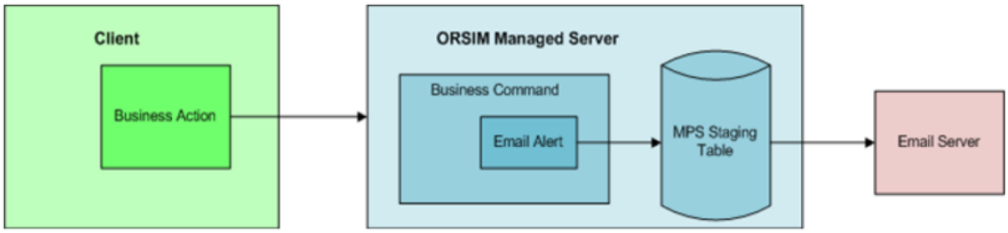
SIM E-mail Notification refers to e-mail messages sent as a direct result of business actions. These notifications enable users to easily remain up-to-date on important business functions and informed if actions need to be taken. This reduces the amount of time in transaction may be completed by reducing waiting periods.

SIM provides a notifications architecture which enables users to send e-mail notifications based on certain business events, as well as the options of implementing additional notifications and expanding messages onto other platforms.

E-mail Notification Workflow

The following diagram depicts the flow of an e-mail notification as it is processed in SIM until it is sent to an external e-mail system.

Figure: *E-mail Notification*



E-mail Server Configuration

The e-mail server is determined through the property MAIL_JNDI_NAME in server.cfg configuration file. The value for this property determines the session name on the server that should be used. The following variable is the default property:

```
MAIL_JNDI_NAME=mail/SimMailSession
```

This jndi name is used to retrieve a Session from the server from which we can get a Transport object and send a MimeMessage object, the fundamental classes of sending e-mails. See "E-mail Server Class Definitions" for further explanation.

The actual mail server associated with mail/SimMailSession is assigned to the server during the build and deploy. The install configuration templates include an input value for this input.mail.host that is populated when the template is loaded by looking in the specific properties files, so the ant.install.properties should contain a value for input.mail.host such as the following:

```
## Properties from Page:MailSessionDetails
input.mail.host = sysserver01
```

E-mail Server Class Definitions

The following is the standard javax package classes that are used as part of the e-mail alert system. For more information, access the documentation for the current version of Java used in the system.

javax.mail.Session

The Session class represents a mail session and is not sub-classed. It collects together properties and defaults used by the mail APIs. A single default session can be shared by multiple applications on the desktop. Unshared sessions can also be created. The Session class provides access to the protocol providers that implement the Store, Transport, and related classes. The protocol providers are configured using the following files:

- javamail.providers and javamail.default.providers
- javamail.address.map and javamail.default.address.map

javax.mail.Transport

An abstract class that models a message transport. Subclasses provide actual implementations. Note that Transport extends the Service class, which provides many common methods for naming transports, connecting to transports, and listening to connection events.

javax.mail.Message

This class models an e-mail message. This is an abstract class. Subclasses provide actual implementations. Message implements the Part interface. Message contains a set of attributes and content. Messages within a folder also have a set of flags that describe its state within the folder.

javax.mail.internet.MimeMesage

This class represents a MIME-style e-mail message. It implements the Message abstract class and the MimePart interface. If you need to create new MIME-style messages instantiate an empty MimeMessage object and then fill it with appropriate attributes and content. This is the type of mail message that SIM uses.

SIM E-mail System Configuration Values

E-mail has several configuration values that can by assigned through the PC client administration screens. See System Administration Options Table in the *Oracle Retail Store Inventory Management Implementation Guide, Volume 1, - Configuration*.

SIM E-mail Batch Jobs

There are batch jobs that are run that produce e-mail alerts in the system. See the Batches section of the documentation for the definition and functionality of these batch jobs.

E-mail Alert Content and Processing

Content for the e-mails is the responsibility of e-mail alerts. All e-mail alert commands implement the e-mail alert interface. Once the e-mail is constructed, it is staged through DEO to the MPS_STAGED_MESSAGE table. This DEO is picked up asynchronously by the EmailNotificationPublisher and published to an external system (or e-mailed). For customizing the EmailNotificationPublisher, see the "RIB-based Integration" in the *Oracle Retail Store Inventory Management Implementation Guide, Volume 2 - Integration with Oracle Retail Applications* on customizing publishers.

EmailAlert.java

This interface is implemented by all e-mail alert commands. It contains the method APIs necessary for the e-mail framework classes to perform generic processes on e-mail alerts.

FulfillmentOrderCreateEmailAlertCommand

This command builds the e-mail that notifies users when a fulfillment order is created.

FulfillmentOrderPickReminderEmailAlertCommand

This command builds the e-mail that notifies users that a pick list has been started but not completed yet.

FulfillmentOrderReceiptEmailAlertCommand

This command builds the e-mail that notifies users when a fulfillment order is received.

ReturnRequestEmailAlertCommand

This command builds the e-mail that notifies users when an active Return is nearing its not-after-date.

TransferDamageEmailAlertCommand

This command builds the e-mail that notifies users when items have been received as damaged.

TransferDispatchedEmailAlertCommand

This command builds the e-mail that notifies users when a transfer has been dispatched.

TransferDispatchedOverdueEmailAlertCommand

This command builds the e-mail that notifies users when a dispatched transfer is overdue.

TransferDispatchRequestEmailAlertCommand

This command builds the e-mail that notifies users when a transfer has been requested.

TransferOverUnderReceiptEmailAlertCommand

This command builds the e-mail that notifies users when a transfer is received with quantities that are different than the amount transferred.

TransferRequestEmailAlertCommand

This command builds the e-mail that notifies users that a transfer request needs to be accepted.

UINStoreAlteredEmailAlertCommand

This command builds the e-mail that notifies users that a UIN has had its store location moved without going through a normal inventory transaction.

UserPasswordAssignmentEmailAlertCommand

This command builds the e-mail that notifies a user when the system has assigned a new password to their account.

Creating or Customizing Notifications

Store Inventory Management's notifications architecture possesses a number of points at which customization can easily take place. Utilization of these different points is dependent on the desired result of these enhancements. ORSIM employs a number of server-side commands to execute business functionality which can be substituted by customized modules for business-specific behaviors. The notifications architecture also employs a series of commands implementing the EmailAlert interface; these commands construct the various e-mail messages to be sent. Any user desiring to overwrite or construct new e-mail messages must create classes that implement EmailAlert and stage the e-mail message.

Customizing E-Mail Alert

The first step of customizing an e-mail alert is to subclass SIM's e-mail alert and add custom logic. The TransferDamageEmailAlertCommand is customized as an example. The easiest way to accomplish this is to sub-class the original command. Every e-mail alert must implement the EmailAlert interface. The interface consists of the following for methods.

EmailAlert Interface

The following describes the EmailAlert interface:

- *Method: String getPermissionName();*

  This returns the name of permission required to receive this e-mail from the system.
- *Method: Long getDestinationStoreId();*

  This method returns store identifier of the store to receive the e-mail.
- *Method: String createSubject(Locale locale);*

  Returns subject line of the e-mail. Locale might be used to internationalize the e-mail subject line.
- *Method: String createBody (Locale locale);*

  Returns content of the e-mail message. Locale may be used to internationalize the e-mail content.

Customized EmailAlert Example

The following is an example of customized EmailAlert:

```
public class CustomizedTransferDamageEmailAlert extends
TransferDamageEmailAlertCommand {
     private Transfer transfer;
     public void setTransfer(Transfer transfer) {
          this.setTransfer(transfer);
          this.transfer = transfer;
     }

public String createSubject(Locale locale) {
        return TranslatorServerCache.getMessage(locale, "Custom Transfer Damaged
Subject Line");
    }

public String createBody(Locale locale) {
        StringWriter emailContent = new StringWriter();
        PrintWriter printWriter = new PrintWriter(emailContent);
   printWriter. println(TranslatorServerCache.getMessage(locale,
              "Damaged goods received for transfer: ", transfer.getId()));
        printWriter.println();
        return emailContent.getBuffer().toString();
    }
}
```

The second step to customizing an e-mail alert is very similar to other customization patterns: the factory implementation is altered to return the customized e-mail alert, and then server.cfg is updated to point at the new implementation. The custom factory implementation should extend the regular factory implementation and then override only the single create() method that it needs to.

Customized Command Factory Example

The following is an example of customized command factory.

```
public class CustomizedCommandFactoryImpl extends CommandFactoryImpl {
    public TransferDamageEmailAlertCommand
             createTransferDamageEmailAlertCommand() {
        return new CustomizedTransferDamageEmailAlert();
    }
}
```

To configure the factory, simply modify server.cfg and set the value to the customized class.

Server.cfg

The following is an example of server.cfg.

```
COMMAND_FACTORY_IMPL=customized.source.CustomizedCommandFactoryImpl
```

Expanding Notification Alerts

In addition to customization, it is possible to create additional alerts in ORSIM through more extensive implementation:

- Custom or create business commands to trigger the e-mail alert.
- Create Email Alert Class.
- Create Email DEO Class.
- Create Stager to write this new message to the MPS Staging Table.
- Create new Publishers to read this message from the MPS Staging Table and invoke intended behavior for the message.
- Create new MPS Worker Type so any new notification types can be independently controlled and tuned for optimal performance.
- Register the new MPS Message Family and type, as well the new Publisher for the message in the server initializer.

Implementation of Business Commands

Each service API is implemented by business commands (see previous sections on services). Business commands can be altered in post-processing fashion to send out notifications using the following command extensibility guidelines.

```
        public class MyBusinessCommand extends CreatePickListCommand {
protected void doExecute() throws Exception {
    super.doExecute();
    // Additional code to send notification that pick needs to take place
}
}
```

Implementation of SMS Alerts Example

Included in this document is example code meant to assist in the expansion of ORSIM notifications architecture to include SMS alerts, using the steps listed in the previous section.

