Retail passion. Innovative solutions.

# Retek® Customer Order Management™ 11.0.1

# Integration Guide

**Corporate Headquarters:**

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403
USA

888.61.RETEK (toll free US)
Switchboard:
+1 612 587 5000

Fax:
+1 612 587 5100

**European Headquarters:**

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46

Fax:
+44 (0)20 7563 46 10

# Customer Support

### Customer Support hours

Customer Support is available 7x24x365 via email, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance. Retek customers on active maintenance agreements may contact a global Customer Support representative in accordance with contract terms in one of the following ways.

| Contact Method | Contact Information |
|---|---|
| **E-mail** | support@retek.com |
| **Internet (ROCS)** | rocs.retek.com<br>Retek's secure client Web site to update and view issues |
| **Phone** | +1 612 587 5800 |

Toll free alternatives are also available in various regions of the world:

| | |
|---|---|
| Australia | +1 800 555 923 (AU-Telstra) or +1 800 000 562 (AU-Optus) |
| France | 0800 90 91 66 |
| Hong Kong | 800 96 4262 |
| Korea | 00 308 13 1342 |
| United Kingdom | 0800 917 2863 |
| United States | +1 800 61 RETEK or 800 617 3835 |
| **Mail** | Retek Customer Support<br>Retek on the Mall<br>950 Nicollet Mall<br>Minneapolis, MN 55403 |

**When contacting Customer Support, please provide:**

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

# Contents

# Chapter 1 – Introduction

This integration guide serves as a Retek Customer Order Management (RCOM) reference to explain the system's application programming interfaces (API)s. Within RCOM, various components, organized by functional area, have their own APIs.

## Overview

RCOM has been designed to have more flexibility to integrate into the enterprise retail environment. RCOM leverages the retail enterprise by exposing merchandise, pricing, promotion, customer, and supply chain information. This integration from RCOM into the enterprise leads to improved fill-rates for customer demand, quicker shipments of customer orders, and increased inventory turns.

By offering a single and complete view of the customer, RCOM provides retailers with a consistent method of creating, managing, and viewing customer interactions across the enterprise for all of a client's channels and brands.

The application is designed and built as a comprehensive business-to-consumer enterprise order management solution and provides integrated business processes into the retail environment, such as into merchandising and warehouse management applications.

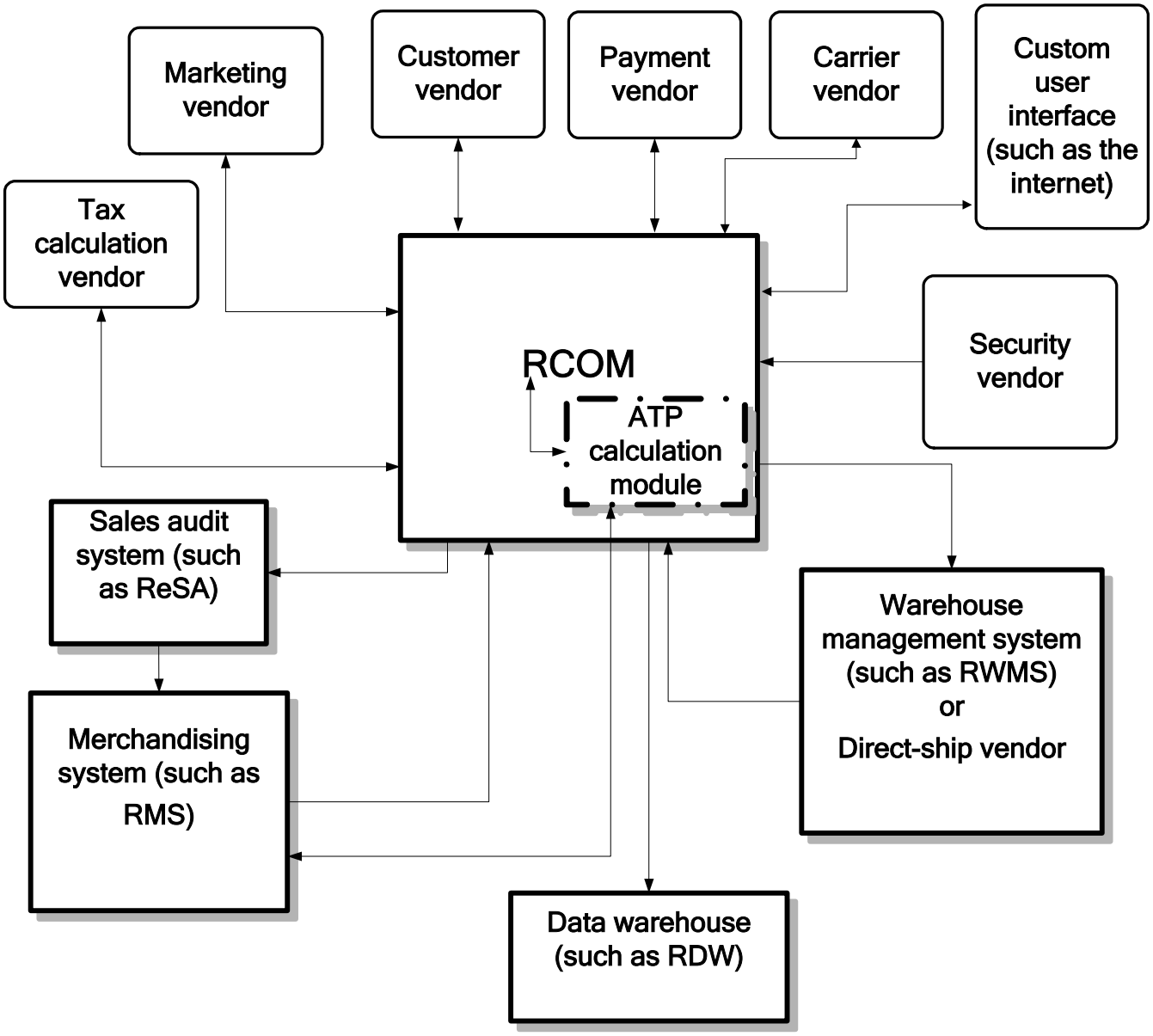
## Who this guide is written for

Anyone who has an interest in better understanding the inner workings of the RCOM system can find valuable information in this guide. There are two audiences in general for whom this guide is written:

- Integrators and implementation staff who have the overall responsibility for implementing RCOM into their enterprise.
- Business analysts who are looking for information about processes and interfaces to validate the support for business scenarios within RCOM and other systems across the enterprise (a merchandising system such as RMS, a warehouse management system, and so on).

# RCOM's integration points into the retail enterprise

The high-level diagram below shows the overall direction of the data among systems and products across the enterprise. For a detailed description of a similar diagram, see "Chapter 4 – Interface process flows".




**RCOM-related dataflow across the enterprise**

# Technical architecture overview

RCOM's technical architecture is an object-orientated J2EE-compliant platform that provides the extensibility of customer order management into the retailing enterprise.

RCOM's architecture is built upon a layering model that allows for a given layer's responsibilities to be encapsulated. In other words, one layer does not need knowledge of how a different layer accomplishes its tasks. Rather, a layer merely needs to understand the application-programming interface (API) that is exposed by another layer. This approach allows for more facile development because an alternative implementation of a layer can be substituted without impacting other layers.

To allow integration with systems other than RCOM's own user interface, and to allow for the future possibility of different user interfaces, RCOM contains all business models and logic in a set of business objects, called 'components'. These components expose a set of well-known business APIs as Java interfaces, and they hide implementation details. The components have been designed to provide common business logic across multiple banners and order capture devices.

For a more detailed description of RCOM's technical architecture, see "Chapter 2 – Technical architecture".

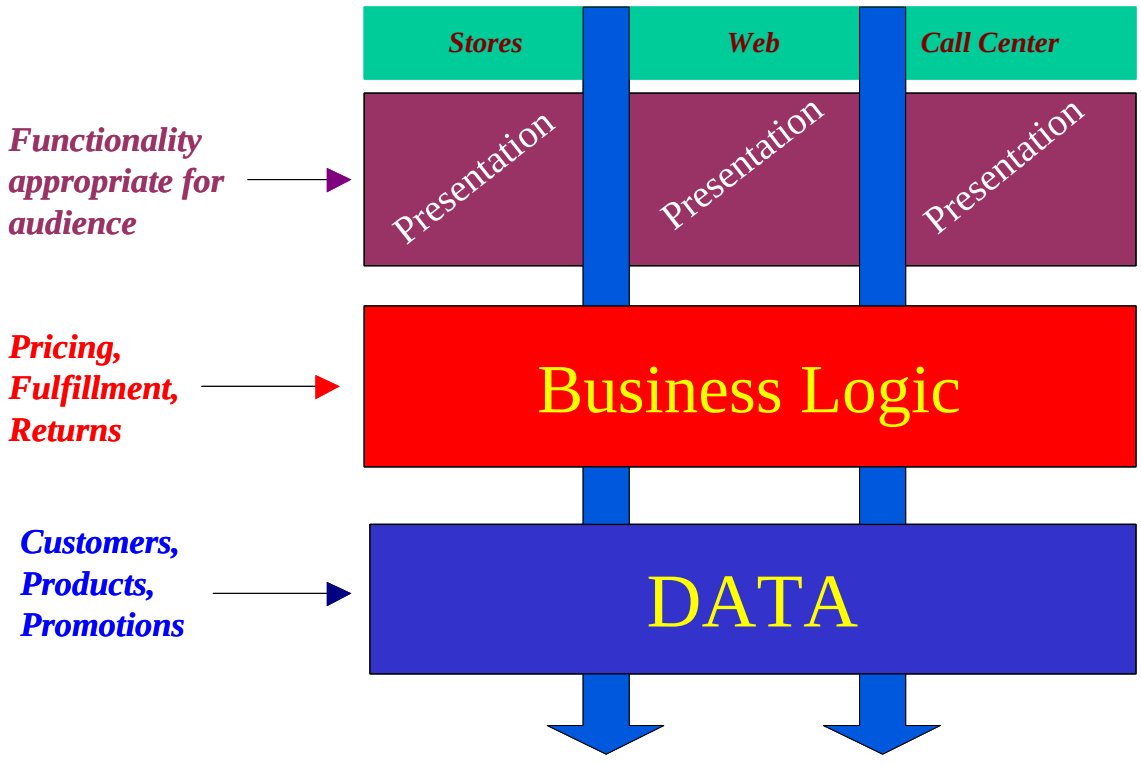
# The business advantages of the layered approach

Because RCOM acts as the order 'engine' that validates, routes, and updates customer orders throughout the order lifecycle, RCOM must be extensible across multiple channels and banners. A layered approach provides this capability.

The business advantages of a layered approach include the following:

- The elimination of duplicate order management business logic across banners
- Reduced operational costs. RCOM provides consistent order management within a variety of order capture devices across channels (such as, for example, store, kiosk, POS, the web, mail order and/or call centers). This capability eliminates the cost of maintaining redundant business logic.
- Increased customer service with an extensible view of real-time ATP merchandise.
- Constant and real-time visibility into the customers' orders and specific order line status through the transaction lifecycle.
- Real-time access to customer and to customer history data across the enterprise.

The following diagram describes a business perspective of the layered approach.




**RCOM's architecture from a business perspective**

# The components and Javadoc

Javadoc is the tool from Sun Microsystems that generates API documentation in HTML format. The RCOM client receives Javadoc documentation generated from RCOM code as a separate documentation deliverable (along with the User Guide, Installation Guide, and so on). Click the HTML file named 'index' in the applicable Javadoc folder to open the Javadoc.

Javadoc can be used in conjunction with this Integration Guide, especially with "Chapter 5 – Component overviews and interface(s)". To better understand the method-level implementation that would be required to leverage a component within RCOM, see its section within Chapter 5 ('Banner and channel component' for example) and see its section within the Javadoc at its hyperlinked location. For example:

- com.retek.component.banner_channel

# Where you can find more information

- RCOM front-end documentation (for example, the RCOM User Guide)
- RCOM Operations Guide
- RCOM Installation Guide
- Retek Warehouse Management System (RWMS) product documentation
- Retek Merchandising System (RMS) product documentation
- Retek Integration Guide and other RIB-related documentation
- Applicable third-party documentation (such as for Vertex, and so on)

# Chapter 2 – Technical architecture

This chapter describes the overall software architecture for RCOM. The chapter provides a high-level discussion of the general structure of the system, including the various layers of Java code. From the content, integrators can learn both about the pieces of the system and where to enter the system. This information is valuable in the following scenarios, among others:

- Interfacing with the system.
- Implementing the system for a different database.

For those who are less familiar with Java terminology, a description of RCOM-related Java terms and standards is provided for your reference at the end of this chapter.

## Overview

RCOM's robust distributed computing platform is a J2EE implementation that enables enhanced performance and allows for scalability. RCOM utilizes a Java platform because it offers the optimum solution to the challenges presented by the need for database independence.

An object oriented system, RCOM can be thought of as a system of animated objects collaborating to fulfill system requirements. Correctly assigning knowledge (data) and logic (process) responsibilities to classes is the fundamental challenge in creating a maintainable object-oriented system. To allow integration with systems other than RCOM's own user interface, and to allow for the future possibility of different user interfaces, RCOM contains all business models and logic in a set of business objects, called 'components'. These components expose a set of well-known business APIs as Java interfaces, and they hide implementation details.

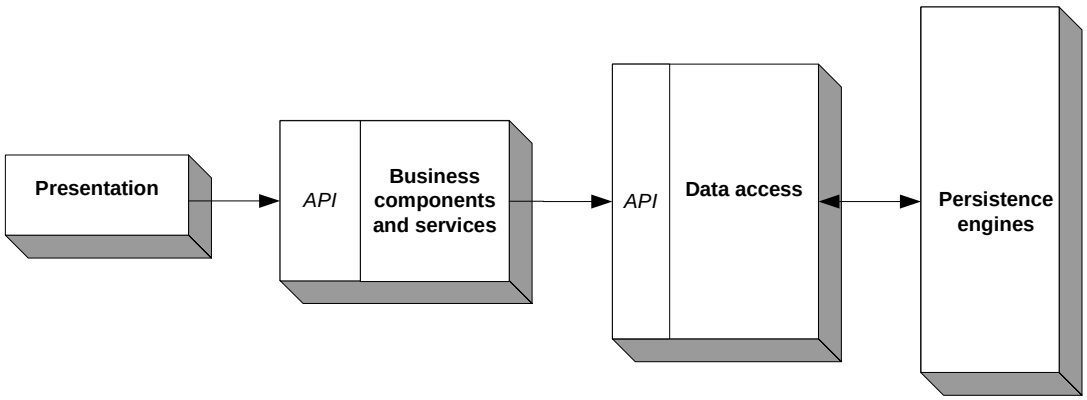
RCOM's architecture is built upon a layered model. This approach provides the advantage of functional encapsulation. That is, any given layer need not be concerned with the internal functional tasks of any other layer. Each must only be able to interpret the application programming interface (API) of any other layer.

The layered model of architecture offers the following advantages, among others:

- The functionality of each layer is provided through its public interface, also known as its API.
- Interaction among layers occurs only through their defined APIs.
- With the exception of the presentation layer, the layers in the model are implemented in independent locations. That is, they are distributed.
- For client layers, the complexities of remote access can be managed.
- Transactional boundaries are defined. For example, when actions must be accomplished under transactional control, the components are designed to participate in an active transaction or to initiate a transaction if none is active.
- Security authentication functionality.
- Java database connectivity (JDBC) in the DAO layer, minimizing the number of interface points that need to be maintained.

# A high-level view of the layered model

The following diagram, together with the explanations that follow, offer a high-level conceptual view of the layers and their responsibilities within the architecture.




**Conceptual view of the layered model**

## Presentation layer

This layer handles the presentation of the application, including its user interface. To facilitate the demands and complexity of order line entries, the RCOM front end facilitates robust client-side processing. The RCOM interface was developed using Swing, which is a toolkit for creating rich graphical user interfaces (GUIs) in Java applications. RCOM thus provides a Swing-based fat client (with the possibility of other interfaces).

## Business components and services layer

This layer is comprised of business components, which represent a logical model of business entities and processes, not geared to any particular presentation and context.

In other words, the data entering the system from its own user interface, a web interface, a third party system's integration into the system, and so on would all be viewed and handled by a component in the same way. A component is indifferent to the source of the data, even if it stems from another component. 'Something' is asking a question of the component, and it returns the answer.

Related data and functionality is encapsulated in the business component; thus, a component typically represents a business concept, such as an order, a customer or a payment. Each component exposes the data and services related to its business concept to the rest of the system through a well-defined interface (an API), and is solely responsible for maintaining its runtime and persisted state.

The API layer insulates a component's clients from its internal implementation details, thus creating well-defined boundaries for refactoring within a component. The API consists of one or more Java interfaces. The business delegate layer includes exactly one implementation (that is, one Java class that implements) for each of these interfaces. Note that individual classes may implement multiple interfaces, and interfaces may have inheritance relationships with one another. For example, there may be a batch-use interface for a component, and a GUI-use interface for the same component, with one class implementing both interfaces.

The business object combines data about the business concept with the logic that processes that data. Business logic includes the following:

- Simple validation (can a field be null or negative?)
- Validation that requires interaction with other components (is this customer's credit card valid?)
- Knowledge of workflow (an order line cannot be fulfilled until the monogramming of items is complete)

Assigning these responsibilities well creates a robust, flexible and maintainable system. Business component objects implement business rules. The interaction of components enables the system to accomplish tasks and internal workflows. A common business object infrastructure allows for the components to be utilized again and again within the enterprise. This layer is used by other layers to access data and implement business processes.

Services within RCOM are synonymous with Enterprise Java Beans (EJB). EJBs within each component have the responsibility for coordinating transaction processing to ensure the atomicity, consistency, isolation, and durability (ACID) properties of the functional request. For a definition of ACID, see the “RCOM-related Java terms and standards” section in this chapter.

## Data access layer

In order to support multiple databases, and the future possibility of using object/relational mapping, RCOM’s business objects and services do not make direct database calls, but instead go through data access objects, which are hidden behind Java interfaces. This layer facilitates the application’s interaction with persistence within the application, most notably, but not limited to, database persistence. The reason for the data access layer is to render the job of persistence ‘abstract’ (not tied to a specific type of database such as Oracle, DB2, and so on). Database independence is achieved because database code does not permeate the actual database that the system uses.

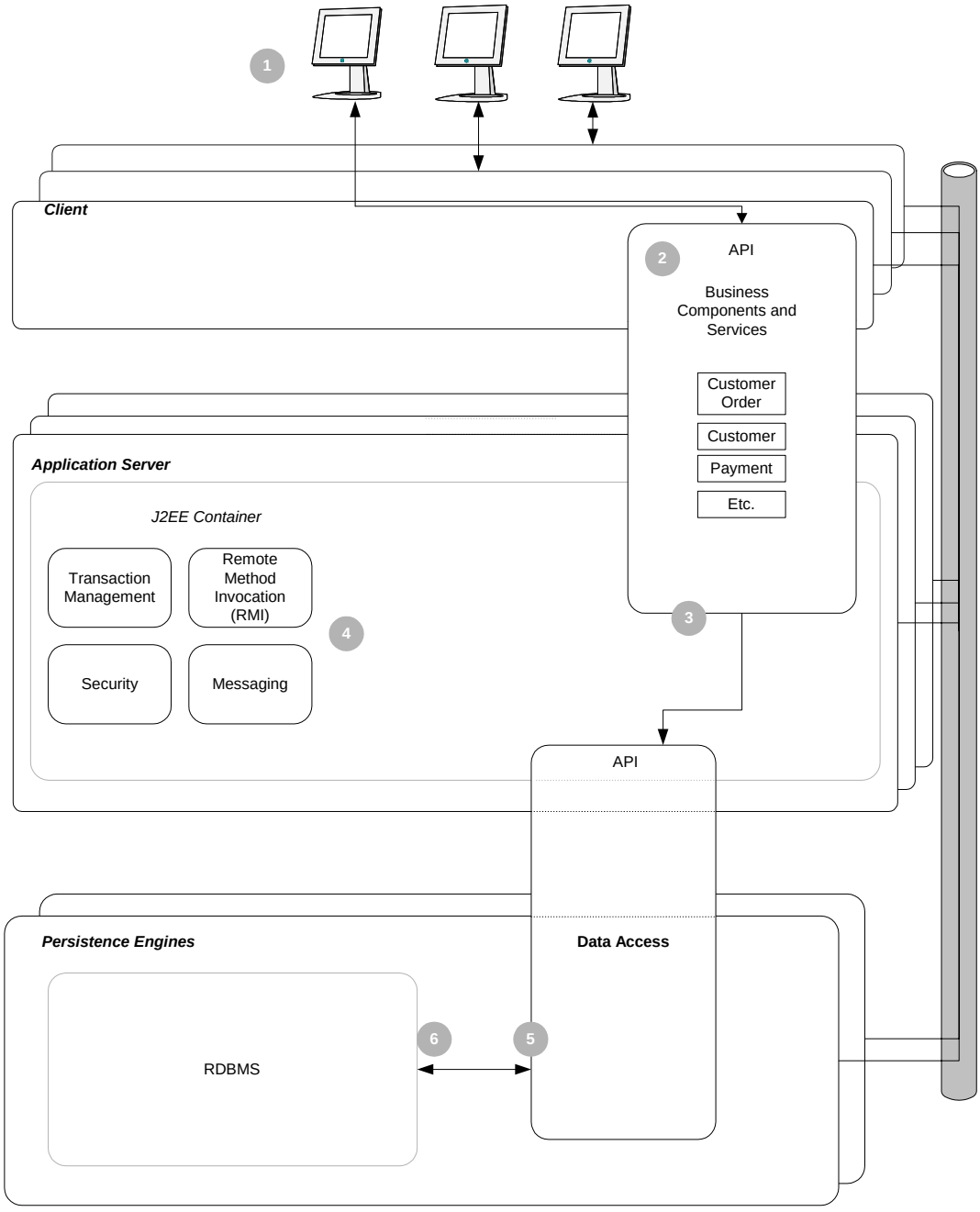
No business logic resides in this layer.

## Database

The database is the application’s storage platform, containing the physical data (user and system) used throughout the application. The database is only intended to deal with the storage and retrieval of information and is not involved in the manipulation or in the delivery of the data. The database responds to queries; it does not initiate them.

# A detailed distributed view of the layered architecture

The following diagram offers a detailed distributed view of the architecture and some of the hardware associated to it. Explanations of each number follow the diagram. Note that the numbers do not reflect the system's order of operation but are provided to facilitate the discussion of the model.




**A detailed distributed view of the layered architecture**

1 The presentation layer runs primarily on the client machines. The number of client machines that are utilized can be scaled through the addition of hardware.

2 Each distributed entity is implemented through a number of interfaces and classes which are deployed across separate Java virtual machines. These layers are accessed by remote clients exclusively through their public APIs. The layers are 'location-independent', meaning that they can be accessed from anywhere in the system. The services in the layer use Enterprise Java Beans (EJB) as session beans (stateless session beans) and EJB design patterns. They appear as if they were deployed on the client machine, regardless of the actual physical machine on which they might reside. In addition, note that these layers can be deployed across multiple application servers, providing potential performance and reliability advantages. All transaction-related activities occur on this layer (for example, commits, rollbacks, and so on).

3 The business components and services layer is responsible for using the services provided by the J2EE application container relative to the unit of work being performed. These responsibilities include the following:

   - The setting and maintaining of transactional boundaries. These components typically define the start and end points for a transaction. The container, however, manages the transaction state through the invocation thread across potentially distributed components. Any applicable component that participates in the resulting execution thread may then choose to dynamically participate.
   - The setting and maintaining of access privileges. These components can define security roles required to invoke specific business logic. Security roles are relative to the user and propagated through the execution thread by the container. As a result, the container enforces security requirements at each invocation in the thread of execution.

4 The J2EE containers act as the interface between a component and the low-level platform-specific functionality that supports the component. Before a Web, enterprise bean, or application client component can be executed, it must be assembled into a J2EE application and deployed into its container.

   Hardware can be added enabling scalability. Additional application servers provide potential performance and reliability benefits. Through the addition of hardware, computational power is enhanced, increasing throughput, and single points of failure can be rectified.

5 The data access layer provides a means to access the varied information systems without affecting the rest of the system. Multiple persistence engines within this layer allow for objects to be flexibly stored, read, and maintained. Each engine may use a different storage format and provide a different access mechanism and/or paradigm.

   This layer enables programming flexibility and bolsters the rapid development of business logic. The system can be configured to offer persistence engine access at runtime. Furthermore, by deploying data access components close to the actual persistence engine it employs, performance can be enhanced. Transparent to the rest of the system, including dependent RCOM functionality, this flexible deployment scheme provides a way to perform on-site tuning and maintenance.

6 The data access layer communicates with the database using a Java Database Connectivity (JDBC) protocol.

# Advantages of the data access object (DAO) layer

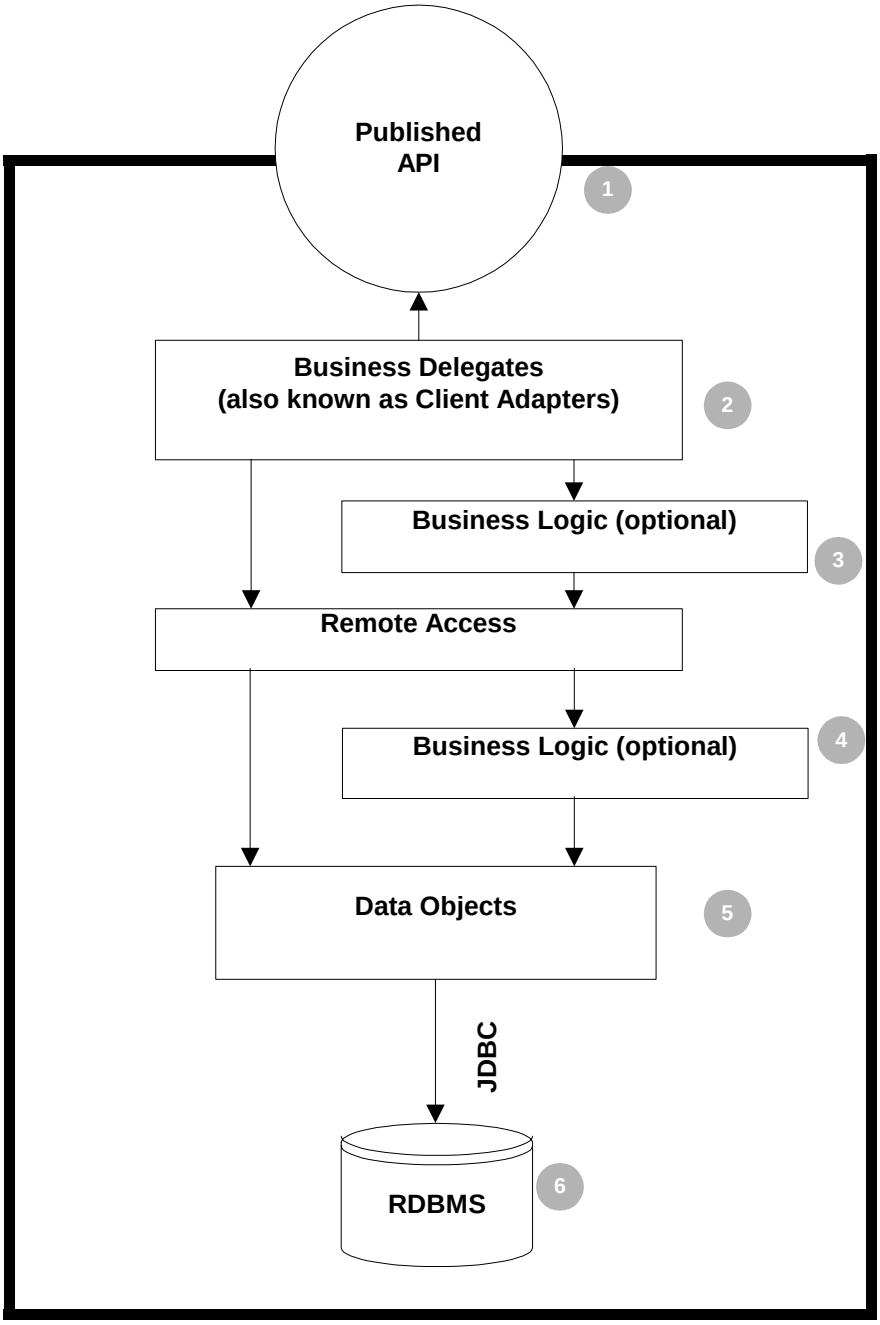
RCOM utilizes a DAO design pattern because it decouples the data access logic from the business logic. In other words, the data access layer can be easily altered or replaced and cause little or no impact to the business layer. The DAO layer abstracts the actual persistence mechanism that is being used to persist business objects. The DAO layer allows for changes to a different database or even to the use of regular files to persist the business objects. In those cases, only the DAO layer would need to be modified due to the change. The remainder of RCOM would continue operating unchanged.