Object Structure

The following is a sample code for an interface defining an SMS alert, and a command to create an SMS alert upon creation of a new Fulfillment Order in ORSIM.

```
public interface SMSAlert {
    /**
     * Returns the name of the permission required to receive this alert.
     */
    String getPermissionName();

    /**
     * Returns the destination store id of the alert.
     */
    Long getDestinationStoreId();

    /**
     * Returns the content of the alert, translated to the locale.
     */
    String createMessage(Locale locale);
```

```
}
```

Email Alert Command

```
public class FulfillmentOrderCreateSMSAlertCommand extends Command
   implements SMSAlert {

    private FulfillmentOrder fulfillmentOrder;

    private static final String MESSAGE = "New Customer Order for Store {0}";

    public void setFulfillmentOrder(FulfillmentOrder fulfillmentOrder) {
        this.fulfillmentOrder = fulfillmentOrder;
    }

    public String getPermissionName() {
        /* Return the permission key required to receive SMS message here */
    }

    public Long getDestinationStoreId() {
        return fulfillmentOrder.getStoreId();
    }

    public String createMessage(Locale locale) {
        /* Customize the message here using fulfillmentOrder's data */
        return TranslatorServerCache.getMessage(
                locale, MESSAGE, getStoreDescription(locale));
         }

    protected void doExecute() throws Exception {
        /* Stage the message from the Stager here */
    }

    private String getStoreDescription(Locale locale) {
        try {
            Store store = DAOFactory.getStoreDao().selectStore(
                              fulfillmentOrder.getStoreId());
            return store.toDisplayString();
        } catch (Throwable exception) {
            return TranslatorServerCache.getText(locale, "Invalid Store");
        }
    }
}
```

MPS Message Stager

The following is sample code for a stager to write this SMSAlert to the MPS Staging Table.

```
public class SMSNotificationStager extends SimMessageStager {

    private SMSAlert smsAlert;
    private String fromAddress;

    public SMSNotificationStager(SMSAlert smsAlert) {
        this.smsAlert = smsAlert;
    }

    protected SimMessageType getMessageType() {
        /* Return the SimMessageType here */
    }

    protected Long getStoreId() {
```

```
        return smsAlert.getDestinationStoreId();
    }

    public List<DataExchangeObject> getDataExchangeObjects() throws Exception {
        fromAddress = getFromAddress();

        List<DataExchangeObject> dataExchangeObjects = new
ArrayList<DataExchangeObject>();
        List<User> users = findNotificationUsers(smsAlert.getPermissionName(),
smsAlert.getDestinationStoreId());
        dataExchangeObjects.addAll(createSMSNotificationDEOs(users));
        return dataExchangeObjects;
    }

    private String getFromAddress() throws Exception {
        /* Obtain the address sending the SMS here */
    }

    private List<User> findNotificationUsers(String permissionName, Long storeId)
{
        /* Find all Users that will receive this SMS based on the permission here
*/
    }

    private List<DataExchangeObject> createSMSNotificationDEOs(List<User> users) {
        /* Set DEO data here */
    }
}
```

MPS Message Publisher

Below is sample code for a publisher to read this SMSAlert from the MPS Staging Table and invoke intended behavior. This will most likely be the most customized module of this example. This publisher utilizes an e-mail gateway of a mobile phone service provider which then converts it into a SMS alert and forwards it onto the user. Depending on specifics, for example if a client wishes to use a Java SMS library and call their own SMS service directly, this module may look substantially different.

```
public class SMSNotificationPublisher implements
SimMessagePublisher<SMSNotificationDEO> {
    public Class<SMSNotificationDEO> getDeoClass() {
        return SMSNotificationDEO.class;
    }
    public void handleMessage(SimMessageType messageType, SMSNotificationDEO deo,
boolean finalExecution) throws Exception {
        Properties props = getProperties();
        Session session = createSession(props);
        Message message = createMessage(session, deo);
        Transport.send(message);
    }

    private Properties getProperties() {
/* One can also set these properties in WebLogic instead */
        Properties props = new Properties();
        props.put("mail.smtp.host", HOST);
        props.put("mail.smtp.socketFactory.port", PORT);
        props.put("mail.smtp.socketFactory.class",
"javax.net.ssl.SSLSocketFactory");
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.port", PORT);
        return props;
    }
```

```
    private Session createSession(Properties props) {
        Session session = Session.getInstance(props, new
javax.mail.Authenticator() {
            protected PasswordAuthentication getPasswordAuthentication() {
                return new PasswordAuthentication(USERNAME, PASSWORD);
            }
        });
        return session;
    }
    private Message createMessage(Session session, SMSNotificationDEO deo) throws
Exception {
        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress(deo.getFrom()));
        message.setRecipients(Message.RecipientType.TO,
InternetAddress.parse(deo.getTo()));
        message.setText(deo.getMessage());
        return message;
    }
}
```

2

SIM Standalone Integration

For more information about RIB-based integrations, see "RIB-based Integration" in the *Oracle Retail Store Inventory Management Implementation Guide, Volume 2 – Integration with Oracle Retail Applications.*

Integration with external systems has been decoupled from SIM. This integration code now resides in a sim-int-<*>, sim-ext-<*> project, where, <*> refers to the external system.

The MPS coordinator thread executes periodically to build a work queue of SimMessage objects. When work is found and the worker limit has not been reached, the coordinator will start a new MPS worker thread.

These workers retrieve work from the queue and process messages using the MpsServices.processSimMessage() service operation. The MPS service locates the SimMessageHandler class registered for the message type and direction. The handler then processes the message data using the specific implementation. The message handlers are constructed using factories that can be configured to use custom implementations, defined in server.cfg. The standard handler factory requires message handlers to be registered, which occurs during execution of the application initializer defined in server.cfg.

The MPS system operation can be configured through settings in the MPS_CONFIG table. MPS workers also allow messages to be processed differently by message family and message direction, according to options in the MPS_WORKER_TYPE table.

SIM's staged message table is placed as a staging process between external systems, including those within the Oracle enterprise, and its internal logic. This allows SIM to be decoupled from the enterprise and easily integrated with other external systems. External messages (payloads) are converted into internal data exchange objects (DEOs) and written to a staging table (MPS_STAGED_MESSAGE). Timers gather messages from this table at configured time intervals and process them through business logic before the information reaches the final tables.

It is strongly recommended that customers use the MPS_STAGED_MESSAGE flow of information as described below. This is the only means to guarantee the data integrity of information entering the system. Bypassing this workflow may lead to incomplete and unusable data that could keep the system from functioning.

In order to integrate SIM with an external system, a java integrator program will need to be written. It is entirely up to the customer how they wish to retrieve information from the external system (database reads, file exports, etc) but the extracted information will need to be formatted into a payload or DEO in order to be imported into SIM. Writing this program will require importing the correct .jars that includes any SIM code being accessed.

Payload Integration Method

There are three simple coding steps to getting information into the system through the payload methodology: instantiate the payload and populate its attributes, instantiate the SimMessageRibInjector, call the inject API.

1. **Payload**

   Instantiate the appropriate payload and fill in its attributes with the exported data. Documentation on payloads can be found in *Oracle Retail Store Inventory Management Implementation Guide, Volume 2 – Integration with Oracle Retail Applications*.

   ```
   ItemSupRef payload = new ItemSupRef();
   payload.setSupplier(111);
   payload.setItem(444);
   ```

2. **Injector**

   Instantiate the injector so that the payload can get injected into the staging table.

   ```
   SimMessageRibInjector injector = new SimMessageRibInjector();
   ```

3. **Inject**

   Execute the injector so that the payload can get injected into the staging table. Using the item identifier as the business identifier for this object.

   ```
   injector.inject(SimRibMessageType.ITEMSUP_DEL, payload.getItem(), payload);
   ```

DEO Integration Method

The customer may also choose to bypass the payload route which may force them to fill in many payload attributes that are not used by SIM. Instead, the customer may simply choose to create DEOs and stage them into the message table. The steps are different for using DEOs.

1. **DEO**

   Instantiate the appropriate DEO and fill in its attributes with the exported data. DEO names and attributes parallel the similar payload. So though DEOs are not documented, the customer may use the payload documentation to determine the meaning of attributes in the payload section of *Oracle Retail Store Inventory Management Implementation Guide, Volume 2 – Integration with Oracle Retail Applications*.

   ```
   ItemSupRefDEO deo = new ItemSupRefDEO();
   deo.setSupplier(111);
   deo.setItem(444);
   ```

2. **Staged Message**

   Create a staged message around the DEO.

   ```
   MpsStagedMessage stagedMessage = BOFactory.createMpsStagedMessage();
   stagedMessage.setStoreId(deo.getBusinessStoreId());
   stagedMessage.setMessageType(SimRibMessageType.ITEMSUP_DEL);
   stagedMessage.setInbound(true);
   stagedMessage.setBusinessId(deo.getBusinessId());
   stagedMessage.setMessageDescription(deo.getMessageDescription());
   stagedMessage.setMessageData(XMLUtility.convertObjectToXml(deo));
   ```

3. **Insert the Message**

   Insert the staged message into the database.

```
List<MpsStagedMessage> stagedMessages = new ArrayList<>();
stagedMessages.add(stagedMessage);

DAOFactory.getMpsStagedMessageDao().insert(stagedMessages);
```

Creating a Custom Consumer

Do the following to create a custom consumer class:

1. Extend SimMessageConsumer.
2. Implement the consume(List<DataExchangeObject> deos) method.
3. Update SimMessageConsumerFactoryImpl to map the desired SimMessageType to your consumer.

Note

The message family to which the message type belongs will define which worker type options to use, which controls how many messages are processed, and how often those messages are processed.

Creating a Custom Stager

Do the following to create a custom stager:

1. Create a custom stager class by extending SimMessageStager.
2. Implement the SimMessageType getMesageType() method.
3. Implement the List<DataExchangeObject> getDataExchangeObjects() method.
4. Implement the String getStoreID() method.
5. Update SimMessageStagerFactoryImpl to return your custom Stager in the method for the desired business object.

Note

The message family to which the message type belongs will define which worker type options to use, which controls how many messages are processed, and how often those messages are processed.

3

Customizing Internationalization

Internationalization is the process of creating software that can be translated easily. Changes to the code are not specific to any particular market. SIM has been internationalized to support multiple languages.

This section describes customization settings and features of the software that enables the base application to be customized.

The PC client provides administrative screens for adding and altering translations.

Customizing a Language

The administrative section of the PC user interface has the ability to customize the translations of any language provided in the release of SIM. Users can create new keys that did not already exist in SIM. An understanding of English is necessary to use this feature of SIM. See the *Oracle Retail Store Inventory Management Users Guide* for more information about how to use these administrative features.

Do the following to add a new language to SIM:

1. Insert a Record into TRANSLATION_LOCALE.
2. Create a TRANSLATION_DETAIL Row.
3. Create a Translation.
4. Assign a Correct Locale to a User.

Insert a Record into TRANSLATION_LOCALE

As an example, if Scottish Gaelic is to be added as a new language, insert the record listed in Table: *TRANSLATION_LOCALE Record* into the database. The LANGUAGE is the two character ISO 639 code for the language. The COUNTRY is the ISO 3166 two-character country code. VARIANT is not required:

Table: *TRANSLATION_LOCALE Record*

Column	Value
LOCALE_ID	Insert a new unique ID here.
LANGUAGE	GD
COUNTRY	GB
VARIANT	Null
DESCRIPTION	Scottish Gaelic
UPDATE_DATE_TIME	The timestamp at the time of insert.

Create a TRANSLATION_DETAIL Row

The second step is to create a row in the TRANSLATION_DETAIL for every key in the TRANSLATION_KEY table. The following SQL inserts a null value as the translation for each key (assumes the new locale ID in previous step was 99):

```
insert into translation_detail
       (detail_id, locale_id, key_id, detail_value, update_date_time)
select to_char(translation_detail_seq.nextval),
       99,
       key.key_id,
       null detail_value,
       trunc(date_utils.get_current_gmt())
from translation_key key;
```

Create a Translation

Create a translation for each of the records inserted into TRANSLATION_DETAIL. The SIM PC client contains administrative screens to perform this step. See the *Oracle Retail Store Inventory Management Users Guide*.

Assign a Correct Locale to a User

Update user information so that SECURITY_USER.LANGUAGE and SECURITY_USER.COUNTRY contain the same two character codes (GD & GB) as the new locale that was inserted.

Rules

The MessageText parameter passed into the RulesInfo constructor within a Rule class is the language key used for translation. Messages from failed rule execution are translated.