The DAO is essentially responsible for creating data transfer objects (DTO)s (both from new and from persisted data). DTOs can be utilized for performance tuning. The component is responsible for maintaining its own state, both in memory and persistence. The responsibility for checking the validity of data is assigned to the business delegates, EJBs and the logic layer classes. As a result, the DAO does not validate values coming from or going into persistent storage.

Through the use of SQL expressions that can handle cross table joins in one SQL fetch, RCOM has the ability to generate 'complete' DTOs in the DAO layer. The DAO interface to the middle layer is thus kept simple, (only one interface is necessary), and SQL tuning and maintenance efforts are minimized. Note that the DAO model contains no validation logic. The calling RCOM component middle layer contains the validation logic.

# Component processing

The following diagram offers a closer look at the processing involved for a given component. Explanations of each number follow the diagram.




**A component (object) in the remote access layer**

1 The clean and minimal Java interfaces are oriented around business processes rather than lower-level object models. They are business entities and managers that present a conceptual model.

2 Ordinary Java classes handle client-side validation logic. These classes map between the API and the remote access object models. They also mask the complexities of the EJBs.

3 Remote access is handled by session EJBs and Data Transfer Objects (DTO). These set transaction and security boundaries, and they determine whether to dispatch to business logic or to go directly to DAOs. To ensure the atomicity, consistency, isolation, and durability (ACID) properties of state transitions, RCOM implements some business logic using a state machine (a workflow engine). This workflow engine manages workflow concerns of orders, payments, and so on as they go from state to state.

4 Ordinary Java classes are responsible for the business logic and can facilitate plug-in customization. These classes use the strategy pattern and isolate individual business rules. If necessary, they may dispatch to other components.

5 DAOs contain no logic. Rather, they embody the low-level data model. They contain the persistent application state.

6 The JDBC relational database belongs to this single component. The data model is not published to other components. The database is only accessed through a component API, except in the case of low-level bulk operations such as replication, loading, and so on.

# Modularization for ‘pluggable’ implementation packages and configuration items

As part of RCOM 11.x development, the application architecture and deployment has been enhanced to support a flexible means for maintaining and deploying different application modules within the RCOM application. This section outlines the details for the technical approach within RCOM to modularize the code for ‘pluggable’ implementation packages and configuration items.

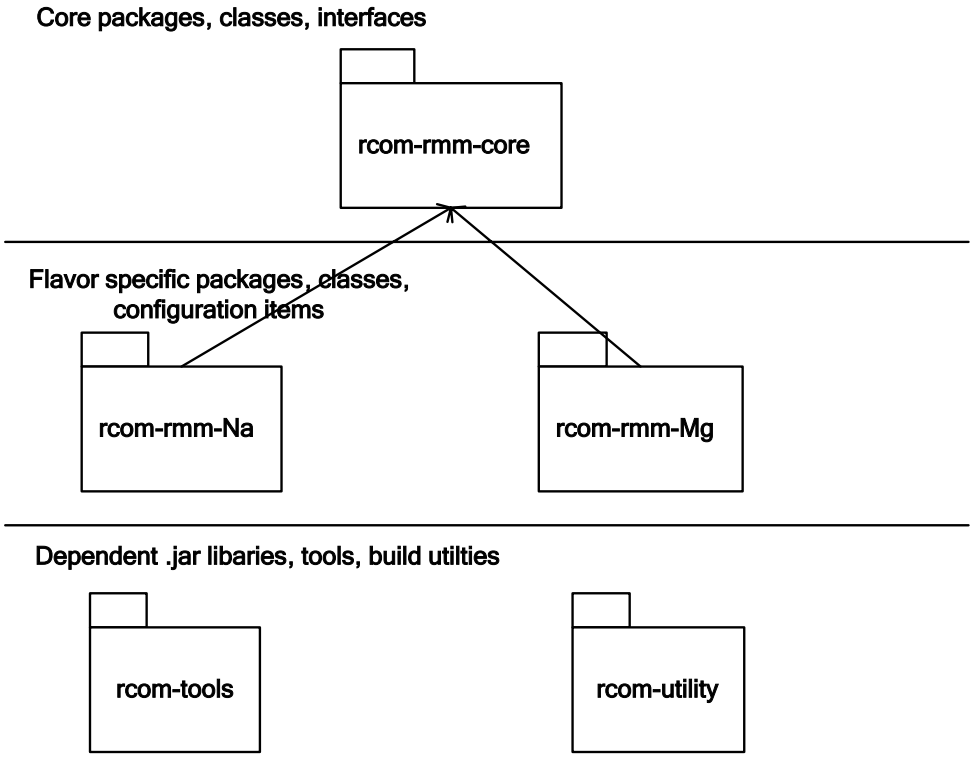
This solution for maintaining and deploying custom modules within the RCOM application server instance was built to satisfy the requirement for Retek software clients to integrate with custom versions of RMS/ATP and the RIB. The design approach within RCOM is referred to as the RCOM plug-in architecture.

This approach was accomplished within the inner-workings of the service layer of the RCOM application through the dynamic loading of implementation classes at runtime for certain application services. These version-specific services (referred to as ‘plug-in’ classes) are encapsulated by Java interfaces.

See the ‘Pluggable application services’ section in this chapter for more details on this pattern. Based on a specification within the deployment configuration, the appropriate ‘plug-in’ class is invoked from a custom or base RCOM package/module.

## RCOM modules

The following diagram depicts a high-level view of the basic composition/modularization of the code:




Different runtime configurations of RCOM are referred to as ‘flavors’. An RCOM ‘flavor’ is a combination of the base RCOM code plus specific configuration and implementation plug-in classes (for example, rcom-rmm-Na is the base flavor supporting RMS 10.2.2). These artifacts are assembled in the .ear as a deployable version of the software.

See the ‘Application deployment’ section later in this chapter for more details on the specific content of the .ear and .jar files.

## Technical design details

### Pluggable application services

Modules within RCOM that require version-specific implementation code (such as RMS integration points) encapsulate specific behavior behind a defined service interface. Within the module, the specific implementation class is loaded for the defined interface through the `PluggableServiceManager` class. This class is a façade in front of the Retek Platform service locator.

The runtime implementation packages are specified and loaded through entries in the following configuration file (this file controls the classes loaded by the `PluggableServiceManager`)

```
retek/services_rcom.xml
```

The following is a sample element from the services .xml definition defining a 'plug-in' package for a custom module implementation.

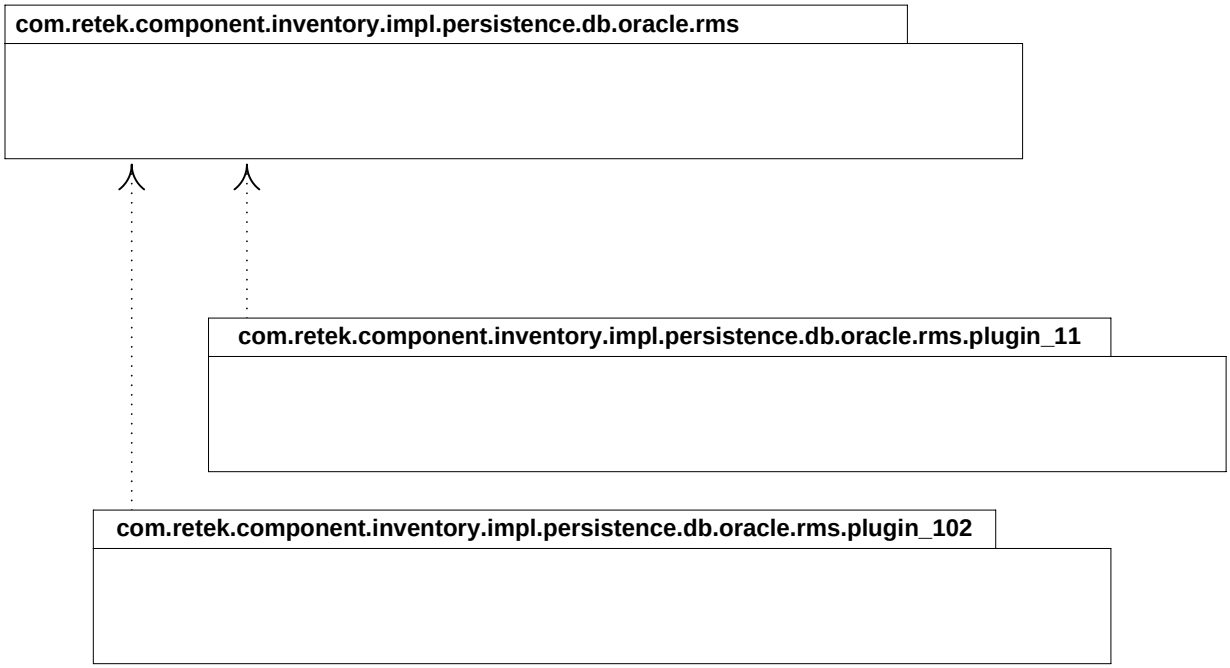
```
<!-- RMS Integration Plug-in -->
<interface
package="com.retek.component.inventory.impl.persistence.db.oracle.rms" app="rcom">
   <impl
package="com.retek.component.inventory.impl.persistence.db.oracle.rms.plugin_102" />
</interface>
```

The services .xml contains entries for all packages specified to run different implementation classes. With the current version of RCOM 11.x, these customizations are specific to particular versions of RMS, ReSA and RIB.

**Note:** Not all integration touch-points in the code where custom implementation classes exist have been migrated to using this approach for runtime loading with version RCOM 11.x. There are still some interfaces which are specified in the configuration by third-party integrators through the Java .properties files (for example, the payment authorization module).

### Custom plug-in example

The following is an example of the package structure of how custom code is implemented within the RCOM project for packages that provide pluggable implementations. This sample is for the code that supports integration with specific versions of RMS from within the Oracle DAO layer of the Inventory component.




| Package | Description |
|---|---|
| com.retek.component.inventory.impl.persistence.db.oracle.rms | Oracle db classes for queries and inserts general to all RMS schema and inventory data.<br>Interfaces defining pluggable services for RMS version specific db operations. |
| com.retek.component.inventory.impl.persistence.db.oracle.rms.plugin_11 | Oracle db classes for queries and inserts specific to RMS 11 schema and inventory data. |
| com.retek.component.inventory.impl.persistence.db.oracle.rms.plugin_102 | Oracle db classes for queries and inserts specific to RMS 10.2 schema and inventory data. |

## Application deployment

### Deployed RCOM EAR

The following is a detailed breakdown of the contents of the RCOM .ear file:

| Contents | Description |
|---|---|
| /lib | Dependent .jar files for third part libraries (ex. Platform .jar files, rib .jar files, apache .jar files, etc.) |
| /rcom.war | RCOM-RMM client application module (Java Webstart application) |
| /rcom-api.jar | RCOM-RMM services ejb module |
| /rib-injector-ejb.jar | RIB injector ejb module |
| /rcom-conf.jar | RCOM service configuration .jar (contains configuration properties for application client and server-side classes) |
| /rcom-plugin.jar | RCOM custom plug-in .jar (contains flavor specific |

The custom code for a specific RCOM flavor (for example, rcom-rmm-na) is deployed within the rcom-plugin.jar file. This .jar file contains the necessary files to configure a specific instance of rcom for a deployment, plus all the custom implementation code for that flavor.

### Custom rcom-plugin.jar contents\

| Contents | Description |
|---|---|
| /com/retek/component/.. | Custom plug-in implementation packages and classes. |
| /retek/jndi_providers.xml | Configuration file for external jndi providers (for example, rib target jndi provider for finding RIBMessagePublisher ejb) |
| /retek/services_rcom.xml | Configuration file which specifies the specific plugin implementation packages for the rcom custom flavor (deployment). |
| /retek/rib/injectors.xml | Configuration file which maps incoming rib messages family types (for RCOM message subscriptions). |

### Build/deployment process overview

With the inclusion of different RCOM build flavors, customizations for each flavor are now kept in a different location in the source from the core source code. A new rcom-flavors directory now exists in the source tree with subdirectories for each supported flavor.

Scripts located in the scripts subdirectory of each flavor handle build and deployment of the different RCOM flavors. These scripts set values for the flavor being built and then pass these values on to the master scripts in the scripts directory in the core, which then handle building and deploying the flavor.

The resulting generic deployable zip distribution is created in the build directory of the core. In the case of a remote deployment, the zip distribution is sent to the target server via FTP and deployed.

### Flavor directory details

The rcom-plugin.jar consists of any customizations that are present in the flavor directory. Each flavor directory is structured as follows:

-rcom-rmm-*flavor*
- -build
- -buildutil
- -conf
- -dist
- -doc
- -libs
- -resource
- -scripts
- -src

The *conf, resource*, and *src* directories serve the same purpose as the identically named directories in rcom-rmm-core. They store the configuration, resource, and source code files. The files stored in the flavor directory, however, are specific to the flavor.

The *buildutil* directory also serves the same purpose as in core. It stores files needed by the build process. In the case of the flavors, the only file stored here is the jboss Oracle configuration file, which differs for each flavor. This file is copied into the Jboss configuration for local Jboss development deploys.

The *libs* directory contains any jar files that are uniquely needed by the flavor. Generally this custom payload jar is needed for the RIB implementation to which the flavor is tied.

The *scripts* directory contains the build and push scripts for the flavor.

The *build* and *dist* directories serve the same purpose as in core. They hold transient files that are created by and for the build process. The *build* directory contains the build log files and any compiled class files specific to the flavor. The *dist* directory holds the rcom-plugin.jar and rcom-plugin-src.jar files created by the flavor's build process.

Finally, the *doc* directory contains any documentation specific to the flavor.

### Flavor build process

This section explains the flow process of the build when building the deployable package for an RCOM flavor. The example used is the *rcom-rmm-na* flavor with *na-test* as the target environment.

All flavor builds are 'kicked off' from the scripts directory in the flavor's directory tree. A build is kicked off using the build.bat or build.sh script and passing in the desired Ant target as a parameter.

```
rcom-flavors/rcom-rmm-na/scripts> sh build.sh package
```

The flavor's build script then calls the build script in core, passing in the location of the flavor directory and target environment's name (na-test).

The core's build script then invokes Ant using the *build.xml* file from the flavor, which sets some properties and in turn calls back to the core's *build.xml*, where the build properties are loaded using the file in core's *targets* directory with the environment's name—*build.na-test.properties* in this case. This build.xml redirection is unfortunate, but necessary in order to facilitate Ant builds of the flavors from within Eclipse.