Example: *ItemMustBeSellableRule*

```
public final class ItemMustBeSellableRule extends SimRule {
private static final RulesInfo RULE_FAILED
= new RulesInfo(ItemMessageText.ITEM_NOT_SELLABLE);
```

PC UI Labels and Titles

Everywhere within the entire SIM Swing framework that a label or title is used, the translation takes place automatically within the component. All title and label strings are the keys into the translation functionality. Failure to find a translation for the label or title simply returns the label or title itself.

Example: *ItemLookupPanel*

```
private RTextFieldEditor itemDescriptionEditor = new RTextFieldEditor("Item Description");
private RIntegerFieldEditor searchLimitEditor = new RIntegerFieldEditor("Item Description");
```

Error Messages and Exception

Error messages are long explanations of some validation or event failure that occurred in the system. The text message within the exception is the key into the translation functionality.

When dealing with error messages on the server, there should be no attempt at formatting or translation. Formatting and translation are strictly a client display responsibility. Simply create a business exception assigning the correct message and parameters and throw the error message to the client.

Dynamic Value Messages in Exceptions

BusinessExceptions are capable of handling dynamic values internally. The constructor that takes parameters uses these parameters to complete the dynamic message string. A very good example is found in the DecimalMask class which is assigned to any editor that edits decimals. It takes the entered value, minimum allowed, and maximum allowed and formats a message. The following example shows how those numbers are placed into a parameter array and passed in the construction of a business exception.

Example: *Invalid Range Exception*

```
Object[] params = new String[3];
params[0] = LocaleManager.getNumberFormatter().format(amount);
params[1] = LocaleManager.getNumberFormatter().format(minimumValue);
params[2] = LocaleManager.getNumberFormatter().format(maximumValue);
return new BusinessException(CommonMessageText.VALUE_NOT_IN_RANGE, params);
```

Dates

No work needs to be done by the developer to internationalize dates within the PC client. Both RDateField and RDateFieldEditor handle all of the logic, the developer simply needs to use a java.util.Date object. All conversion from text to Date and Date to text is handled by these editors. Example: *InventoryAdjustmentFilterDialog* shows how setDate() and getDate() are called on the editor. The date and calendar are displayed in the language and style of the locale set for the user who has logged on.

Example: *InventoryAdjustmentFilterDialog*

```
private RDateFieldEditor fromDateEditor = new RDateFieldEditor("From Date");
private RDateFieldEditor toDateEditor = new RDateFieldEditor("To Date");

public void setFilter(InventoryAdjustmentQueryFilter filter) throws Exception {
        model.setFilter(filter);

        fromDateEditor.setDate(filter.getFromDate());
        toDateEditor.setDate(filter.getToDate());
// Additional Code
  }

         private void doSearch() throws Exception {
              InventoryAdjustmentQueryFilter filter = model.getFilter();

filter.setDateRange(fromDateEditor.getDateAtStartOfDay(), toDateEditor.getDateAtEndOfDay());
             filter.setInventoryAdjustmentId(adjustmentEditor.getLongOrNull());
// Additional Code
}
    }
```

When formatting your own dates on PC Client (not through an editor), use the LocaleManager object, which has several methods for formatting dates. This automatically uses the currently assigned locale.

Another way to format dates is by using the DateMask, which allows the developer to assign a format type before formatting the date. DateMask can be used on any visual component that takes a mask. DateDisplayer formats a Date only in the SHORT format, but can used on any visual component that takes a displayer.

There are four dates allowed within the system following a standard java convention: SHORT, MEDIUM, LONG and FULL. All RDateFieldEditors within the application are assigned the SHORT format. In addition, the format values for each of the date formats will be standard JAVA format sequences by default. These default values can be overridden in the date.cfg file by defining the JAVA format sequence to use for the date

format type. Once the file is updated and the servers restarted, the various entry and display points in the application will begin using the newly defined format.

The Date.cfg file keys begin with the 2-character language and country code for the locale to be formatted. This is followed by the .dateType to be formatted. The format value must be in standard java convention. firstDayOfWeek determines the first day of week to be displayed on calendars. wirelessInput is the entry parser of handheld date entry fields. wirelessOutput is the date formatting of handheld date entry fields. wirelessDisplay is for handheld date display only.

Example: *Date.cfg*

```
# ENGLISH - AUSTRALIAN
#enAU.firstDayOfWeek=1
#enAU.entryDate=d/MM/yy
#enAU.shortDate=d/MM/yy
#enAU.mediumDate=d/MM/yyyy
#enAU.longDate=d MMMM yyyy
#enAU.fullDate=EEEE, d MMMM yyyy
enAU.monthPattern=MM-dd
enAU.wirelessInput=dd-MM-yy,ddMMyy
enAU.wirelessOutput=dd-MM-yy
enAU.wirelessDisplay=dd-mm-yy
```

Money

SimMoney is the data object that represents money within the system. Currency is a standard JAVA object that represents the type of money being represented. A Money object consists of a BigDecimal amount and a String currencyCode (though only Currency can set in the constructor). This is because once a money object is created, changing its currency would invalidate any amounts it represented. SIM does not handle currency conversion.

SimMoney is very similar to Date in that the RMoneyFieldEditor handles all of the internationalization for the user. RMoneyFieldEditor edits a Locale, Currency and BigDecimal in a generic fashion. The SimMoneyFieldEditor is a subclass that edits the SimMoney field directly. The following example demonstrates using the SimMoneyFieldEditor.

Note

Currency is handled separately from Locale by the editor. Currency describes the type of money being displayed while Locale indicates the desired language display format for the currency.

Example: *ItemTicketDetailPanel*

```
 private SimMoneyFieldEditor overridePriceEditor = new SimMoneyFieldEditor("Override Ticket
Price");
   private void loadEditorInformation(ItemTicket itemTicket) throws Exception {
             overridePriceEditor.setMoney(itemTicket.getOverridePrice());
          // Code To load rest of information
}

  private void doPriceModified() {
       try {
          model.getItemTicket().setOverridePrice(overridePriceEditor.getMoney());
       } catch (BusinessException buException) {
```

```
            // Deal With Exceptions
        }
    }
```

The control of the display of money is handled by a MoneyMask object assigned to the money related editors. All MoneyMask(s) are created through a MoneyMaskFactory. This factory is defined in the common.cfg configuration file. You can create a custom money mask factory returning the desired money formatting and simply point the configuration file to it in the same manner as all other factories described previously in the document. This allows the customization of money display and entry without altering original source code.

Wireless Internationalization

Wireless Internationalization is not easily customizable. Many of the previous techniques described will automatically update the same content on the HH forms, but that is not always the case. The following documentation describes how to code HH forms to take advantage of the built in localization that exists in SIM. Wireless customization requires original source code.

Internationalization is handled by different approaches based on where in the Wireless code the translation is needed.

Forms

In the xml files that define the handheld forms, the display of labels is usually surrounded with a $[ ] indicator. In the Form<name>.java classes, the text within the brackets is wrapped by an AppGlobal.getString() call which translates the text.

Example: *Screen_ContainerLookupScreen.xml*

```
<Screen name="ContainerLookupDetail">
  <LogicalScreen>
    <field name="containerId" type="string" length="21"/>
    // More Field Names…
    <field name="asnNumber" type="string" length="21"/>
  </LogicalScreen>

  <PhysicalScreens deviceclass="dnw">
    <PhysicalScreen seq="0">
    <label y="0" x="0" width="21" height="1" style=".heading1">
         $[Lookup Results]</label>
    <label y="2" x="0" width="11" height="1" >
         $[Container]$[wireless.delimiter]</label>
     <label y="2" x="11" width="21" height="1" name="containerId"
         field="containerId"/>
     <label y="3" x="0" width="8" height="1" >
         $[Status]$[wireless.delimiter]</label>
     // More labels…
     <label y="12" x="13" width="21" height="1" name="totalCases"
         field="totalCases" />

     <cmdkey key="&exit;" y="16" x="0" height = "0" width="0"
         name="return" action="callMethod" target="doExit"/>
      </PhysicalScreen>
  </PhysicalScreens>
</Screen>
```

Example: *Form_dnw_ContainerLookupDetail_0.java*

```
public Form_dnw_ContainerLookupDetail_0(String id, EventHandler_ContainerLookupDetail handler)
  throws WaveLinkError {
 super(id, false, handler);

 add(new RFPrintLabel(wlio, AppGlobal.getString("Container") +
     AppGlobal.getString("wireless.delimiter"), 0, 2, 11, 1, termWidth));
 add(new RFPrintLabel(wlio, AppGlobal.getString("Status") +
     AppGlobal.getString("wireless.delimiter"), 0, 3, 8, 1, termWidth));
```

EventHandlers

Every event handler extends from SimEventHandler, which contains numerous methods to assist in formatting and retrieving information in an internationalized manner. These helper methods in the superclass should always be used to perform these types of tasks when coding event handlers.

Key methods include:

SetFormDate()

Formats a date for the locale and country and places it in the form.

SetFormInteger()

Formats an integer for the locale and places it in the form.

SetFormDecimal()

Formats a decimal for the locale and places it in the form.

SetFormQuantity()

Formats a quantity for the locale and places it in the form.

GetText()

Retrieves the translation of the text.

GetLabel()

Retrieves the translation of the text followed by the label delimiter

GetMessage()

Retrieves the translation of the message

GetFormInteger()

Retrieves entered text as an integer

GetFormDecimal()

Retrieves entered text as a decimal

GetFormQuantity()

Retrieves entered text as a quantity

HandleException()

Handles displaying an exception (translating the message).

Here are some examples of standard eventhandler code using these internationalization methods. In the first example, we are translating the label **Select PO From**.

Example: *EventHandler_DirectDeliverySelectPO*

```
protected void onFormOpen() {
  try {
    setFormData(FIELD_INSTRUCTIONS, getLabel("Select PO From"));
    setFormData(FIELD_SUPPLIER,
        DirectDeliveryWirelessUtility.getContext().getSource().getName());
    // Some Code
  } catch (Exception e) {
    handleException(e);
  }
}
```

Example: *EventHandler_ContainerLookupDetail*

```
EventHandler_ContainerLookupDetail
    protected void onFormOpen() {
        try {
            LogService.debug(this, this.getClass().getSimpleName() + ".onFormOpen()");
            ShipmentCartonVO cartonVO = (ShipmentCartonVO)
SimWirelessRepository.getValue(ContainerWirelessKeys.USER_CONTEXT_CONTAINER);
            SourceVO fromLocation = cartonVO.getFromLocation();

            setFormText("containerId", cartonVO.getCartonId());
            setFormText("status", getText(cartonVO.getStatus().toString()));
            setFormText("fromLocation", fromLocation.getId() + " " + fromLocation.getName());
            setFormText("asnNumber", cartonVO.getAsnId());
            setFormDate("eta", cartonVO.getEta());
            setFormDate("receiptDate", cartonVO.getReceiveDate());
            setFormText("receiptTime",
LocaleWirelessUtility.formatTime(cartonVO.getReceiveDate()));
            setFormInteger("totalCases", cartonVO.getNumberOfCasesExpected());
        } catch (Exception e) {
            handleException(e);
        }
    }
```

<name>WirelessUtility

Every wireless utility class extends from WirelessUtility, which like SimEventHandler, contains numerous methods to assist in formatting and retrieving information in an internationalized manner. These helper methods in the superclass should always be used to perform these types of tasks when coding utility methods. Read the javadoc on WirelessUtility for complete descriptions.

Key methods include:

GetText()

Retrieves the translation of the text.

Alert()

Handles displaying an alert message (translating the message).