The flavor's build.xml contains the bulk of the Ant build functionality. The file compiles the both the core and the flavor and creates a generic binary distribution of the core and the flavor plugin in core's *dist* directory, along with a zip file of the contents of the dist directory in core's *build* directory.

### Flavor deploy process

The RCOM flavor distribution can be installed on the server by using FTP to copy the binary distribution zip file to the server, unzipping it, and running the *deploy.sh* script. The deploy script takes as an argument a build properties file with the configuration settings for the local WebSphere instance to deploy to.

```
dist> sh deploy.sh build.mspdev37.properties
```

The deploy process reads in the WebSphere configuration settings for the application, customizes several of the files in the jars, creates a deployable ear file, and installs the ear on the target WebSphere instance.

The following files are customized during the deploy:

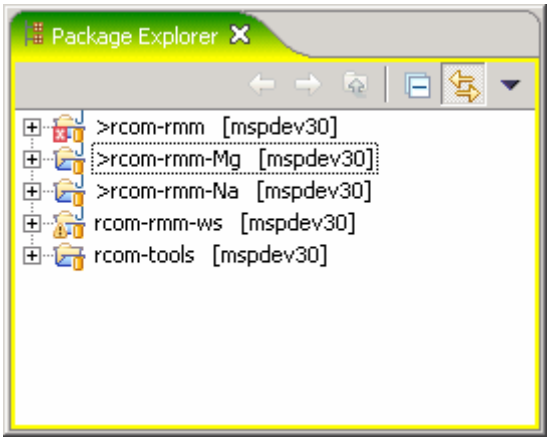
```
rcom-conf.jar!/jndi.properties
rcom-conf.jar!/salesAuditManager.properties
rcom-conf.jar!/com/retek/component/security/security.properties

rcom-plugin.jar!/retek/jndi_providers.xml

rcom.war!/rcom.jnlp
rcom.war!/rmm.jnlp
rcom.war!/index.hml

rcom.war!/META-INF/application.xml
```

## Eclipse development

### Eclipse project organization

The projects within the eclipse development environment have been reorganized to support a core project for the base code and new projects to maintain each of the RCOM server flavors.

| Eclipse Project | Description |
|---|---|
| rcom-rmm | Core application classes, interfaces and configuration items and tests. This project contains the |
| rcom-rmm-Na | Base rcom flavor, contains deployment configuration and plug-in implementation packages for RMS 11 data access overrides, ResA 11 integration classes. |
| rcom-rmm-Mg | Base rcom flavor, contains deployment configuration and plug-in implementation packages for RMS 10.2 data access overrides, ResA 10.2 integration classes. |
| rcom-rmm-tools | Configuration file which maps incoming rib messages family types (for RCOM message subscriptions) |
| rcom-rmm-utility | Utility scripts and tools for internal build process |



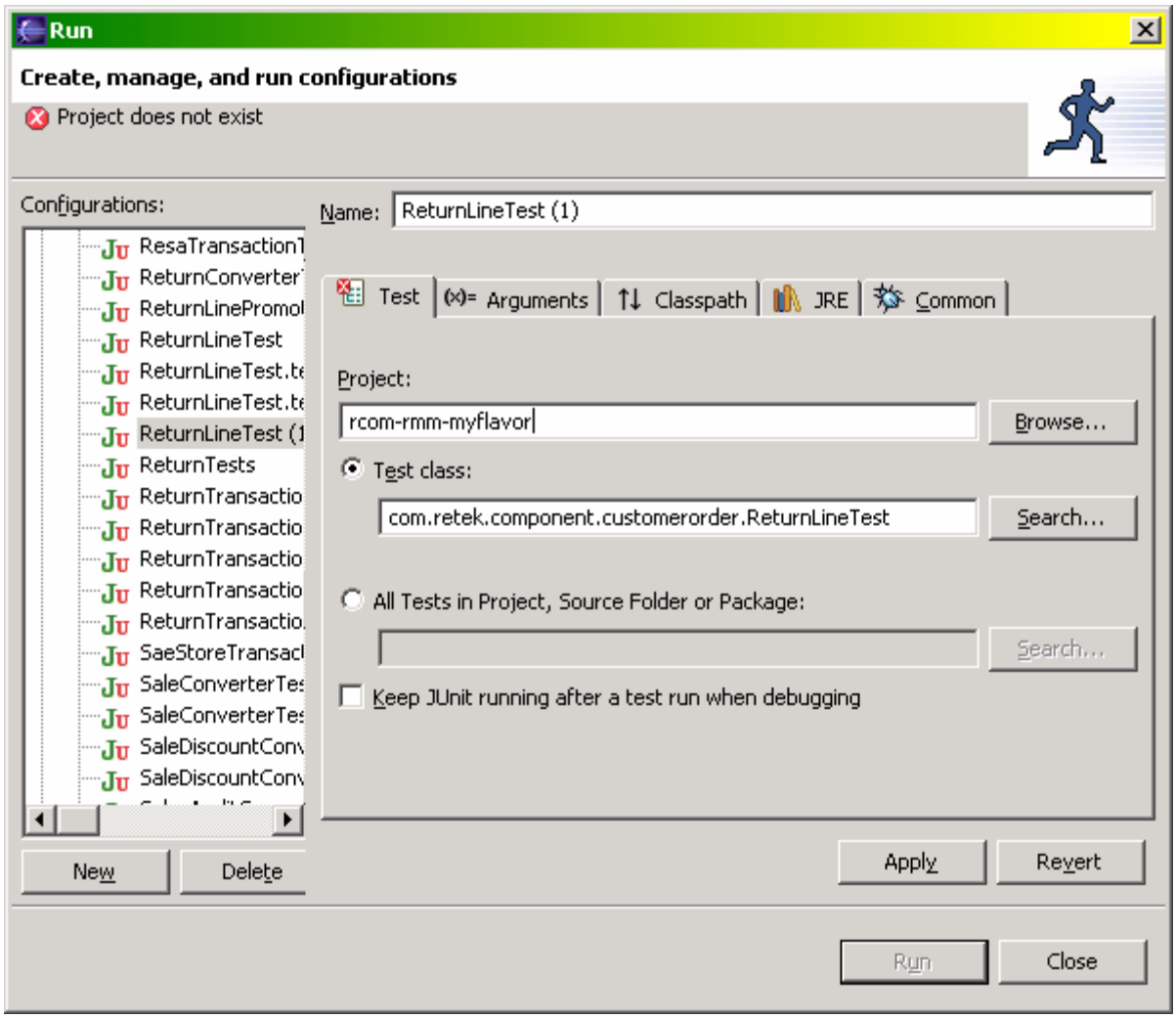
### Building RCOM-RMM in Eclipse

The following build scripts appear in the launch menu for each of the specific flavors (where xx is the specific custom flavor).

| Scripts | Description |
|---|---|
| xx-build-cache | Build the application for API junit testing (builds application for memory cache dao mode testing). |
| xx-build-oracle | Build the application for Oracle junit testing (builds application for oracle dao mode testing). |
| xx-deploy-local | Deploys the application to Jboss for local testing. |
| xx-build-clean | Clean the build directory. Deletes all old class files and config. files for a new local build or deploy. |

### Running tests in Ecplise

In order for the Junit tests to successfully run, they must be run through a custom flavor. To change the flavor that the test is running through, go to the Run menu for the specific test and change the project.




# RCOM-related architectural Java terms and standards

RCOM is deployed using the J2EE technologies, methods, versions and/or design patterns defined in this section.

**ACID**

ACID represents the four properties of every transaction:

- Atomicity: Either all of the operations bundled in the transaction are performed successfully or none of them are performed.
- Consistency: The transaction must leave any and all datastores that are affected by the transaction in a consistent state.
- Isolation: From the application's perspective, the current transaction is independent, in terms of application logic, from all other transactions running concurrently.
- Durability: The transaction's operations against a datastore must persist.

**Business delegate**

A J2EE design pattern in which the business delegate class exists primarily to pass calls through to another object for processing. For example, a business delegate on the client tier might handle requests for services that are fulfilled by a session EJB. Business delegates shield the API layer from implementation details. In practical terms, business delegates catch EJB-specific exceptions and either deal with the exceptional condition with additional logic, or throw a business-oriented exception that does not reveal to the outside world the fact that J2EE is used internally. In other words, one benefit of this pattern is that the business delegate can shield the component's client (the GUI, for example) from exceptions having to do with the remote call, and instead throw a business-appropriate exception.

**Data access object (DAO)**

A J2EE design pattern that isolates data access and persistence logic. The rest of the component can thus ignore the persistence details (the database type or version, for example).

**Data transfer object (DTO)**

A J2EE design pattern that provides a way to transfer data in bulk between the client and server. Consider the following scenario:

> You have a remote reference to a server-side ‘customer object’, and you want to retrieve the customer's name, address and phone number. Without a DTO, you would need to make three remote calls, getName(), getAddress() and getPhone(). These three calls represent a resource-expensive way to get three data elements. After all, each remote call requires a network access, the marshalling and unmarshalling objects, and so on. With a CustomerDTO object, on the other hand, you can accomplish the same task (for example, moving customer data from the server to the client) with just one remote call, getCustomerDTO.

**Enterprise Java Beans (EJB)**

EJB technology is from Sun. See http://java.sun.com/products/ejb/. EJB refers to a specification for a server-side component model. RCOM uses only stateless, session EJBs, which are stateless and clusterable, and which offer a remotely accessible entry point to an application server.

**Enterprise Java Beans (EJB) container**

An EJB container is the physical context in which EJBs exist. A container is a physical entity responsible for managing transactions, connection pooling, clustering, and so on. One example of an RCOM EJB container is Websphere. This container manages the execution of enterprise beans for J2EE applications.

**Instantiate**

In the context of RCOM, to ‘instantiate’ means to create a new instance of a class.

**J2EE server**

The runtime portion of a J2EE product. A J2EE server provides EJB and Web containers.

**The Java 2 Enterprise Edition (J2EE)**

The Java standard infrastructure for developing and deploying multi-tier applications. Implementations of J2EE provide enterprise-level infrastructure tools that enable such important features as database access, client-server connectivity, distributed transaction management, and security.

**Java Development Kit (JDK), version 1.3.1**

Standard Java development tools from Sun Microsystems.

**JDBC**

JDBC is a means for Java-architected applications such as RCOM to execute SQL statements against an SQL-compliant database, such as Oracle. Part of Sun's J2EE specification, most database vendors implement this specification.

**Java Messaging Service (JMS) topic**

A JMS topic is part of Retek's message-oriented middleware. RCOM uses a JMS view of the RIB. The topic can be thought of as broadcasting a message from the RIB. RCOM (and potentially other subscribers) can subscribe to that broadcast. For example, the JMS topic, EtBannerFromRMS, handles banner-related messages.

**Naming conventions in Java**

- Packages: The prefix of a unique package name is always written in all-lowercase letters (for example, com.retek.component.banner_channel.integration.rib)
- Classes: These descriptive names are unabbreviated nouns that have both lower and upper case letters. The first letter of each internal word is capitalized (for example, BannerCreateInjector).
- Interfaces: These descriptive names are unabbreviated nouns that have both lower and upper case letters. The first letter of each internal word is capitalized.
- Methods: Methods begin with a lowercased verb. The first letter of each internal word is capitalized (for example, getName(), getAddress() and getPhone()).

**Persistence**

The protocol for transferring the state of an entity bean between variables and an underlying database.

**Persistent connections**

The state of connection between an application and the database. A transactional front-end application most often maintains a continuous connection with the database as the user navigates from form to form to complete a transaction.

**Remote interface**

The client side interface to an EJB. This interface defines the server-side methods available in the client tier.

**Session enterprise Java bean (EJB)**

A type of J2EE distributed component that gives client programs access to application server business functionality (also known as the middle-tier). Conceptually similar to other Java classes, EJBs also have to take care of all the details of being distributed and transactional.

# Chapter 3 – RCOM and the Retek Integration Bus (RIB)

## Overview

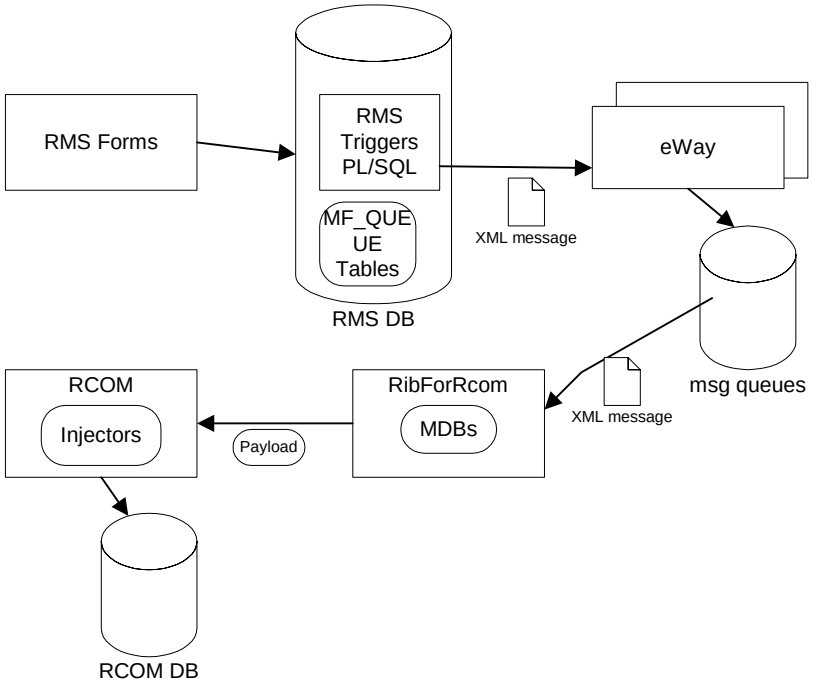
Retek utilizes a publish and subscribe (pub/sub) messaging paradigm with some guarantee of delivery for a message. In a pub/sub messaging system, an adapter publishes a message to the integration bus that is then forwarded to one or more subscribers. The publishing adapter does not know, nor care, how 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 RIB.

See the latest Retek Integration Guide and other RIB-related documentation for additional information.

# General message flow

## RCOM Message Subscription

The diagram below depicts the flow for a message subscription within RCOM. The diagram illustrates the injection of information into RCOM from another application through the RIB. The example in this diagram is a subscription from RMS.




**Message subscription flow**

1 Information is collected from RMS forms and then entered into RMS tables.

2 RMS table triggers fire. These call PL/SQL to generate XML message content based on specified DTD templates. XML messages are written to RMS MF_QUEUE tables.