GetLabel()

Retrieves the translation of the text followed by the label delimiter.

GetMessage()

Retrieves the translation of the message.

HandleException()

Handles displaying an exception (translating the message).

Here is an example of standard utility code using these internationalization methods.

Example: *StockCountWirelessUtility*

```
public static String getDescription(StockCount stockCount) {
    StringBuilder buffer = new StringBuilder();
    try {
        if (stockCount.getType() == StockCountType.PROBLEM_LINE) {
            buffer.append(getLabel("wireless.problemLineABBV"));
        }
        String text = stockCount.getCountDescription().trim();
        int index = text.lastIndexOf("(");
        if (index < 0) {
            buffer.append(text);
        } else {
            buffer.append(text.substring(index));
            buffer.append(StringConstants.SPACE);
            buffer.append(text.substring(0, index).trim());
        }
        return buffer.toString();
    } catch (Exception e) {
        return getText("wireless.noDescABBV");
    }
}
```

In Example: *StockCountWirelessUtility* business logic is required to create a stock count description. The utility uses the getLabel() and getText() helper methods to guarantee that the text is translated as the description is being built.

LocaleWirelessUtility

If there are no superclass helper methods available for what you want to accomplish, you can directly use the LocaleWirelessUtility to perform internationalized functions.

Wireless Labels

Wireless labels have an additional consideration that is not needed on the PC. The width of a wireless form, which is displayed on a handheld device, is very narrow. That means only a small amount of space is allocated to a label. The English labels used as translation keys are defined by the space they take up. Oracle Retail suggests that instead of using a standard English label such as **status** for the key, use **wireless.status** instead, so that it is clear in the GUI administrative screen that the translation being supplied is for the wireless device.

4

Web Services

SIM has exposed web services to grant easier access to business logic within SIM. Web services are easily accessed from custom clients or from external systems in general. A generic approach like this enables SIM to be more modular and allow retailers to more easily integrate with certain SIM transactions.

SIM provides a number of Web services that expose various parts of the Store Inventory Management server's functionality. These Web services generally serve two purposes:

- Enable integration with external back-end systems
- Provide functionality to external client programs

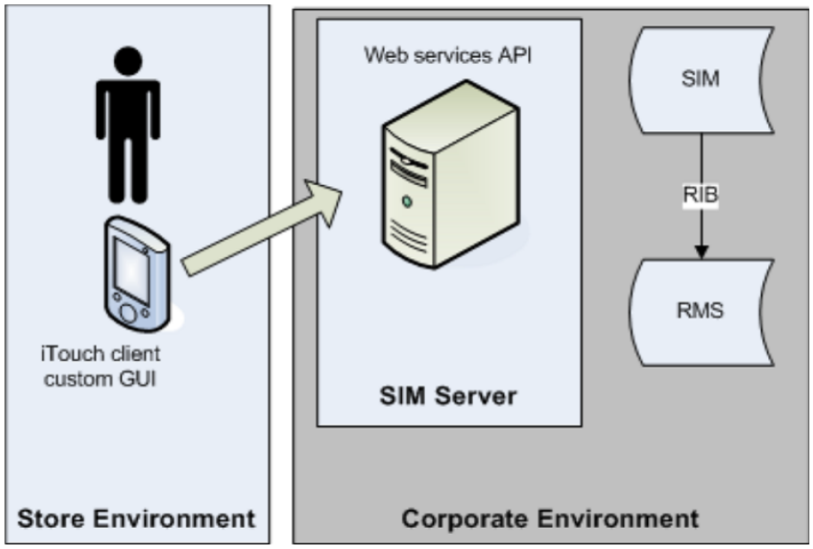
Some of the Store Inventory Management Web services are primarily intended for the backend of third-party systems, such as point-of-sale systems, to interact with Store Inventory Management.

Other Web services in Store Inventory Management are primarily intended to be used by mobile client applications built by Store Inventory Management customers or other third parties. Examples of these are Store to Store Transfer and Return to Warehouse.

The Web services are not strictly segregated. There are some Web services that serve both purposes (for example, Item Lookup), and all Web services should be considered usable by any external system, whether or not it is a client program.

The Figure: Web *Service Process Flow* demonstrates a Web service process flow:

Figure: *Web Service Process Flow*



Asynchronous Operations

Most web services in Store Inventory Management work synchronously; the web services accept input, validate and process the input, then return results. However, some web services in Store Inventory Management work on bulk data and work asynchronously (for example, sales processing). These web services accept input of many data items at once, but do not process them all immediately. Instead the web services perform minor validation on the data, store the data, and then return to the caller. After the web service

returns to the caller, Store Inventory Management's asynchronous processing framework initiates the remainder of the processing. The Web services that follow this pattern mention refer to this functionality in their service descriptions in the WSDL.

Application Programming Interface (API)

Store Inventory Management's Web services are SOAP-based Web services. All API services are defined in one WSDL and detailed information about these services is provided in the documentation produced by the Retail Technology Group.

Implementation

Web service implementation on the Store Inventory Management server is done in a top-down fashion, meaning that a Web service's API is first defined in WSDL and XSD, and then supporting Java classes are generated from the WSDL using Oracle tools. These classes are then used to connect the web services to existing Store Inventory Management server functionality.

Security

Store Inventory Management web services are installed out-of-the-box with basic HTTP authentication enabled. If a different scheme is required, the services can be secured at a client site with any SOAP-friendly Web service authentication mechanism, for example, WS-Security.

Deployment

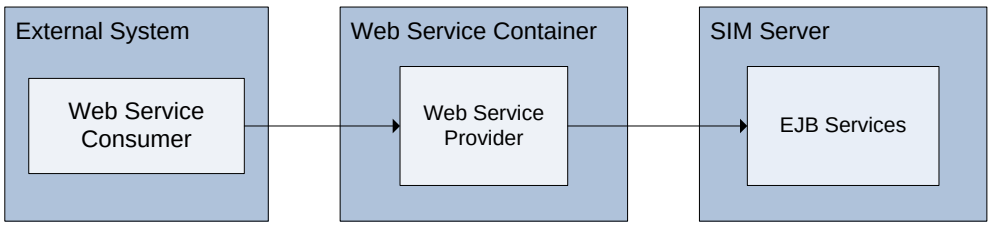
The Store Inventory Management server application is deployed as a Java EE application (sim.ear). The Store Inventory Management Web service WSDL, XSD, generated Java classes, and implementation Java classes are all deployed as a Web module (sim-ws.war) inside the server application. To access the WSDL in a running Store Inventory Management instance, go to the following URL:

`<http://<orsim-server-url>:<orsim-server-port>/sim-ws/simWebService?WSDL`

Extensibility

SIM does not support the extension or customization of SOAP web services. The WSDL and XSD represent a hard contract and thus any extension would automatically be a violation of that contract. SIM also puts very little logic in the web service layer. Simple validation of the SOAP message is followed by converting the data to an internal business object and making an EJB service call (which manages transactional state).

Figure: *EJB Services*



If customized functionality is required, the customer should build brand new web services to fit their needs. In this case, the customer may build their own specific WSDL and XSD for a specific SOAP web service, or it could be REST Service. The web service

provider the customer writes can deploy in their own web service container, or they could deploy it in a SIM server container. This provider should be responsible for validating the incoming payload and calling the appropriate SIM EJB services to finish the task involved. All of SIM's EJB services deployed on our service are documented. From there, any business logic or other processing that needs to be customized should follow all the normal extension patterns found throughout this guide.

A

Appendix: Code Examples

This appendix includes code samples referenced in **Example Code** in Chapter 1, "Customization".

Example: *CustomSerialNumber*

```
package oracle.retail.sim.closed.customuin;

import oracle.retail.sim.closed.business.BusinessObject;
import oracle.retail.sim.closed.common.BusinessException;
import oracle.retail.sim.closed.uin.UINStatus;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;

/*
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
public class CustomSerialNumber extends BusinessObject {
    private static final long serialVersionUID = 7328521596108816814L;

    private Long id;
    private UINStatus status;

    public void setId(Long id) {
        if (id != null) {
            this.id = id;
        }
    }

    public Long getId() {
        return id;
    }

    public UINStatus getStatus() {
        return status;
    }

    public void setStatus(UINStatus status) throws BusinessException {
        checkForNullParameter("Status", status);
        executeRule("setStatus", status);
        this.status = status;
    }

    public boolean equals(Object object) {
        if (object == this) {
            return true;
        }
        if (object == null || object.getClass() != getClass()) {
            return false;
        }
        CustomSerialNumber other = (CustomSerialNumber) object;
        EqualsBuilder builder = new EqualsBuilder();
        builder.append(id, other.id);
        return builder.isEquals();
    }

    public int hashCode() {
```

```
        HashCodeBuilder builder = new HashCodeBuilder();
        builder.append(id);
        return builder.toHashCode();
    }
}
```

Example: *CustomUINBean*

```
package oracle.retail.sim.closed.ejb;

import javax.ejb.*;
import oracle.retail.sim.closed.abstractbean.AbstractSimServiceBean;
import oracle.retail.sim.closed.business.ServerServiceFactory;
import oracle.retail.sim.closed.logging.LogNames;
import oracle.retail.sim.closed.logging.LogService;
import oracle.retail.sim.closed.util.CompressedObject;
import oracle.retail.sim.closed.util.SimSession;
import oracle.retail.sim.closed.util.UniversalContext;

/**
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
@Stateless(mappedName = "CustomUINBean")
public class CustomUINBean extends AbstractSimServiceBean implements CustomUINInterface {
    public CompressedObject moveSerialNumbersToInStock(CompressedObject compressed0,
                               CompressedObject compressedSimSession) throws Exception {
        long startTime = System.currentTimeMillis();
        if (LogService.isDebugEnabled(LogNames.SERVICE_TIMINGS)) {
            LogService.debug(LogNames.SERVICE_TIMINGS,
                  "Starting service method: CustomUINBean.moveSerialNumbersToInStock()");
        }
        try {
            java.util.List<oracle.retail.sim.closed.custom.CustomSerialNumber> arg0 =
           (java.util.List<oracle.retail.sim.closed.custom.CustomSerialNumber>)
                   compressed0.recoverObject();
            UniversalContext.setSession((SimSession) compressedSimSession.recoverObject());
            new CustomerUINServerServices().moveSerialNumbersToInStock(arg0);
            return new CompressedObject(null);
        } catch (Throwable t) {
            handleException(t, true);
            return null;
        } finally {
            completeServiceContext();
            long endTime = System.currentTimeMillis();
            if (LogService.isDebugEnabled(LogNames.SERVICE_TIMINGS)) {
                LogService.debug(LogNames.SERVICE_TIMINGS, "Took " + (endTime - startTime)
                    + "ms. for invocation of: CustomUINBean.moveSerialNumbersToInStock()");
            }
        }
    }

}
```

Example: *CustomUINCountTableEditor*

```
package oracle.retail.sim.shared.swing.custom;

import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JComponent;
import javax.swing.JTextField;
import oracle.retail.sim.closed.swing.table.SimPopupTableEditor;
```

```
import oracle.retail.sim.closed.swing.table.SimTableEditorEventAdaptor;
import oracle.retail.sim.closed.swing.table.SimTableEditorListener;
import oracle.retail.sim.closed.swing.tableeditor.PopupTableEditorListener;

/*******************************************************************************************
 * Table editor that handles a boolean true/false value, displays a non-editable check box.
 * If the cell
 * is clicked twice, a popup dialog is triggered and the data model is passed to the dialog.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 *******************************************************************************************/

public class CustomUINCountTableEditor extends JTextField implements SimPopupTableEditor {
    private static final long serialVersionUID = -4236326754891359417L;

    private PopupTableEditorListener listener;
    private SimTableEditorEventAdaptor eventAdaptor;
    private Object model;

    /*******************************************************************************************
 * Constructors
 *******************************************************************************************/

    public CustomUINCountTableEditor(PopupTableEditorListener listener) {
        eventAdaptor = new SimTableEditorEventAdaptor(this);
        addMouseListener(buildMouseListener());
        setHorizontalAlignment(RIGHT);
        setOpaque(true);
        setEnabled(false);
        setListener(listener);
    }

    /*******************************************************************************************
 * Basic Property Methods of a Table Editor
 *******************************************************************************************/

    public Class getValueClass() {
        return Integer.class;
    }

    public void setValueClass(Class valueClass) {
        // Ignore
    }

    public void setModel(Object model) {
        this.model = model;
    }

    public JComponent getComponent() {
        return this;
    }

    private void setListener(PopupTableEditorListener listener) {
        this.listener = listener;
    }

    public void setCoordinates(int row, int column) {
        // Ignore
    }
```

```
/*****************************************************************************************
 * Get, Set and Check the data value of the editor
 *****************************************************************************************/

    public Object getValue() {
        return getText();
    }

    public void setValue(Object value) {
        if (value != null) {
            setText(value.toString());
            popupDialog();
        }
    }

    public boolean checkValue() {
        return true;
    }


/*****************************************************************************************
 * Table Editor Listener

*****************************************************************************************/

    public void addTableEditorListener(SimTableEditorListener listener) {
        eventAdaptor.addTableEditorListener(listener);
    }

    public void removeTableEditorListener(SimTableEditorListener listener) {
        eventAdaptor.removeTableEditorListener(listener);
    }

    /*****************************************************************************************
 * Determines if keystroke is invalid for field. Always false for a boolean editor.
 *****************************************************************************************/
    public boolean isInvalidKeystroke(KeyEvent event) {
        return false;
    }

    /*****************************************************************************************
 * Mouse Listener That Pops The Dialog
 *****************************************************************************************/
    private MouseListener buildMouseListener() {
        return new MouseAdapter() {
            public void mouseClicked(MouseEvent event) {
                if (event.getClickCount() == 2) {
                    popupDialog();
                }
            }
        };
    }


/*****************************************************************************************
 * Display Dialog
 *****************************************************************************************/

    private void popupDialog() {
        if (listener != null) {
            listener.popupDialog(model);

            CustomUINCreateWrapper wrapper = (CustomUINCreateWrapper) model;
```

```
            setText(String.valueOf(wrapper.getSerialNumberCount()));
        }
    }
}
```

Example: *CustomUINCreateDialog*

```
package oracle.retail.sim.shared.swing.custom;

import java.awt.GridBagLayout;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import oracle.retail.sim.closed.application.Application;
import oracle.retail.sim.closed.common.ErrorKey;
import oracle.retail.sim.closed.simclient.util.SimClientErrorKey;
import oracle.retail.sim.closed.swing.dialog.RDialog;
import oracle.retail.sim.closed.swing.displayer.AttributeDisplayer;
import oracle.retail.sim.closed.swing.editor.RDisplayLabelEditor;
import oracle.retail.sim.closed.swing.event.RActionEvent;
import oracle.retail.sim.closed.swing.event.REventListener;
import oracle.retail.sim.closed.swing.panel.RPanel;
import oracle.retail.sim.closed.swing.table.SimTable;
import oracle.retail.sim.closed.swing.table.SimTableAttribute;
import oracle.retail.sim.closed.swing.table.SimTableDefinition;
import oracle.retail.sim.closed.swing.table.SimTableEditorEvent;
import oracle.retail.sim.closed.swing.table.SimTableEditorListener;
import oracle.retail.sim.closed.swing.table.SimTablePane;
import oracle.retail.sim.closed.swing.table.SimTableSortAttribute;
import oracle.retail.sim.closed.swing.util.GridTool;
import oracle.retail.sim.closed.swing.util.UIException;
import oracle.retail.sim.closed.swing.widget.RButton;
import oracle.retail.sim.closed.uin.SerialNumberValue;
import oracle.retail.sim.closed.uin.UINStatus;
import oracle.retail.sim.closed.util.CollectionUtil;
import oracle.retail.sim.shared.swing.core.SimNavigation;
import oracle.retail.sim.shared.swing.uin.SerialNumberProperty;
import oracle.retail.sim.shared.swing.uin.SerialNumberWrapper;

/*******************************************************************************************
 * This dialog handles entering UIN information for line item from the main UIN Create screen.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 *******************************************************************************************/

public class CustomUINCreateDialog extends RDialog implements REventListener {
    private static final long serialVersionUID = -7885503433032637549L;

    private CustomUINCreateDialogModel model = new CustomUINCreateDialogModel();

    private RDisplayLabelEditor itemEditor = new RDisplayLabelEditor("Item");
    private RDisplayLabelEditor itemDescEditor = new RDisplayLabelEditor("Item Description");

    private CustomUINCreateDialogTableEditor serialNumberEditor
                                            = new CustomUINCreateDialogTableEditor();

    private SimTable serialNumberTable = new SimTable(new CreateUINTableDefinition());
    private SimTablePane serialNumberTablePane = new SimTablePane(serialNumberTable);

    private RButton doneButton = new RButton(SimNavigation.DONE);
    private RButton addButton = new RButton(SimNavigation.ADD);
```

```
    private RButton deleteButton = new RButton(SimNavigation.DELETE);
    private RButton cancelButton = new RButton(SimNavigation.CANCEL);

    /**********************************************************************************************
 * Build Dialog
 **********************************************************************************************/
    public CustomUINCreateDialog() {
        super(Application.getFrame());
        setStatusBarVisible(false);
        setTitle("UIN");
        setSize(550, 400);
        initializeWidgets();
        layoutContent();
        centerWindow();
    }

    private void initializeWidgets() {
        serialNumberEditor.addTableEditorListener(buildSerialNumberValueListener());

        serialNumberTable.setMultipleRowSelectionMode();

        doneButton.registerAction(this, SimNavigation.DONE);
        addButton.registerAction(this, SimNavigation.ADD);
        deleteButton.registerAction(this, SimNavigation.DELETE);
        cancelButton.registerAction(this, SimNavigation.CANCEL);
    }

    private void layoutContent() {
        addButton(doneButton);
        addButton(addButton);
        addButton(deleteButton);
        addButton(cancelButton);

        RPanel headerPanel = new RPanel(new GridBagLayout());
        headerPanel.add(itemEditor, GridTool.constraints(0, 0, 1, 1, 1, 0, 0, 3, 3, 0, 0, 10));
        headerPanel.add(itemDescEditor, GridTool.constraints(1, 0, 1, 1, 1, 0, 0, 3, 3, 0, 0, 0));

        RPanel mainPanel = new RPanel(new GridBagLayout());
        mainPanel.add(headerPanel, GridTool.constraints(0, 0, 1, 1, 1, 0, 0, 3, 0, 0, 5, 0));
        mainPanel.add(serialNumberTablePane, GridTool.constraints(0, 1, 1, 1, 1, 1, 0, 3, 0, 0, 0,
0));

        setContentPane(mainPanel);
    }


/**********************************************************************************************
 * Build Dialog
 **********************************************************************************************/

    public void setWrapper(CustomUINCreateWrapper wrapper) {
        model.setWrapper(wrapper);
        try {
            itemEditor.setData(model.getItemId());
            itemDescEditor.setData(model.getItemDescription());
            serialNumberTable.setRows(model.getSerialNumberWrappers());
        } catch (Throwable exception) {
            displayException(exception);
        }
    }


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

```
 * Listen to serial number entry field in order to validate UIN entered
 ********************************************************************************************/

    private SimTableEditorListener buildSerialNumberValueListener() {
        return new SimTableEditorListener() {
            public void performTableEditorEvent(SimTableEditorEvent event) {
                if (event.isFocusLostEvent()) {
                    Set<String> serialNumberSet = CollectionUtil.newHashSet();
                    List<SerialNumberWrapper> wrappers = serialNumberTable.getAllRowData();
                    for (SerialNumberWrapper wrapper : wrappers) {
                        SerialNumberValue serialNumber = wrapper.getSerialNumberValue();
                        if (serialNumber != null) {
                            if (serialNumberSet.contains(serialNumber.getUin())) {
                                serialNumberTable.removeRow(wrapper);
                                displayException(
                                       new UIException(ErrorKey.UIN_DUPLICATE_ERROR_PC));
                                continue;
                            }
                            if (serialNumber.getStatus() != UINStatus.UNCONFIRMED) {
                                serialNumberTable.removeRow(wrapper);
                                displayException(new UIException(
                                                        "Serial Number already exists!"));
                                continue;
                            }
                            serialNumberSet.add(serialNumber.getUin());
                        }
                    }
                    // Temporary fix - only add the new row if the button panel does not have
                    if (isMouseOverButton()) {
                        return;
                    }
                    try {
                        doAddRow();
                    } catch (Throwable exception) {
                        displayException(exception);
                    }
                }
            }
        };
    }


/*********************************************************************************************
 * Handle Actions
 ********************************************************************************************/
    public void performActionEvent(RActionEvent event) {
        String command = event.getEventCommand();
        try {
            if (command.equals(SimNavigation.DONE)) {
                doDone();
            } else if (command.equals(SimNavigation.ADD)) {
                doAddRow();
            } else if (command.equals(SimNavigation.DELETE)) {
                doDeleteRow();
            } else if (command.equals(SimNavigation.CANCEL)) {
                doCancel();
            }
        } catch (Throwable exception) {
            displayException(exception);
        }
    }
```

```
/**********************************************************************************************
 * Add Action
 **********************************************************************************************/

    private void doAddRow() {
        serialNumberTable.stopEditing();
        List<SerialNumberWrapper> wrappers = serialNumberTable.getAllRowData();
        for (SerialNumberWrapper wrapper : wrappers) {
            if (wrapper.getSerialNumberValue() == null) {
                return;
            }
        }
        serialNumberTable.addRow(model.createSerialNumberWrapper());
        serialNumberTable.editCellInLastRow(SerialNumberProperty.SERIAL_NUMBER);
    }


/**********************************************************************************************
 * Delete Action
 **********************************************************************************************/

    private void doDeleteRow() throws Exception {
        List<SerialNumberWrapper> selectedWrappers = serialNumberTable.getAllSelectedRowData();
        if (selectedWrappers.isEmpty()) {
            throw new UIException(SimClientErrorKey.NO_ROWS_SELECTED_DELETE);
        }
        for (SerialNumberWrapper wrapper : selectedWrappers) {
            serialNumberTable.removeRow(wrapper);
        }
        serialNumberTable.refreshTable();
    }


/**********************************************************************************************
 * Done Action
 **********************************************************************************************/

    private void doDone() throws Exception {
        serialNumberTable.stopEditing();
        List<SerialNumberWrapper> wrappers = serialNumberTable.getAllRowData();
        model.saveSerialNumbers(wrappers);
        closeWindow();
    }