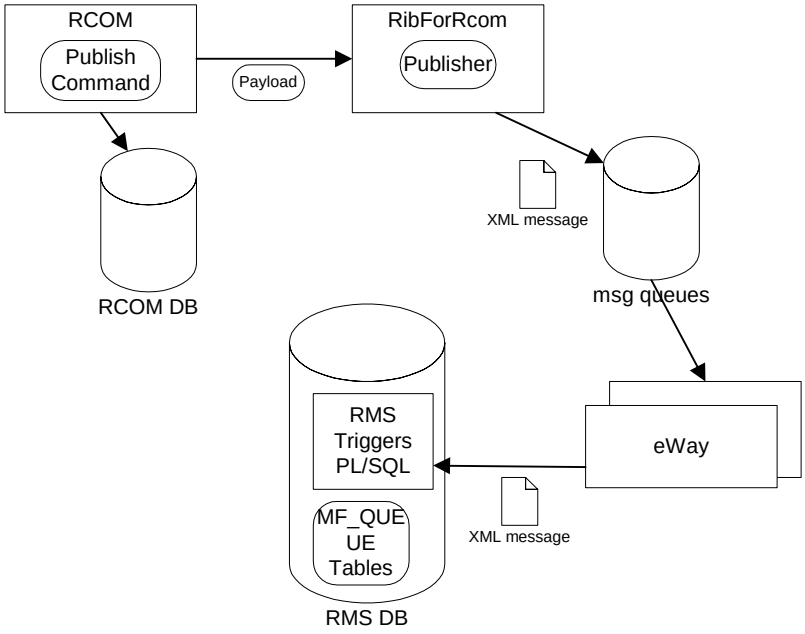
3 SeeBeyond eWay processes move the messages from the MF_QUEUE tables into message queues.

4 RibForRcom Message-Driven beans (MDBs) read the XML messages from the SeeBeyond message queues via JMS.

5 RCOM injector classes are invoked (by RIB MDBs) which process the message and push data into the RCOM application.

## RCOM message publication

The diagram below depicts the flow for messages being published from RCOM to other applications. The example in this diagram is a publication to RMS.




**Message publication flow**

1 Information is collected in RCOM.

2 RCOM remote command logic is invoked through business component EJBs (for example, during the order submit process). RIB payload objects are created and populated.

3 RIB payload objects are published by RCOM through a remote call to RibForRcom through the J2EE client plug-in EJB.

4 RibForRcom publisher EJB translates payload objects into XML messages which are written via JMS to SeeBeyond message queues.

5 The SeeBeyond eWay processes listen for the message and call inject procedures in PL/SQL on the RMS side.

# Subscribers mapping table

The table below lists the following:

- The RIB message family and message type name.
- The document type definition (DTD) that describes the XML message.
- The component (for more information about components, see "Chapter 5 – Component overviews").
- The subscribing classes that are interfaces. These classes are described in the code as 'injectors'.
- The Impl class that applies the business logic. The component is configured to act upon and/or validate the data.

For additional information, see the latest Retek Integration Guide and other RIB documentation.

| Family | Type | DTD/ Payload | Component | Injector | Injector Impl Class |
|---|---|---|---|---|---|
| ASNOUT | ASNOUTCRE | ASNOutDesc | customerorder | ShipmentConfirmationInjector | com.retek.component.customerorder.impl.rib.ShipmentConfirmationInjector |
| BANNER | BANNERCRE | BannerDesc | banner_channel | BannerCreateInjector | com.retek.component.banner_channel.impl.rib.BannerCreateInjector |
| BANNER | BANNERMOD | BannerDesc | banner_channel | BannerModifyInjector | com.retek.component.banner_channel.impl.rib.BannerModifyInjector |
| BANNER | CHANNELCRE | ChannelDesc | banner_channel | ChannelCreateInjector | com.retek.component.banner_channel.impl.rib.ChannelCreateInjector |
| BANNER | CHANNELMOD | ChannelDesc | banner_channel | ChannelModifyInjector | com.retek.component.banner_channel.impl.rib.ChannelModifyInjector |
| CUSTRETURN | CORETCRE | CustRetDesc | customerorder | ReturnConfirmationInjector | com.retek.component.customerorder.impl.rib.ReturnConfirmationInjector |
| DIFFGRP | DIFFGRPDTLCRE | DiffGrpDtlDesc | item | DifferentiatorGroupDetailCreateInjector | com.retek.component.item.impl.rib.DifferentiatorGroupDetailCreateInjector |
| DIFFGRP | DIFFGRPDTLMOD | DiffGrpDtlDesc | item | DifferentiatorGroupDetailModifyInjector | com.retek.component.item.impl.rib.DifferentiatorGroupDetailModifyInjector |

| Family | Type | DTD/ Payload | Component | Injector | Injector Impl Class |
|---|---|---|---|---|---|
| DIFFGRP | DIFFGRPHDRCRE | DiffGrpHdrDesc | item | DifferentiatorGroupHeaderCreateInjector | com.retek.component.item.impl.rib.DifferentiatorGroupHeaderCreateInjector |
| DIFFGRP | DIFFGRPHDRMOD | DiffGrpHdrDesc | item | DifferentiatorGroupHeaderModifyInjector | com.retek.component.item.impl.rib.DifferentiatorGroupHeaderModifyInjector |
| DIFFS | DIFFCRE | DiffDesc | item | DifferentiatorCreateInjector | com.retek.component.item.impl.rib.DifferentiatorCreateInjector |
| DIFFS | DIFFMOD | DiffDesc | item | DifferentiatorModifyInjector | com.retek.component.item.impl.rib.DifferentiatorModifyInjector |
| GIFTREG | GIFTREGACKCRE | GiftRegAckDesc | customerorder | GiftRegistryUpdateAcknowledgementInjector | com.retek.component.customerorder.impl.rib.GiftRegistryUpdateAcknowledgementInjector |
| ITEMLOC | ITEMLOCCRE | ItemLocDesc | item | ItemLocationCreateInjector | com.retek.component.item.impl.rib.ItemLocationCreateInjector |
| ITEMLOC | ITEMLOCMOD | ItemLocDesc | item | ItemLocationModifyInjector | com.retek.component.item.impl.rib.ItemLocationModifyInjector |
| ITEMS | ITEMBOMCRE | ItemBOMDesc | item | PackItemCreateInjector | com.retek.component.item.impl.rib.PackItemCreateInjector |
| ITEMS | ITEMCRE | ItemDesc | item | ItemCreateInjector | com.retek.component.item.impl.rib.ItemCreateInjector |
| ITEMS | ITEMHDRMOD | ItemHdrDesc | item | ItemModifyInjector | com.retek.component.item.impl.rib.ItemModifyInjector |
| ITEMS | ITEMSUPCRE | ItemSupDesc | item | ItemSupplierCreateInjector | com.retek.component.item.impl.rib.ItemSupplierCreateInjector |
| ITEMS | ITEMSUPCTYCRE | ItemSupCtyDesc | item | ItemSupplierCountryAttributeCreateInjector | com.retek.component.item.impl.rib.ItemSupplierCountryAttributeCreateInjector |

| Family | Type | DTD/ Payload | Component | Injector | Injector Impl Class |
|---|---|---|---|---|---|
| ITEMS | ITEMSUPCTYMOD | ItemSupCtyDesc | item | ItemSupplierCountryAttributeModifyInjector | com.retek.component.item.impl.rib.ItemSupplierCountryAttributeModifyInjector |
| ITEMS | ITEMSUPMOD | ItemSupDesc | item | ItemSupplierModifyInjector | com.retek.component.item.impl.rib.ItemSupplierModifyInjector |
| ITEMS | ITEMUDAFFCRE | ItemUDAFFDesc | item | ItemFreeFormUdaCreateInjector | com.retek.component.item.impl.rib.ItemFreeFormUdaCreateInjector |
| ITEMS | ITEMUDAFFMOD | ItemUDAFFDesc | item | ItemFreeFormUdaModifyInjector | com.retek.component.item.impl.rib.ItemFreeFormUdaModifyInjector |
| ITEMS | ITEMUDALOVCRE | ItemUDALOVDesc | item | ItemUdaListOfValuesCreateInjector | com.retek.component.item.impl.rib.ItemUdaListOfValuesCreateInjector |
| MEDIA | DROPCODECRE | DropCodeDesc | media | DropCodeCreateInjector | com.retek.component.media.impl.rib.DropCodeCreateInjector |
| MEDIA | DROPCODEDEL | DropCodeRef | media | DropCodeDeleteInjector | com.retek.component.media.impl.rib.DropCodeDeleteInjector |
| MEDIA | MEDIACRE | MediaDesc | media | MediaCreateInjector | com.retek.component.media.impl.rib.MediaCreateInjector |
| MEDIA | SOURCECODECRE | SourceCodeDesc | media | SourceCodeCreateInjector | com.retek.component.media.impl.rib.SourceCodeCreateInjector |
| MEDIA | SOURCECODEDEL | SourceCodeRef | media | SourceCodeDeleteInjector | com.retek.component.media.impl.rib.SourceCodeDeleteInjector |
| ORDER | PODTLCRE | PODesc | customerorder | ECDDRecalculationInjector | com.retek.component.customerorder.impl.rib.ECDDRecalculationInjector |

| Family | Type | DTD/ Payload | Component | Injector | Injector Impl Class |
|---|---|---|---|---|---|
| ORDER | PODTLMOD | PODesc | customerorder | ECDDRecalculationInjector | com.retek.component.customerorder.impl.rib.ECDDRecalculationInjector |
| ORDER | PODTLDEL | PORef | customerorder | ECDDRecalculationInjector | com.retek.component.customerorder.impl.rib.ECDDRecalculationInjector |
| SEEDDATA | CODEDTLCRE | CodeDtlDesc | codes | CodeCreateInjector | com.retek.component.codes.impl.rib.CodeCreateInjector |
| SEEDDATA | CODEDTLMOD | CodeDtlDesc | codes | CodeUpdateInjector | com.retek.component.codes.impl.rib.CodeUpdateInjector |
| SEEDDATA | CODEHDRCRE | CodeHdrDesc | codes | CodeHeaderCreateModifyInjector | com.retek.component.codes.impl.rib.CodeHeaderCreateModifyInjector |
| SEEDDATA | CODEHDRMOD | CodeHdrDesc | codes | CodeHeaderCreateModifyInjector | com.retek.component.codes.impl.rib.CodeHeaderCreateModifyInjector |
| SEEDDATA | DIFFTYPECRE | DiffTypeDesc | item | DifferentiatorTypeCreateInjector | com.retek.component.item.impl.rib.DifferentiatorTypeCreateInjector |
| SEEDDATA | DIFFTYPEMOD | DiffTypeDesc | item | DifferentiatorTypeModifyInjector | com.retek.component.item.impl.rib.DifferentiatorTypeModifyInjector |
| SOSTATUS | SOSTATUSCRE | SOStatusDesc | customerorder | StockStatusInjector | com.retek.component.customerorder.impl.rib.StockStatusInjector |
| STORES | STORECRE | StoreDesc | location | StoreCreateInjector | com.retek.component.location.impl.rib.StoreCreateInjector |
| STORES | STOREMOD | StoreDesc | location | StoreModifyInjector | com.retek.component.location.impl.rib.StoreModifyInjector |
| UDAS | UDAHDRCRE | UDADesc | item | UdaHeaderCreateInjector | com.retek.component.item.impl.rib.UdaHeaderCreateInjector |

| Family | Type | DTD/ Payload | Component | Injector | Injector Impl Class |
|---|---|---|---|---|---|
| UDAS | UDAHDRMOD | UDADesc | item | UdaHeaderModifyInjector | com.retek.component.item.impl.rib.UdaHeaderModifyInjector |
| UDAS | UDAVALCRE | UDAValDesc | item | UdaValueCreateInjector | com.retek.component.item.impl.rib.UdaValueCreateInjector |
| UDAS | UDAVALMOD | UDAValDesc | item | UdaValueModifyInjector | com.retek.component.item.impl.rib.UdaValueModifyInjector |
| VENDOR | VENDORADDRCRE | VendorAddrDesc | supplier | VendorAddressCreateInjector | com.retek.component.supplier.impl.rib.VendorAddressCreateInjector |
| VENDOR | VENDORADDRMOD | VendorAddrDesc | supplier | VendorAddressModifyInjector | com.retek.component.supplier.impl.rib.VendorAddressModifyInjector |
| VENDOR | VENDORCRE | VendorDesc | supplier | VendorCreateInjector | com.retek.component.supplier.impl.rib.VendorCreateInjector |
| VENDOR | VENDORHDRMOD | VendorHdrDesc | supplier | VendorHeaderModifyInjector | com.retek.component.supplier.impl.rib.VendorHeaderModifyInjector |
| WH | WHCRE | WHDesc | location | WarehouseCreateInjector | com.retek.component.location.impl.rib.WarehouseCreateInjector |
| WH | WHMOD | WHDesc | location | WarehouseModifyInjector | com.retek.component.location.impl.rib.WarehouseModifyInjector |

# Publishers mapping table

This table illustrates the relationship among the message family, message type, the DTD/payload object, and the Impl class that applies the business logic. For additional information, see the latest Retek Integration Guide and other RIB documentation.

| Family | Type | DTD/Payload | Publisher Impl Class |
|---|---|---|---|
| COCOGS | COGSCRE | CogsDesc | com.retek.rib.binding.payload.CogsDesc |
| CODSRCPT | DSRCPTCRE | DSRcptDesc | com.retek.rib.binding.payload.DSRcptDesc |

| Family | Type | DTD/Payload | Publisher Impl Class |
|---|---|---|---|
| CORRESPONDENCE | CUSTCORRESCRE | CustCorresDesc | com.retek.rib.binding.payload.CustCorresDesc |
| COSALE | CUSTSALECRE | CustSaleDesc | com.retek.rib.binding.payload.CustSaleDesc |
| CUSTORDER | COCRE | CODesc | com.retek.rib.binding.payload.CODesc |
| CUSTORDER | CODEL | CORef | com.retek.rib.binding.payload.CORef |
| CUSTRETURN | CALLTAGCRE | CallTagDesc | com.retek.rib.binding.payload.CallTagDesc |
| GIFTREG | GIFTREGUPDMOD | GiftRegUpdDesc | com.retek.rib.binding.payloadGiftRegUpdDesc |
| INVADJUST | INVADJUSTCRE | InvAdjustDesc | com.retek.rib.binding.payload.InvAdjustDesc |
| PAYMENTS | REFDPAYSTLMTCRE | RefdPayStlmtDesc | com.retek.rib.binding.payload.RefdPayStlmtDesc |
| PENDRETURN | PENDRETCRE | PendRtrnDesc | com.retek.rib.binding.payload.PendRtrnDesc |
| WOOUT | OUTBDWOCRE | WOOutDesc | com.retek.rib.binding.payload.WOOutDesc |

# Configuration of customized RCOM RIB classes

### Injector configuration

The injector implementation classes which are invoked for a particular application message subscription within RCOM are controlled by the following configuration file, which is deployed within the `rcom-plugin.jar.`

```
/retek/rib/injectors.xml
```

The RIB ApplicationMessageInjector ejb (deployed in the rib-injector-ejb.jar on the RCOM server instance) is the entry point for a RIB message injection (subscription) into RCOM. This ejb looks up the appropriate injector class for a given message family/type using the aforementioned .xml configuration file.

The following is an example of an entry within the `injectors.xml` file. It specifies an injector for each type / family which can be injected into RCOM.

```
<family name="BANNER">
```

```
   <injector
class="com.retek.component.banner_channel.impl.rib.plugin__11.Banner
CreateInjector">
      <type>BANNERCRE</type>
   </injector>
   …
</family>
```

### Publisher/payload factory configuration

The custom publisher implementation classes which are invoked for a particular application message publication within RCOM are controlled by the following configuration file, which is deployed within the `rcom-plugin.jar`.

```
/retek/services_rcom.xml
```

Any customization which is needed within RCOM to provide variances in payload creation is encapsulated via the implementation of a factory interface as specified by this xml. The custom implementations are deployed in a custom plug-in package.

The following is an example of an entry within the services_rcom.xml file:

```
<!-- CustomerOrder Integration Plug-in -->
<interface
package="com.retek.component.customerorder.impl.remote.command"
app="rcom">
   <impl
package="com.retek.component.customerorder.impl.remote.command.plugi
n__11" />
</interface>
```

# RCOM RIB component

The RcomRib component contained within RCOM 11 provides an encapsulation layer for all RCOM business component interaction with the RIB. The purpose of this component is to provide a simple layer of abstraction around RCOM's dependency to the RIB client API.

This component provides interfaces, classes and services for RIB payload message publications and subscriptions. Business components within RCOM leverage this component to interact with the RIB. Specific payload object creation or injection logic is implemented by business component specific publish commands and injector classes. Any communication with the RIB (such as publishing of Payload objects) is handled through this component.

### RCOM RIB Package Overview

The RcomRib component is composed of the following packages:

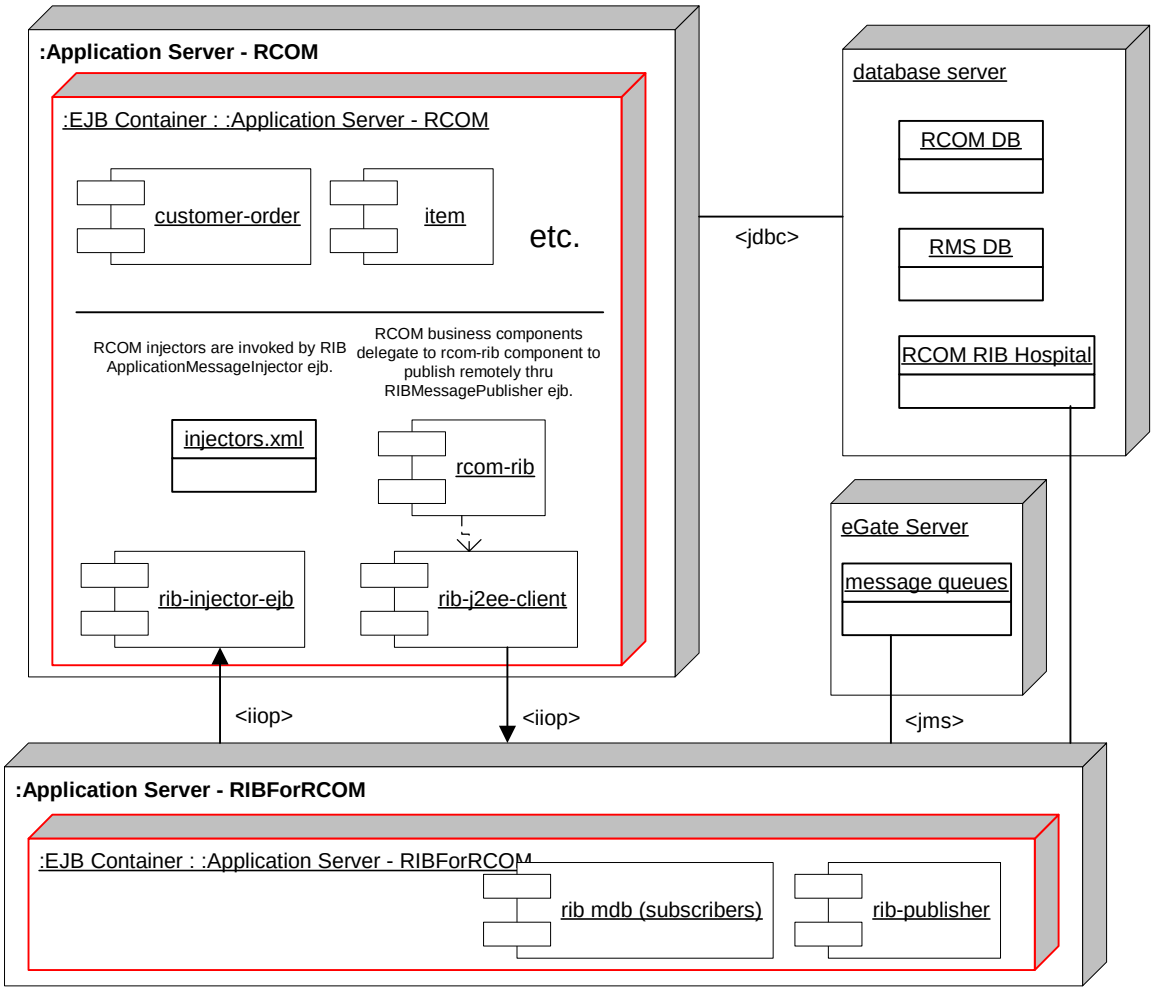
| Package | Description |
|---|---|
| com.retek.component.rcomrib | RcomRib public api and utilities. |
| com.retek.component.rcomrib.impl | RcomRib implementation |

| Package | Description |
|---|---|
| com.retek.component.rcomrib.impl.remote | RcomRib remote services |
| com.retek.component.rcomrib.test | RcomRib test services and utilities. |

# Application deployment

## RCOM RIB deployment diagram

The following diagram depicts a view of how components are deployed within RCOM with respect to the RIB.



## RCOM RIB deployment details

The following is a table of the .jar files required within the RCOM server deployment relating to the RIB and a brief description of their purpose.

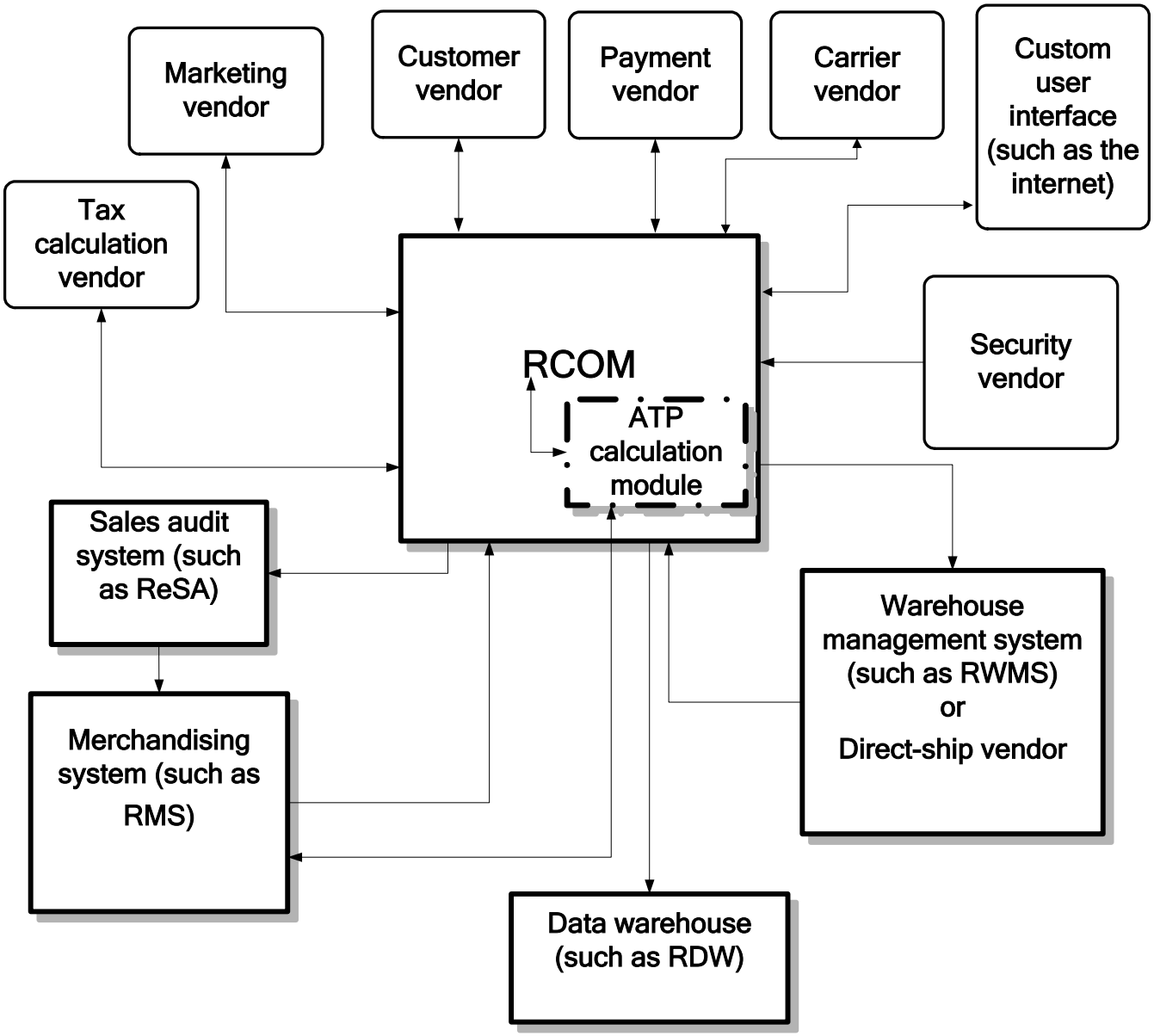
| Jar files | Description |
|---|---|
| rib-injector-ejb.jar | RIB server-side ejb module for remote injector services. This .jar is specified as an ejb module in the application.xml deployment descriptor within the rcom.ear. |
| rib-j2ee-client.jar | RIB client-side ejb plug-in containing stubs for remote services. |
| retek-payload.jar | RIB *base* message payloads. |
| custom-payload.jar | RIB *custom* message payloads |
| retek-rib-support.jar | RIB common utilities, interfaces and support classes. |

# Chapter 4 – Interface process flows

## Overview

This chapter provides an overview as to how RCOM is functionally integrated with other systems (including other Retek systems). The discussion primarily concerns the flow of RCOM-related business data across the enterprise.

A diagram shows the overall direction of the data among the products. The accompanying explanations of this diagram are written from a system-to-system perspective, illustrating the movement of data. Note that this discussion focuses on the functional use of data; the means of data movement (for example, the RIB, batch, and so on) is not illustrated in this chapter.




**RCOM-related dataflow across the enterprise**

# Available to promise (ATP) processing

For more information regarding ATP processing, see the section, ‘Inventory component (including the ATP module)’ in “Chapter 5 – Component overviews and interface(s)”.

**Note:** Although the ATP module resides within the Inventory component of the RCOM application, the module is depicted outside the application to better illustrate the logic of the ATP processing dataflow.

## From RCOM to the ATP module

RCOM uses the ATP module for one of the following reasons:

- To place order line reservations and/or back orders.
- To release order lines because the merchandise has either been cancelled or shipped.

In other words, ATP is used to either reserve inventory, reserve a back order, or release a reservation or a back order.

The ATP module determines whether or not an order line will be fulfilled as a direct ship.

The data that RCOM sends the ATP module, thereby driving the reservation process, includes:

- Item
- Banner
- Channel type
- Requested quantity

## From the merchandising system to the ATP module

The merchandising system sends the ATP module the following data:

- Stock on hand quantity data
- Unavailable quantity data
- Approved purchase order data
- In-transit quantity data
- Transfer expected quantity data
- Transfer reserved quantity data
- Virtual warehouse reserved quantity

## From the ATP module to RCOM

The ATP module sends RCOM the following merchandise quantity data:

- How much merchandise is reserved
- How much merchandise is back ordered
- How much merchandise is no longer available

- The estimated ship date
- In a direct ship scenario, the ATP module determines whether or not the order line is a direct ship and sends RCOM the direct-ship supplier-related data.

# Custom user interface (such as the internet)

### From RCOM to the custom user interface (such as the internet)

- Selling SKUs for a given selling item
- Current stock and delivery information for a given selling SKU
- Customers and associated customer data for a given set of search criteria, including customer number, name, and address information
- History events for a given customer
- Existing orders and associated order data for a given set of search criteria, including order number, customer number, and various customer information
- Catalog request data for a given concept and subconcept, to be sent to a specified customer address
- Pricing, shipping, and tax information, and associated totals, for a given set of selling SKUs (with quantities)
- Shipped container information

### From the custom user interface (such as the internet) to RCOM

- The custom user interface sends the input arguments that RCOM needs to perform its tasks. RCOM must have order and product information so that it can submit the order in its system. Data includes:
  - Customer
  - Catolog request
  - Customer order

# Foundation and code data

### From the merchandising system to RCOM

RCOM receives foundation data from the merchandising system including:

- Banners and channels
- Codes (for example, item types, carriers, shipping methods, return reasons and so on)
- Differentiator groups
- Differentiator identifiers
- Items

- Locations (stores and warehouses)
- User-defined attributes (UDAs)
- Vendor and vendor addresses

### From the marketing vendor to RCOM

From the third party marketing system, RCOM imports the following:

- Drop code
  A media may have different 'drops', which can be thought of as subsets of a media. For example, a catalog might use a different cover, although the pricing structure within remains the same. A media may have more than one drop code.
- Source code relationship
  Customers and prospects can be grouped by characteristics, such as demographics. This grouping takes place in a third-party marketing system. Source codes are used to uniquely identify such customer segments.