/**********************************************************************************************
 * Override method to alter functionality. Default stops editing and closes window.
 **********************************************************************************************/
    private void doCancel() {
        serialNumberTable.stopEditing();
        closeWindow();
    }


/**********************************************************************************************
 * UIN Table Definition
 **********************************************************************************************/

    private class CreateUINTableDefinition extends SimTableDefinition {

        public Class getDataClass() {
            return SerialNumberWrapper.class;
```

```
        }

        public List<SimTableSortAttribute> getSortAttributes() {
            return Collections.singletonList(new SimTableSortAttribute("serialNumberValue"));
        }

        public List<SimTableAttribute> getAttributes() {
            List<SimTableAttribute> attributes = CollectionUtil.newArrayList(1);
            attributes.add(new SimTableAttribute(model.getUINLabel(), "serialNumberValue",
                           new AttributeDisplayer("uin"), serialNumberEditor));
            return attributes;
        }
    }
}
```

Example: *CustomUINCreateDialogModel*

```
package oracle.retail.sim.shared.swing.custom;

import java.util.List;
import oracle.retail.sim.closed.business.FunctionalArea;
import oracle.retail.sim.closed.uin.SerialNumberValue;
import oracle.retail.sim.closed.uin.UINType;
import oracle.retail.sim.closed.util.CollectionUtil;
import oracle.retail.sim.closed.util.RkConfigManager;
import oracle.retail.sim.shared.swing.core.SimScreenModel;
import oracle.retail.sim.shared.swing.uin.SerialNumberWrapper;

/*****************************************************************************************
 * The business logic model for the Custom UIN Create Dialog.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 *****************************************************************************************/

public class CustomUINCreateDialogModel extends SimScreenModel {

    private CustomUINCreateWrapper lineItemWrapper;

    public void setWrapper(CustomUINCreateWrapper wrapper) {
        lineItemWrapper = wrapper;
    }

    public String getItemId() {
        if (lineItemWrapper != null) {
            return lineItemWrapper.getStockItem().getId();
        }
        return null;
    }

    public String getItemDescription() {
        if (lineItemWrapper != null) {
            if (RkConfigManager.isItemShortDescription()) {
                return lineItemWrapper.getStockItem().getShortDescription();
            }
            return lineItemWrapper.getStockItem().getLongDescription();
        }
        return null;
    }

    public List<SerialNumberWrapper> getSerialNumberWrappers() {
        List<SerialNumberWrapper> wrappers = CollectionUtil.newArrayList();
```

```
            for (SerialNumberValue value : lineItemWrapper.getSerialNumbers()) {
                SerialNumberWrapper wrapper = createSerialNumberWrapper();
                wrapper.setSerialNumberValue(value);
                wrapper.setDefaultAction();
                wrapper.setValidated();

                wrappers.add(wrapper);
            }
            if (wrappers.isEmpty()) {
                wrappers.add(createSerialNumberWrapper());
            }
            return wrappers;
        }

        public SerialNumberWrapper createSerialNumberWrapper() {
            return new SerialNumberWrapper(FunctionalArea.MANUAL,
                      lineItemWrapper.getStockItem().getId(), getUINType(), getUINLabel());
        }

        public UINType getUINType() {
            if (lineItemWrapper != null) {
                return lineItemWrapper.getStockItem().getUINType();
            }
            return UINType.SERIAL;
        }

        public String getUINLabel() {
            if (lineItemWrapper != null) {
                return lineItemWrapper.getStockItem().getUINLabel();
            }
            return null;
        }

        public void saveSerialNumbers(List<SerialNumberWrapper> wrappers) {
            List<SerialNumberValue> serialNumbers = CollectionUtil.newArrayList();
            for (SerialNumberWrapper wrapper : wrappers) {
                SerialNumberValue value = wrapper.getSerialNumberValue();
                if (value != null) {
                    serialNumbers.add(value);
                }
            }
            lineItemWrapper.setSerialNumbers(serialNumbers);
        }
    }
```

Example: *CustomUINCreateDialogTableEditor*

```
package oracle.retail.sim.shared.swing.custom;

import java.awt.event.FocusEvent;
import java.awt.event.FocusListener;
import java.awt.event.KeyEvent;
import javax.swing.JComponent;
import oracle.retail.sim.closed.application.Application;
import oracle.retail.sim.closed.business.ClientServiceFactory;
import oracle.retail.sim.closed.business.FunctionalArea;
import oracle.retail.sim.closed.common.BusinessException;
import oracle.retail.sim.closed.common.ErrorKey;
import oracle.retail.sim.closed.common.locale.StringConstants;
import oracle.retail.sim.closed.common.type.AbstractDisplayer;
import oracle.retail.sim.closed.locale.StringUtility;
import oracle.retail.sim.closed.locale.Translator;
import oracle.retail.sim.closed.simclient.util.SimClientErrorKey;
```

```
import oracle.retail.sim.closed.swing.dialog.RErrorDialog;
import oracle.retail.sim.closed.swing.displayer.AttributeDisplayer;
import oracle.retail.sim.closed.swing.logging.UILog;
import oracle.retail.sim.closed.swing.sim.SimRepository;
import oracle.retail.sim.closed.swing.table.SimTable;
import oracle.retail.sim.closed.swing.table.SimTableEditor;
import oracle.retail.sim.closed.swing.table.SimTableEditorEventAdaptor;
import oracle.retail.sim.closed.swing.table.SimTableEditorListener;
import oracle.retail.sim.closed.swing.util.UIErrorKey;
import oracle.retail.sim.closed.swing.util.UIExceptionFactory;
import oracle.retail.sim.closed.swing.widget.RTextField;
import oracle.retail.sim.closed.uin.SerialNumberValue;
import oracle.retail.sim.shared.swing.core.SimName;
import oracle.retail.sim.shared.swing.uin.SerialNumberWrapper;

/*******************************************************************************************
 * Table Editor for editing a CustomerSerialNumber business object. The model for the table
 * row that uses this editor
 * must be a UINWrapper.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 *******************************************************************************************/

public class CustomUINCreateDialogTableEditor extends RTextField
                                              implements SimTableEditor, FocusListener {
    private static final long serialVersionUID = -1463881160730583097L;

    private AbstractDisplayer displayer;
    private SimTableEditorEventAdaptor eventAdaptor;
    private SerialNumberWrapper wrapper;
    private SerialNumberValue serialNumberValue;
    private String lastCheckedSerialNumber = StringConstants.EMPTY;
    private FocusEvent lastFocusEvent;
    private boolean isErrorState;
    private int row = -1;
    private int column = -1;

    public CustomUINCreateDialogTableEditor() {
        eventAdaptor = new SimTableEditorEventAdaptor(this);
        displayer = new AttributeDisplayer("uin");
        setIdentifier(SimName.SERIAL_NUMBER);
        setMargin(null);
        setBorder(null);
        addFocusListener(this);
    }

    public Class getValueClass() {
        return SerialNumberValue.class;
    }

    public void setValueClass(Class valueClass) {
        // Ignore
    }

    public void setModel(Object model) {
        wrapper = (SerialNumberWrapper) model;
    }


/*******************************************************************************************
 * Assign coordinates to the editor.
```

```
 ********************************************************************************************/
    public void setCoordinates(int row, int column) {
        this.row = row;
        this.column = column;
    }


/*********************************************************************************************
 * Reactivate editing within the table cell.
 ********************************************************************************************/
    private void reactivateEditing(Object object) {
        if (object instanceof SimTable) {
            SimTable table = (SimTable) object;
            try {
                if (table.getSelectedRow() != row) {
                    table.setRowSelectionInterval(row, row);
                }
                table.editCellAt(row, column);
            } catch (Throwable ex) {
                UILog.debug(getClass(), ex.getMessage());
            }
        }
    }

    public Object getValue() {
        if (isErrorState()) {
            return null;
        }
        return serialNumberValue;
    }

    public void setData(Object value) {
        if (value instanceof SerialNumberValue) {
            setValue(value);
            return;
        }
        throw new IllegalArgumentException(
                              "CustomSerialNumberTableEditor only edits UINValue!");
    }

    public void setValue(Object value) {
        if (value instanceof String) {
            setText((String) value);
            checkValue();
        } else if (value instanceof SerialNumberValue) {
            serialNumberValue = (SerialNumberValue) value;
            setText(displayer.getDisplayText(serialNumberValue));
        } else if (value == null) {
            serialNumberValue = null;
            clear();
        }
        eventAdaptor.fireTypeEditorEvent();
    }

    public JComponent getComponent() {
        return this;
    }

    public boolean checkValue() {
        String enteredSerialNumber = getText();

        setErrorState(false);
```

```
        if (StringUtility.isNullOrEmpty(enteredSerialNumber)) {
            if (serialNumberValue == null) {
                return true;
            }
            displayErrorWithReset(SimClientErrorKey.ITEM_BLANK_ERROR);
            return false;
        }

        try {
            if (serialNumberValue != null
               && enteredSerialNumber.equals(serialNumberValue.getUin())
               && enteredSerialNumber.equals(lastCheckedSerialNumber)) {
                return true;
            }
            SerialNumberValue tempValue =
                  ClientServiceFactory.getUINServices().findSerialNumberValueOrCreate(
                      SimRepository.getStoreId(), wrapper.getItemId(), enteredSerialNumber,
                     wrapper.getType(), FunctionalArea.MANUAL);
            if (tempValue == null) {
                String message = Translator.getMessage(ErrorKey.UIN_NOT_FOUND_FOR_ITEM,
                 wrapper.getType().getDescription(), enteredSerialNumber, wrapper.getItemId());
                displayErrorWithReset(message);
                return false;
            }
            serialNumberValue = tempValue;
            lastCheckedSerialNumber = enteredSerialNumber;
            return true;
        } catch (BusinessException businessException) {
          displayError(UIExceptionFactory.buildError(businessException).getLocalizedMessage());
        } catch (Throwable t) {
            displayError(UIErrorKey.DEFAULT_FATAL_DIALOG_MESSAGE);
        }
        lastCheckedSerialNumber = StringConstants.EMPTY;
        serialNumberValue = null;
        clear();
        return false;
    }


/*********************************************************************************************
 * Add Remove Table Editor Listeners
 *********************************************************************************************/

    public void addTableEditorListener(SimTableEditorListener listener) {
        eventAdaptor.addTableEditorListener(listener);
    }

    public void removeTableEditorListener(SimTableEditorListener listener) {
        eventAdaptor.removeTableEditorListener(listener);
    }

    /*********************************************************************************************
 * Error State
 *********************************************************************************************/
    protected void setErrorState(boolean errorState) {
        isErrorState = errorState;
    }

    protected boolean isErrorState() {
        return isErrorState;
    }
```

```
/*****************************************************************************************
 * Determines if keystroke is invalid for input into the field.
 *****************************************************************************************/
    public boolean isInvalidKeystroke(KeyEvent event) {
        return false;
    }

    /*****************************************************************************************
 * Implement the focus listener methods to call do value modified when focus is lost.
 *****************************************************************************************/
    public void focusGained(FocusEvent event) {
    }

    public void focusLost(FocusEvent event) {
        if (event.isTemporary()) {
            return;
        }
        if (event != lastFocusEvent) {
            lastFocusEvent = event;
            if (checkValue()) {
                eventAdaptor.fireTypeEditorEvent(true);
                return;
            }
            reactivateEditing(event.getOppositeComponent());
        }
    }


/*****************************************************************************************
 * Display Error Methods
 *****************************************************************************************/

    protected void displayErrorWithReset(String message) {
        if (displayer != null) {
            setText(displayer.getDisplayText(serialNumberValue));
        }
        setErrorState(true);
        displayError(message);
    }

    protected void displayError(String message) {
        RErrorDialog dialog = new RErrorDialog(Application.getFrame());
        dialog.setTitle("Table Editor Error");
        dialog.setMessage(message);
        dialog.activate();
    }
}
```

Example: *CustomUINCreateModel*

```
package oracle.retail.sim.shared.swing.custom;

import java.util.List;
import oracle.retail.sim.closed.application.RepositoryManager;
import oracle.retail.sim.closed.custom.CustomSerialNumber;
import oracle.retail.sim.closed.custom.CustomUINEJBServices;
import oracle.retail.sim.closed.uin.SerialNumberValue;
import oracle.retail.sim.closed.util.CollectionUtil;
import oracle.retail.sim.shared.swing.core.SimScreenModel;

/*****************************************************************************************
 * Model for the Create UIN entry screen.
```

```
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 ********************************************************************************************/

public class CustomUINCreateModel extends SimScreenModel {

    private String CUSTOM_UIN_SERVICES_KEY = "CUSTOM_UIN_SERVICES_KEY";

    public void saveSerialNumbers(List<CustomUINCreateWrapper> wrappers) throws Exception {
        List<CustomSerialNumber> serialNumbers = CollectionUtil.newArrayList();
        for (CustomUINCreateWrapper wrapper : wrappers) {
            for (SerialNumberValue serialNumberValue : wrapper.getSerialNumbers()) {
                CustomSerialNumber serialNumber = new CustomSerialNumber();
                serialNumber.setId(serialNumberValue.getUINDetailId());
                serialNumber.setStatus(serialNumberValue.getStatus());
                serialNumbers.add(serialNumber);
            }
        }
        getCustomUINService().moveSerialNumbersToInStock(serialNumbers);
    }

    private CustomUINEJBServices getCustomUINService() {
        CustomUINEJBServices services = (CustomUINEJBServices)
                        RepositoryManager.getStateObject(CUSTOM_UIN_SERVICES_KEY);
        if (services == null) {
            services = new CustomUINEJBServices();
            RepositoryManager.addStateObject(CUSTOM_UIN_SERVICES_KEY, services);
        }
        return services;
    }
}
```

Example: *CustomUINCreatePanel*

```
package oracle.retail.sim.shared.swing.custom;

import java.util.Collections;
import java.util.List;
import java.util.Set;
import oracle.retail.sim.closed.item.StockItem;
import oracle.retail.sim.closed.swing.displayer.AttributeDisplayer;
import oracle.retail.sim.closed.swing.displayer.IntegerDisplayer;
import oracle.retail.sim.closed.swing.event.RActionEvent;
import oracle.retail.sim.closed.swing.event.REventListener;
import oracle.retail.sim.closed.swing.frame.ScreenPanel;
import oracle.retail.sim.closed.swing.table.SimTable;
import oracle.retail.sim.closed.swing.table.SimTableAttribute;
import oracle.retail.sim.closed.swing.table.SimTableDefinition;
import oracle.retail.sim.closed.swing.table.SimTableEditorEvent;
import oracle.retail.sim.closed.swing.table.SimTableEditorListener;
import oracle.retail.sim.closed.swing.table.SimTablePane;
import oracle.retail.sim.closed.swing.tableeditor.PopupTableEditorListener;
import oracle.retail.sim.closed.util.CollectionUtil;
import oracle.retail.sim.shared.swing.uom.StockItemTableEditor;

/********************************************************************************************
 * UIN Create Panel - Contains the visual elements of the UIN Create screen.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
```

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

public class CustomUINCreatePanel extends ScreenPanel implements REventListener {
    private static final long serialVersionUID = 7667002106326748099L;

    private CustomUINCreateModel model = new CustomUINCreateModel();

    private StockItemTableEditor stockItemTableEditor = new StockItemTableEditor();

    private SimTable stockItemTable = new SimTable(new CustomCreateUINTableDefinition());
    private SimTablePane stockItemPane = new SimTablePane(stockItemTable);

    /*********************************************************************************************
 * Initialize Panel
 ********************************************************************************************/

    public CustomUINCreatePanel() {
        initializePanel();
        layoutPanel();
    }

    private void initializePanel() {
        stockItemTableEditor.addTableEditorListener(buildStockItemListener());
        stockItemTable.setColumnSize("serialNumberCount", SimTable.LABEL_WIDTH);
    }

    private void layoutPanel() {
        setContentPane(stockItemPane);
    }

    public boolean isStartable() {
        return true;
    }

    public void start() throws Throwable {
        stockItemTable.addRow(new CustomUINCreateWrapper());
    }

    public void stop() {
    }

    public void performActionEvent(RActionEvent event) {
    }

    /*********************************************************************************************
 * Handle Add Item Action
 ********************************************************************************************/

    public void doAddItem() {
        List<CustomUINCreateWrapper> wrappers = stockItemTable.getAllRowData();
        for (CustomUINCreateWrapper wrapper : wrappers) {
            if (wrapper.getStockItem() == null) {
                return;
            }
        }
        stockItemTable.addRow(new CustomUINCreateWrapper());
    }

    /*********************************************************************************************
 * Handle Delete Item Action
 ********************************************************************************************/

    public void doDeleteItem() {
```

```
        List<CustomUINCreateWrapper> wrappers = stockItemTable.getAllSelectedRowData();
        for (CustomUINCreateWrapper wrapper : wrappers) {
            stockItemTable.removeRow(wrapper);
        }
    }

    /**********************************************************************************************
 * Handle Done Action
 **********************************************************************************************/

    public void doHandleDone() throws Exception {
        List<CustomUINCreateWrapper> wrappers = stockItemTable.getAllRowData();
        model.saveSerialNumbers(wrappers);
    }

    /**********************************************************************************************
 * Stock Item Listener - Accessed when stock item is added to table of items.
 **********************************************************************************************/

    private SimTableEditorListener buildStockItemListener() {
        return new SimTableEditorListener() {
            public void performTableEditorEvent(SimTableEditorEvent event) {
                Set<StockItem> stockItems = CollectionUtil.newHashSet();
                List<CustomUINCreateWrapper> wrappers = stockItemTable.getAllSelectedRowData();
                for (CustomUINCreateWrapper wrapper : wrappers) {
                    if (stockItems.contains(wrapper.getStockItem())) {
                        displayError("Duplicate item has been added.");
                        stockItemTable.removeRow(wrapper);
                        return;
                    }
                    stockItems.add(wrapper.getStockItem());
                }
            }
        };
    }


/**********************************************************************************************
 * Table Definition
 **********************************************************************************************/

    private class CustomCreateUINTableDefinition extends SimTableDefinition {

        public Class getDataClass() {
            return CustomUINCreateWrapper.class;
        }

        public List<String> getOverrideEditableAttributes() {
            return Collections.singletonList(CustomUINCreateWrapper.UIN_COUNT_PROPERTY);
        }

        public List<SimTableAttribute> getAttributes() {
            List<SimTableAttribute> attributes = CollectionUtil.newArrayList(3);
            attributes.add(new SimTableAttribute("Item",
                    CustomUINCreateWrapper.STOCK_ITEM_PROPERTY, new AttributeDisplayer("id"),
                    stockItemTableEditor));
            attributes.add(new SimTableAttribute("Item Description",
                    CustomUINCreateWrapper.DESCRIPTION_PROPERTY));
            attributes.add(new SimTableAttribute("UIN Qty",
                    CustomUINCreateWrapper.UIN_COUNT_PROPERTY, new IntegerDisplayer(),
                    new CustomUINCountTableEditor(new UINPopupListener())));
            return attributes;
        }
```

```
    }


/*********************************************************************************************
 * UIN POPUP LISTENER - Pops up when UIN column is double-clicked.
 *********************************************************************************************/

    private class UINPopupListener implements PopupTableEditorListener {
        public void popupDialog(Object lineItem) {
            CustomUINCreateDialog dialog = new CustomUINCreateDialog();
            dialog.setWrapper((CustomUINCreateWrapper) lineItem);
            dialog.setVisible(true);
            stockItemTable.setRows(stockItemTable.getAllRowData());
        }
    }
}
```

Example: *CustomUINCreateScreen*

```
package oracle.retail.sim.shared.swing.custom;

import oracle.retail.sim.closed.application.NavigationEvent;
import oracle.retail.sim.closed.swing.sim.SimScreen;
import oracle.retail.sim.shared.swing.core.SimNavigation;

/*********************************************************************************************
 * UIN Create Screen - This screens provides the ability to create and insert new UINs into
 * SIM without validation or generating matching transactions. This should be used exclusively
 * to dataseed UINs into the application.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 *********************************************************************************************/

public class CustomUINCreateScreen extends SimScreen {
    private static final long serialVersionUID = 487712289865375658L;

    private CustomUINCreatePanel panel = new CustomUINCreatePanel();

    public CustomUINCreateScreen() {
        add(panel);
    }


/*********************************************************************************************
 * Screen Methods
 *********************************************************************************************/

    public String getScreenName() {
        return "UIN Create";
    }

    public boolean isStartable() {
        return panel.isStartable();
    }

    public void start() throws Throwable {
        showMenu();
        panel.start();
    }

    public void stop() {
```

```
        panel.stop();
    }


/**********************************************************************************************
 * Handle Navigation Events
 **********************************************************************************************/

    public void navigationEvent(NavigationEvent event) {
        String command = event.getCommand();
        try {
            if (command.equals(SimNavigation.ADD_ITEM)) {
                panel.doAddItem();
            } else if (command.equals(SimNavigation.DELETE_ITEM)) {
                panel.doDeleteItem();
            } else if (command.equals(SimNavigation.DONE)) {
                panel.doHandleDone();
            }
        } catch (Throwable exception) {
            displayException(panel, event, exception);
        }
    }
}
```

Example: *CustomUINCreateWrapper*

```
package oracle.retail.sim.shared.swing.custom;

import java.util.List;
import oracle.retail.sim.closed.common.BusinessException;
import oracle.retail.sim.closed.common.locale.StringConstants;
import oracle.retail.sim.closed.item.StockItem;
import oracle.retail.sim.closed.uin.SerialNumberValue;
import oracle.retail.sim.closed.util.CollectionUtil;
import oracle.retail.sim.closed.util.RkConfigManager;

/**********************************************************************************************
 * Wrapper for single entry row in the serial number table.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 **********************************************************************************************/

public class CustomUINCreateWrapper {

    public static final String STOCK_ITEM_PROPERTY = "stockItem";
    public static final String DESCRIPTION_PROPERTY = "description";
    public static final String UIN_COUNT_PROPERTY = "serialNumberCount";

    private StockItem stockItem;
    private List<SerialNumberValue> serialNumbers = CollectionUtil.newArrayList();

    public StockItem getStockItem() {
        return stockItem;
    }

    public String getDescription() {
        if (stockItem == null) {
            return StringConstants.EMPTY;
        }
        if (RkConfigManager.isItemShortDescription()) {
            return stockItem.getShortDescription();
```

```
        }
        return stockItem.getLongDescription();
    }

    public void setStockItem(StockItem stockItem) throws BusinessException {
        if (stockItem != null && !stockItem.isSerialNumberRequired()) {
            throw new BusinessException("Item does not support serial numbers.");
        }
        this.stockItem = stockItem;
    }

    public Integer getSerialNumberCount() {
        return serialNumbers.size();
    }

    public List<SerialNumberValue> getSerialNumbers() {
        return serialNumbers;
    }

    public void setSerialNumbers(List<SerialNumberValue> serialNumbers) {
        if (serialNumbers != null) {
            this.serialNumbers = serialNumbers;
        }
    }

    public boolean isPropertyModifiable(String property) {
        if (property.equals(STOCK_ITEM_PROPERTY)) {
            return stockItem == null;
        }
        if (property.equals(UIN_COUNT_PROPERTY)) {
            return true;
        }
        return false;
    }
}
```