  Source codes are associated to drops, and drops are associated to the media. For example, drop 1 has these source codes associated to it; drop 2 has these source codes associated to it, and so on. Once the media header is created based on this data, RCOM knows, for example, that this media has these 4 drops with source codes associated to each.

For more information, see the section, 'Media component' in "Chapter 5 – Component overviews and interface(s)".

### From the customer vendor to RCOM

- New customer data and updates to existing customer data
- Merge requests

### From RCOM to the customer vendor

- New customer data and updates to existing customers
- Merge requests

## Order fulfillment

### From the distribution management system to RCOM

- Returns data
  Returns data that includes the item and inventory is communicated to RCOM from the distribution management system.
- Shipment confirmation data
  RCOM uses the line level status to facilitate payment settlements for shipped products (for partial shipments) and for container shipment tracking for the line items shipped on the customer order line.

  - Return Merchandise Authorization (RMA) number
    A Return Merchandise Authorization (RMA) number is a number that authorizes the return of a product.

- Stock order statuses when applicable
  The distribution management system sends customer order line status update data at the order line container level to RCOM. Such data includes:
  - Pick exception
    - Insufficient inventory data
    - Expired order lines data
  - Cancel confirmation
    The distribution management system sends RCOM a successful delete message to confirm in RCOM that an order line can be cancelled.

### From RCOM to the warehouse management system

- Released order line data (in ‘live’ status)
  Once RCOM determines that the item has been reserved and that the payment has been authorized, RCOM sends the released order line data to the distribution warehouse system with a ‘live’ status. The distribution management system facilitates the picking, packing, and shipping of the item(s) in the order line.
- Other order line data not to be picked (in ‘shipped’ status, ‘reserved’ status, and so on)
  To prevent customer confusion, RCOM publishes the rest of the item data on an order (in addition to the item data that is being shipped) and marks it with a different status (‘shipped’, ‘reserved’, and so on). For example, even if only one of ten items is sent to the warehouse for picking, a printed invoice shows the status of all ten items.
- ‘Attempt to cancel’ data
  RCOM sends the distribution management system data that attempts to cancel the order line. Whether the order line can be cancelled or whether the item(s) have already been shipped is determined by the distribution management system.
- Pending returns

# Payment processing

### From RCOM to the payment vendor

- Authorization and settlement data
  To facilitate the authorization and the settlement of fulfilled item(s), RCOM submits the following data:
  - Payment type and associated attributes (account types, SID numbers, and so on)
  - Authorization/settlement amount
  - Customer name and address information

### From the payment vendor to RCOM

- Authorization status data
  The payment vendor sends the authorization status to RCOM for the authorization amount.
- Address verification system (AVS) code data
  Once the payment vendor processes and matches the customer name and address information from RCOM, the payment vendor returns an AVS code.

# Reporting

### From RCOM to the data warehouse

- For a data warehouse's reporting purposes, RCOM can provide both dimension data and fact data that it obtains during normal, day-to-day processing. Note that the data that RCOM sends to the data warehouse is determined by the data warehouse. See the RCOM Operations Guide for more information about RETL batch processing and the data that is sent.

# Sales and other transactions processing

### From RCOM to the sales audit system

- Sales and other transactions data through the RTLOG and/or sales audit XML files
  A sales audit system can accept transaction data from various front-end systems and move the data through a series of processes that culminates in 'clean data'. It flags inaccurate data for sales auditors, who can then correct the errors. By running transactions from both the customer order line management and point of sale applications through a sales audit system, a standard transaction data flow is enforced cross the entire enterprise.

  For more information regarding the RTLOG and RCOM's interaction with a sales audit system, see the RCOM Operations Guide.

# Security processing

### From a security vendor to RCOM

**Note:** RCOM never writes data to Active Directory.

- User-related data
  An RCOM batch process runs and pulls new and/or modified user-related data from the security vendor and, after a validation step, persists the data within RCOM (thus ensuring that the two systems are in sync). Such data could include, for example, username, location, supervisor, and so on. The security vendor's system does not contain mappings of users to roles or roles to permissions. RCOM provides the mappings between users and roles and roles and permissions. User locations are validated through the use of Vertex (which contains valid address data).

# Shipment tracking

### From the distribution management system to RCOM

- Shipment tracking data
  To facilitate shipment tracking (utilizing a tracking number by shipment), the distribution management system informs RCOM that a particular order line has shipped and informs RCOM about the carrier details associated with that shipment.

### Accessing the carrier vendor from RCOM

- Shipment status data
  To facilitate customer service, RCOM provides access to the carrier's website, which tracks the status of a shipment. RCOM sends the tracking number to the website, and the website shows the tracking information on the website.

### From the carrier vendor to RCOM

- Every container has a tracking number unique to the carrier that delivers it. RCOM accepts the carrier ID, tracking number, and delivery date as delivery confirmation data. These fields allow RCOM to find the correct shipped container and update it with the delivery date/time.

# Tax calculation

### From the tax calculation vendor to RCOM

- Calculated taxes for a given order line
  The vendor uses its tax calculation engine to return the following types of calculated taxes for an order line:
  - Net merchandise tax
  - Shipping and handling tax (including additional delivery charge tax)
  - Gifting tax
  - Personalization tax
- Error code data
  The tax calculation vendor passes error codes to the order entry system.
- GEOCODE(s) data
  The tax calculation vendor sends RCOM valid GEOCODE(s) for a given postal code that is passed to the tax calculation vendor from RCOM.
- Tax rate data
  The tax calculation vendor passes error codes to the order entry system.
- Customer tax exemption data
  The tax calculation vendor determines whether a customer is tax exempt.

## From RCOM to the tax calculation vendor

- Amounts that require tax calculations
  - Gifting
  - Personalization
  - Net merchandise amount
  - Shipping and handling amount (including additional delivery charge tax)
- Postal code data
  RCOM sends postal codes to the tax calculation vendor anticipating all the matching city-state-county combinations (the GEOCODE) that are valid for that postal code.
- Tax credits data
  - Credits specific to a tax credit accommodation type
  - Returns-related credits to update the tax liability

# Chapter 5 – Component overviews and interface(s)

## Introduction

This chapter provides information concerning the various aspects of RCOM's components (for example, the customer component, the location component, the inventory component, and so on).

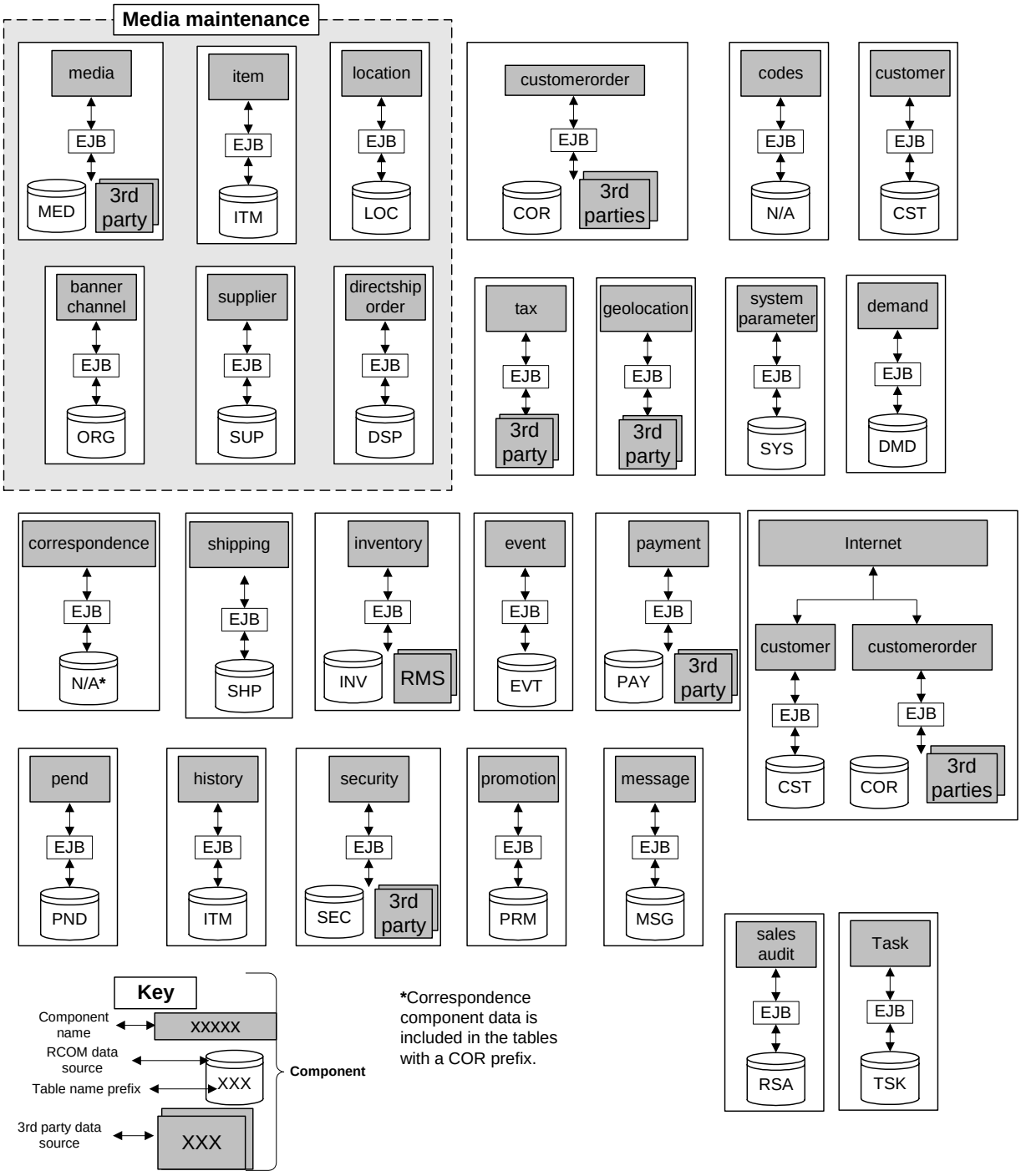
At the beginning of the chapter, a high-level RCOM component map is provided for your reference.

Information within each component section could include the following:

- A functional overview that includes the business processing for which the component is responsible
- Information about the component's integration with 3rd party system(s)
- Sample flat file(s) that would be output from RCOM to third party system(s)
- Parameters associated to the component
- The location of a component's package(s) within Javadoc
- Whether or not batch process(es) are associated with the component
- Whether or not the RIB is associated with the component
- Other important information related to the component

# RCOM component map with interfaces

The following diagram provides a high-level diagram of RCOM's components.




**High-level mapping of components and data sources**

# Banner and channel component (including banner-level parameters)

## Functional overview

### Banner

Banner plays a critical role within RCOM. With the exception of some customer data, which is held globally, most of RCOM's data is held at the banner level.

Some of the important elements of business functionality that depend upon banner-related data include the following:

- Although customers are global and are not tied to a banner, preferences exist within 'a customer' that are tied to a banner.
- Credit cards
- Sales data exported to a sales audit system
- History
- Orders
- Activity requests may or may not be tied to a specific banner.
- Shipping methods and carrier
- Personalization fees
- Monogramming fees
- Gifting services and gifting seasons

### Channel

Some of the important elements of the business functionality that depend upon channel data include the following:

- Channel is used in inventory management. Stores and warehouses are assigned to channels. The channel type is a code that originates in the merchandising system. Channel type is often used behind the scenes during, for example, aggregations of inventory. Channel types include brick and mortar, catalog, web/internet, iiosk, and so on.
- Tender types and order sources are related to channels. If an internet order is being taken, cash may not be a valid tender type, and so on. This relationship is configurable.

## A functional description of the banner_channel subscription from the RIB

**Note:** In a multi-channel environment, before a location can be successfully consumed by the subscribing application, any channels or banners that the location references must have already been successfully consumed by the same subscribing application.

RCOM subscribes to banner-related messages from the RIB. The messages are published by the merchandising system.

Valid values for banner data must be kept in sync with the external system. All banner data within RCOM is a reflection of the banner data in the merchandising system.

RCOM subscribes to channel-related messages from the RIB. The messages are published by the merchandising system.

Valid values for the channel-attribute data must be kept in sync with the external system. All channel attribute data within RCOM is a reflection of the channel-attribute data in the merchandising system.

## Banner-level parameters

Banner-level parameters are configured according to the retailer's needs during initial implementation. Some banner-level parameters are entered through the front end but are included in the table below for reference purposes. They reside on the following RCOM table:

- ORG_BANNER_PREFERENCE

As a reference, they are described below. See the RCOM Installation Guide for more information about how and in what order to enter them into the system.