Example: *CustomUINDataBean*

```
package oracle.retail.sim.shared.dataaccess.databean.generated;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import java.util.ArrayList;
import java.util.List;
import oracle.retail.sim.closed.dataaccess.FullDataBean;

/**
 *
 * This class is an object representation of the database table UIN_DETAIL<BR>
 * Followings are the column of the table: <BR>
 *     ID(PK) -- NUMERIC(12)<BR>
 *     STORE_ID -- NUMERIC(10)<BR>
 *     STATUS -- NUMERIC(2)<BR>
 *
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
public class CustomUinDataBean extends FullDataBean<CustomUinDataBean> {
    private static String getSelectSqlText() {
        return "select ID, STORE_ID, STATUS from UIN_DETAIL ";
    }

    private static String getInsertSqlText() {
```

```
    return "insert into UIN_DETAIL (ID, STORE_ID, STATUS) values (?, ?, ?)";
}

private static String getUpdateSqlText() {
    return "update UIN_DETAIL set STORE_ID = ?, STATUS = ? ";
}

private static String getDeleteSqlText() {
    return "delete from UIN_DETAIL ";
}

public static final String TABLE_NAME = "UIN_DETAIL";
public static final String COL_ID = "UIN_DETAIL.ID";
public static final String COL_STORE_ID = "UIN_DETAIL.STORE_ID";
public static final String COL_STATUS = "UIN_DETAIL.STATUS";

public static final int TYP_ID = Types.NUMERIC;
public static final int TYP_STORE_ID = Types.NUMERIC;
public static final int TYP_STATUS = Types.NUMERIC;

public static final String SELECT_SQL = getSelectSqlText();
public static final String INSERT_SQL = getInsertSqlText();
public static final String UPDATE_SQL = getUpdateSqlText();
public static final String DELETE_SQL = getDeleteSqlText();

private Long id;
private Long storeId;
private Long status;

public CustomUinDataBean() {
}

public String getSelectSql() {
    return SELECT_SQL;
}

public String getInsertSql() {
    return INSERT_SQL;
}

public String getUpdateSql() {
    return UPDATE_SQL;
}

public String getDeleteSql() {
    return DELETE_SQL;
}

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public void setId(long id) {
    this.id = id;
}

public Long getStoreId() {
    return storeId;
}
```

```
    public void setStoreId(Long storeId) {
        this.storeId = storeId;
    }

    public void setStoreId(long storeId) {
        this.storeId = storeId;
    }

    public Long getStatus() {
        return status;
    }

    public void setStatus(Long status) {
        this.status = status;
    }

    public void setStatus(long status) {
        this.status = status;
    }

    public CustomUinDataBean read(ResultSet resultSet) throws SQLException {
        CustomUinDataBean bean = new CustomUinDataBean();
        bean.id = getLongFromResultSet(resultSet, "ID");
        bean.storeId = getLongFromResultSet(resultSet, "STORE_ID");
        bean.status = getLongFromResultSet(resultSet, "STATUS");
        return bean;
    }

    public List<Object> toList(boolean includePrimaryKey) {
        List<Object> list = new ArrayList<Object>();
        if (includePrimaryKey) {
          addToList(list, id, TYP_ID);
        }
        addToList(list, storeId, TYP_STORE_ID);
        addToList(list, status, TYP_STATUS);
        return list;
    }
}
```

Example: *CustomUINEJBServices*

```
package oracle.retail.sim.closed.custom;

import oracle.retail.sim.closed.common.BusinessException;
import oracle.retail.sim.closed.common.DowntimeException;
import oracle.retail.sim.closed.common.SimServerException;
import oracle.retail.sim.closed.ejb.CustomUINInterface;
import oracle.retail.sim.closed.logging.LogNames;
import oracle.retail.sim.closed.logging.LogService;
import oracle.retail.sim.closed.util.CompressedObject;
import oracle.retail.sim.closed.util.EJBServicesManager;
import oracle.retail.sim.closed.util.SimObjectUtils;
import oracle.retail.sim.closed.util.UniversalContext;

/**
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
public class CustomUINEJBServices extends CustomUINServices {
    private CustomUINInterface lookup() throws Exception {
        try {
            return (CustomUINInterface) EJBServicesManager.cachedLookup("CustomUINBean");
        } catch (Throwable t) {
```

```
            throw new DowntimeException(
"An error occurred accessing CustomUINServices. Please contact your system administrator.", t);
        }
    }

    private void removeCache() {
        EJBServicesManager.removeCache("CustomUINBean");
    }

    public void moveSerialNumbersToInStock(
                    java.util.List<oracle.retail.sim.closed.custom.CustomSerialNumber> arg0)
                    throws Exception {
        Throwable causeOfException = null;
        int maxAttempts = EJBServicesManager.getMaxConnectAttempts();
        CompressedObject compressedSimSession =
                      new CompressedObject(UniversalContext.getSession());
        CompressedObject compressed0 = new CompressedObject(arg0);
        if (LogService.isDebugEnabled(LogNames.SERIALIZED_OBJECT_SIZES)) {
            LogService.debug(LogNames.SERIALIZED_OBJECT_SIZES,
"CustomUINBean.moveSerialNumbersToInStock(compressed0, compressedSimSession) (remote call) is
sending " + SimObjectUtils.calculateByteSize(compressed0, compressedSimSession) + " bytes in
serialized object(s) for the remote call.");
        }
        for (int i = 0; i < maxAttempts; i++) {
            CustomUINInterface bean = lookup();
            try {
                long startTime = System.currentTimeMillis();
                CompressedObject compressedReturnVal =
                       bean.moveSerialNumbersToInStock(compressed0, compressedSimSession);
                long endTime = System.currentTimeMillis();
                if (LogService.isDebugEnabled(LogNames.SERVICE_TIMINGS)) {
                    LogService.debug(LogNames.SERVICE_TIMINGS,
"CustomUINBean.moveSerialNumbersToInStock(compressed0, compressedSimSession) (remote call) took "
+ (endTime - startTime) + "ms to complete");
                }
                if (LogService.isDebugEnabled(LogNames.SERIALIZED_OBJECT_SIZES)) {
                    LogService.debug(LogNames.SERIALIZED_OBJECT_SIZES,
"CustomUINBean.moveSerialNumbersToInStock(compressed0, compressedSimSession) (remote call)
received " + SimObjectUtils.calculateByteSize(compressedReturnVal) + " bytes in the returned
serialized object.");
                }
                return;
            } catch (SimServerException sse) {
                throw sse;
            } catch (BusinessException be) {
                throw be;
            } catch (Throwable t) {
                causeOfException = t;
                LogService.error(this, "Unexpected error in EJB connection attempt: " + i, t);
                removeCache();
            }
        }
        throw new DowntimeException("An error occurred accessing CustomUINServices. Please contact
your system administrator.", causeOfException);
    }

}
```

Example: *CustomUINInterface*

```
package oracle.retail.sim.closed.ejb;

import javax.ejb.Remote;
import oracle.retail.sim.closed.util.CompressedObject;

/**
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
@Remote
public interface CustomUINInterface {
    CompressedObject moveSerialNumbersToInStock(CompressedObject compressed0, CompressedObject
compressedSimSession) throws Exception;

}
```

Example: *CustomUINOracleDAO*

```
package oracle.retail.sim.shared.dataaccess.dao;

import java.util.List;
import oracle.retail.sim.closed.common.SimServerException;
import oracle.retail.sim.closed.custom.CustomSerialNumber;
import oracle.retail.sim.closed.dataaccess.BaseOracleDao;
import oracle.retail.sim.closed.dataaccess.BatchParametricStatement;
import oracle.retail.sim.closed.uin.UINStatus;
import oracle.retail.sim.closed.util.CollectionUtil;
import oracle.retail.sim.shared.dataaccess.databean.generated.UinDetailDataBean;

/*******************************************************************************************
 * This class contains the database layer code for inserting new serial numbers into
 * the database.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 *******************************************************************************************/
public class CustomUINOracleDao extends BaseOracleDao {

    public void saveSerialNumbers(Long storeId, List<CustomSerialNumber> serialNumbers)
                                                                    throws SimServerException {
        StringBuilder sql = new StringBuilder();
        sql.append(" UPDATE ").append(UinDetailDataBean.TABLE_NAME);
        sql.append("    SET ").append(UinDetailDataBean.COL_STATUS).append(" = ? ");
        sql.append("  WHERE ").append(UinDetailDataBean.COL_ID).append(" = ? ");
        sql.append("    AND ").append(UinDetailDataBean.COL_STORE_ID).append(" = ? ");
        sql.append("    AND ").append(UinDetailDataBean.COL_STATUS).append(" = ? ");

        BatchParametricStatement batchStatement = new BatchParametricStatement(sql.toString());
        List<Object> params = null;

        for (CustomSerialNumber serialNumber : serialNumbers) {
            params = CollectionUtil.newArrayList(2);
            params.add(UINStatus.IN_STOCK.getCode());
            params.add(serialNumber.getId());
            params.add(storeId);
            params.add(serialNumber.getStatus().getCode());

            batchStatement.addParams(params);
        }
        executeBatch(batchStatement);
    }
}
```

Example: *CustomUINServerServices*

```
package oracle.retail.sim.closed.custom;

import java.util.List;
import oracle.retail.sim.closed.util.UniversalContext;
import oracle.retail.sim.shared.dataaccess.dao.CustomUINOracleDao;

/**
 * Implements the Custom UIN services.
 * <p>
 * This is future work-in-progress code for future SIM 14.0 or later release.
 * <p>
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
public class CustomUINServerServices extends CustomUINServices {

    public void moveSerialNumbersToInStock(List<CustomSerialNumber> serialNumbers)
                                                                        throws Exception {
        CustomUINOracleDao dao = new CustomUINOracleDao();
        dao.saveSerialNumbers(UniversalContext.getStoreId(), serialNumbers);
    }
}
```

Example: *CustomUINServices*

```
package oracle.retail.sim.closed.custom;

import java.util.List;

/**
 * Defines the API for Custom UIN Services.
 *
 * Copyright 2004, 2011, Oracle. All rights reserved.
 */
public abstract class CustomUINServices {
    /**
     * This API will move all serial numbers include in the parameters from the current status
     * to IN_STOCK. This will only happen if the serial number is currently in the status
     * specified  in the parameter object. This will write a history record of the movement.
     * @param serialNumbers The serial numbers to update.
     */
    public abstract void moveSerialNumbersToInStock(List<CustomSerialNumber> serialNumbers) throws
Exception;
}
```