| Parameter Name | Description | Values |
|---|---|---|
| Shipping method ID | This parameter, set up at the banner level, is used to determine which default shipping method is used. | Shipping Method Id from the SHP_SHIP_METHOD table |
| Event hold days | This parameter, set up at the banner level, is used in the calculation for the release date on an order line with an event hold. The Release Date = estimated customer delivery date (ECDD) – (event holds release time parameter + warehouse outbound handling days + item/location outbound handling days + service level delivery days). | Number (in days) |
| Credit card authorization lead days | This parameter, set up at the banner level, determines when to authorize/reauthorize a credit card payment. | Number (in days) |
| Default backorder delivery days | This parameter, set up at the banner level, specifies the number of days to be added to the date an order line is backordered (without a PO). The parameter facilitates the calculation of the ECDD. The attribute is set up and maintained at the concept level. | Number (in days) |
| Monogram fee and personalization fee | This parameter, set up at the banner level, is a flat rate price associated to monogramming and personalization. It is held at the concept level and is defaulted from RMS. | Number (in dollar amount) |
| Default cancel days | This parameter, set up at the banner level, specifies the number of days a pended order is held until it is systematically cancelled. | Number (in days) |

| Parameter Name | Description | Values |
|---|---|---|
| Personal hold delivery date limit | This parameter, set up at the banner level, is used to validate that the personal hold date requested by the customer is within 'n' number of days from today's date. | Number (in days) |
| Minimum amount for merchandise credit | This parameter, set up at the banner level, is used to determine whether a merchandise certificate or gift certificate needs to be issued. | Number (minimum dollar amount) |
| A sales audit system shipping and handling export SKU | This parameter, set up at the banner level, holds the item number value of shipping and handling (S&H) that is sent to a sales audit system. | Must be a real RMS generated SKU set up for each banner. |
| A sales audit system VAS monogram export SKU | This parameter, set up at the banner level, holds the item number value of VAS monogramming that is sent to a sales audit system. | Must be a real RMS generated SKU set up for each banner. |
| A sales audit system VAS gift card export SKU | This parameter, set up at the banner level, holds the item number value of VAS gift card that is sent to a sales audit system. | Must be a real RMS generated SKU set up for each banner. |
| A sales audit system VAS gift wrap export SKU | This parameter, set up at the banner level, holds the item number value of VAS gift wrap that is sent to a sales audit system. | Must be a real RMS generated SKU set up for each banner. |
| A sales audit system VAS personalization export SKU | This parameter, set up at the banner level, holds the item number value of VAS personalization that is sent to a sales audit system. | Must be a real RMS generated SKU set up for each banner. |
| Return threshold amount | This parameter, set up at the banner level, is used to determine whether a returned item needs to be returned. Any return item that is under this amount does not need to be returned. | Number (dollar amount) |
| Under payment amount tolerance | The amount an order can be underpaid by. This parameter along with the under payment percent tolerance parameter determines whether an under paid order can be released. | Number (dollar amount) |

| Parameter Name | Description | Values |
|---|---|---|
| Under payment percent tolerance | The percent of the order total an order can be underpaid by. This parameter along with the under payment amount tolerance parameter determines whether an underpaid order can be released. | Number (percent) |
| Backorder notification lead days | This parameter determines how many days before the customer original ECDD the notification should be sent of the new ECDD. | Number (in days, required, but can be 0) |
| Backorder notification delay days | This parameter determines how frequently a notification is sent. For example, if an ECDD only changes by a few days, a high parameter value established here would result in the system's not generating a notification. | Number (in days, required but can be 0) |
| A sales audit system VAS general export SKU | The item number used in the TITEM transaction in a sales audit system for a post-sale order header accommodation. This value must be a valid RMS item number. | Must be a real RMS generated SKU set up for each banner. |
| Merchandise credit as certificate | This parameter determines which type of merchandise credit should be issued.. | When the value is = 1, a merchandise certificate is issued. When the value is = 0, a merchandise card is issued. |
| Default return warehouse | This parameter determines which default warehouse should be used for an item being returned at the banner level. | From the LOC_WAREHOUSE table - WAREHOUSE_ID<br>This parameter is not required, but if it does not exist for a banner, the item is returned to the fulfilling warehouse. |
| Internet summary pend cancel days | This parameter, set up at the banner level, specifies the number of days that an internet pended order is held until it is systematically cancelled. | Number (in days) |

## The banner_channel packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's banner_channel component, see the following packages in the RCOM Javadoc:

- com.retek.component.banner_channel
- com.retek.component.banner_channel.integration.rib

## banner_channel RIB integration

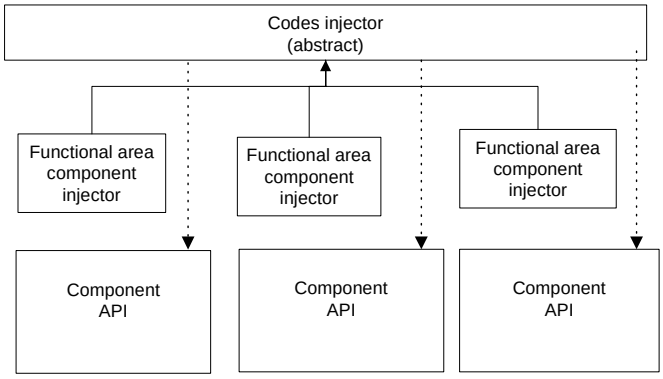
This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, 'Subscribers mapping table' and/or the 'Publisher's mapping table' in "Chapter 3 – RCOM and the Retek Integration Bus (RIB)". The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data's entry into the application's business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Codes component

## Processing overview

Code data that is published to the RIB stems from a single source in the merchandising system. Because the data is leveraged by multiple components within RCOM, the codes component acts as a common tool that can leveraged by multiple component APIs.

Within RCOM, code-related injector processing is slightly different than functional-area injector processing. The following flow diagram and its accompanying explanation provide a brief overview to this process.




**RCOM's code subscription process**

Within RCOM, the data within a code-related DTD is utilized by multiple functional area component injectors. The codes injector has thus been made abstract (so that it cannot act on its own).

In a given subscription scenario, a code.properties file maps the code types to the applicable functional area injectors. The functional area component injector extends the codes injector by implementing some key methods. The component injector is instructing the codes injector to inject the payload into the applicable component API(s). The component API is configured to act upon and/or validate the data.

## Codes processing summary

To see tables that describes RCOM's codes, code types, code descriptions, required indicators, and code sequences, see the RCOM Installation Guide.

## The codes package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's codes component, see the following package in the RCOM Javadoc:

- com.retek.component.codes

## Codes RIB integration

This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, ‘Subscribers mapping table’ and/or the ‘Publisher’s mapping table’ in “Chapter 3 – RCOM and the Retek Integration Bus (RIB)”. The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data’s entry into the application’s business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Correspondence component

## Functional overview

To enhance customer service, the correspondence component ensures that RCOM provides customer order and customer order line data intended for form letters and notifications to be sent out by a third-party system.

When certain systematic events occur in the system, a correspondence-related message is published to the RIB. The client is responsible for pulling the data from the RIB and integrating it to the third-party mailing system. The third-party mailing system ‘owns’ the templates (for email, for paper letters, and so on) into which the client places the order or order line data.

Based on the event, either the entire order’s data is published or order line data is published. If an order has multiple order lines, information for each order line is captured for the correspondence and published to the RIB. Each order/order line can have multiple correspondences associated to it.

In this version of RCOM, correspondence types that are systematically created are based on the following system events (triggers):

- First backorder notifications (order line information)
- Subsequent back order notifications (order line information)
- Mail order back order notifications (order line information)
- Order confirmations (entire order data)
- Return confirmations (order line information)
- Ship confirmations (order line data)

The list of template names is loaded into the RCOM schema via a database script.

Through administrative screens in RCOM’s front end, the client sets up the relationship among the systematic events, template names, and delivery methods (email or hard copy). Using internal data, RCOM determines whether a delivery method is inapplicable (for example, when a customer does not have an email address). The template name and delivery method gets published to the RIB along with the order/order line information.

## An overview of the correspondence process

The following steps provide an overview to the process when a systematic event, such as an order confirmation occurs:

1 The system determines the banner.

2 Based on the systematic event and system parameters, the system finds the applicable template name.

3 The system publishes the template name, delivery method, and order or order line data to the RIB.

4 The third-party system receives the data and uses it to determine which template and which fields to utilize in its merging and mailing process.

## The correspondence package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's correspondence component, see the following package in the RCOM Javadoc:

- com.retek.component.correspondence

## A note about correspondence-related batch processing

PublishCorrespondenceBatch is a batch process that is related to correspondence but is located within the Customer order component. See the RCOM Operations Guide for more information.

## A note about correspondence-related RIB integration

The correspondence-related publication is processed from within the Customer order component. This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, 'Subscribers mapping table' and/or the 'Publisher's mapping table' in "Chapter 3 – RCOM and the Retek Integration Bus (RIB)". The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data's entry into the application's business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Customer component

## Functional overview

The customer component is responsible for processing related to the following areas, among others:

- Activity requests
  An activity request initiates further investigation into an area related to an order or to a potential order. Customer information is attached to activity requests, which can be related to care cards, refund checks, product information, special orders, refunds and gift certificates, ‘where is my order’, and so on. RCOM allows for a selling item to be associated to an activity request.
- Catalog requests and catalog types
  A catalog request is user-defined. Existing and new customers may request catalogs. The customer may request catalogs without placing an order; moreover, the customer can be provided with an estimated customer delivery date for each catalog.
- Customer

  **Note:** Customers are global and are not tied to a banner. However, inside customers, data (credit cards, history, orders, and so on) exists that is or can be tied to a banner.

  This processing relates to general customer data. Some of the more important pieces of data include the following:

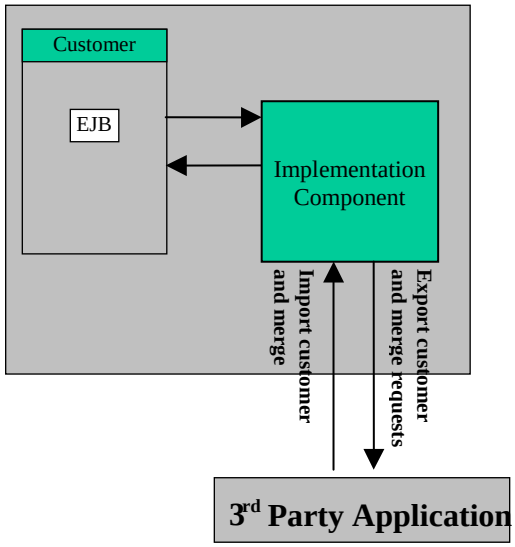
  - Credit card (note that the customer’s credit card is associated to a banner)

    **Note:** A customer can have 25 ship to addresses, 43 email addresses and 102 phone numbers (or more).

  - Addresses (primary bill to and primary ship to)
    - Active address (ensures that a customer address with the same address text as the address parameter exists in this customer's active address list)
  - Email addresses (active, primary, and so on)
    - Active email (ensures that a customer email with the same email address text as the email parameter exists in this customer's active email list)
  - Telephone numbers (primary day, inactive, and so on)
    - Active telephone (ensures that a customer telephone with the same number and extension as the telephone parameter exists in this customer's active telephone list)
    - Names (first, last, middle initial, and so on)
    - Initial banner (describes the banner through which the customer ‘came into’ the system; the data is used so that marketing can associate a customer to a banner)
    - Acquisition method (system-level data that describes how the customer ‘came into’ the system; for example, he or she requested a catalog, and so on)
  - Customer preferences (do not email, do not call, and so on) associated to the banner

- Match code functionality
  During new customer creation, match code functionality causes the system to query existing customers for records that might match a new customer. This functionality helps prevent the duplication of customer data within the system. Note that match code rules are configurable through the front end.

## Customer component's interface with a 3rd party customer-related application




**Customer component-3rd party interface**

RCOM has been designed to interact with an integration application that can call the customer integration API directly. Neither the RIB nor the subscribing classes (injectors) are involved in this interface.

The customer integration API acts as the interface between the customer application and the RCOM customer component. It sends the success and error codes back to the customer application. The API sends to the integration application either an exception for an error, or nothing when the process was a success.

The four existing export/import API's include:

- Export Customer
- Export Customer Merge Request
- Import Customer (create/update)
- Import Customer Merge (triggers actual merge of multiple customer records)

## Customer component batch processing

Java batch processing is associated with this component. For more information about batch processing, including file layouts, see the RCOM Operations Guide.

## The customer packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's customer component, see the following packages in the RCOM Javadoc:

- com.retek.component.customer
- com.retek.component.customer.integration
- com.retek.component.customer.integration.batch
- retek.component.customer.integration.xml

# Customer order component

## Functional overview

The customer order component manages the customer order throughout its lifecycle and provides the raw data necessary for historical reference (see the 'History component' section of this chapter). Because a customer can have multiple addresses, order lines, phone number, methods of payment and so on, this component ties the applicable information together for a given customer order. In other words, this component acts as the 'hub' for a customer order, updating it as it progresses.

Stripped of its complexity, the customer order process involves these steps:

1 A customer order header is created with a customer order number and a customer ID; multiple order lines are divided by ship tos.

2 The order lines are divided by the multiple ship to addresses, where applicable. RCOM processing identifies what is ready to ship and sends those ship requests to the warehouse management system (WMS).

3 From the WMS, RCOM receives ship confirmations for what has been shipped and how. RCOM processing updates the order as applicable.

Of course, processing becomes much more complicated with the addition of data related to taxes, accommodations, handling, payments, and so on.

Some of the more important data that the component associates to the customer order/customer order line includes the following:

- New or existing customer data (name and primary contact information)
- Source code data
- Bill-to address data
- Ship-to address data
- Because a customer order can have multiple ship-to addresses, order lines are separated by ship-to data.
- Carrier services
- Total and individual charges (order total, rush, merchandise, accommodation amount, service, personalization, taxes, and so on)
- Messages (invoice pack slip, shipping label, and so on)
- Warehouse instructions

- Order lines
  - Items
  - Order line type (up-sell, cross-sell, substitute, standard, partial, return)
  - Price and any additional charges
  - Shipping information
  - Estimated customer delivery date
  - Hold event data
- Value added service (VAS) lines data
  - Personalization
  - Monogramming services
  - Gift wrap data
  - Care card data
- Accommodation data
  Accommodations provide the customer with an ad hoc discount (in dollars or as a percentage).
  - Accommodation reason
- Discount/promotion data
- Container delivery confirmation data
- Type of payment data
  - Credit card
  - Merchandise voucher
  - Gift certificate
- Payment authorization response data (approved, authorized, cancelled, and so on)
- Payment history
  This data represents the payment history associated to an order. For example, a credit card could be settled multiple times during the course of an order. For each settled payment line, RCOM displays all of the items that are paid for within that payment settlement line.
- Settlement data
- Shipment confirmation data
  - Shipped container data
- Pended status reason code
- Value added service (VAS) data
- Gift certificate data
- Cancellation reason

- Return reason code
  These return reason codes originate in the merchandising system and can be used both in the WMS and within RCOM.
- Shipment request data

### Mail orders

The customer order component is responsible for processing related to mail orders. Mail orders are customer orders that have been mailed by a customer for processing. Mail orders typically consist of the completed order form (taken from insert within catalog) and the payment (tender) for the order.

Due to the need to manage the payments (tender) received with mail orders, additional management and reconciliation is required for these customer orders. To facilitate the tender reconciliation of these orders, batch mail order headers are created for a group of customer orders received by mail. Batches are assigned to specific business users. Batches are typically grouped into 50 orders.

All batches regardless of tender or type are controlled. Batches with a tender type other than credit card must be reconciled for the total physical tender amount received in the batch.

Because not all of the initial order information is available during order creation, mail orders undergo a different validation process than those customer orders which are created over the phone. The mail order validation process runs after orders have been entered in the system and confirms that the customer order has the required information to fully process the order. The validation process performs order functions such as the following:

- ATP
- Discounts
- Shipping and handling
- Tax calculations
- Payment authorizations (CC Authorization and AVS Fraud validation)

The alternative selling lists (upsell, cross sell, or substitutions) and scripting are disabled for mail order entry and validation. The validation process also checks the thresholds for over/under payments for the customer order.

Customer orders that pass the validations are released. Orders that fail validation are pended with a reason code explaining the issue.

## Functional reasons for RIB publication and subscription

### The customer order publication

RCOM contains publication functionality that ensures that, for each transmission of a ship request to the warehouse, informational data is included for all other items on the order, whether shipped or not. RCOM publishes items on the order that are ready to be picked, and marks those items 'live'. Using the same message, RCOM publishes the rest of the item data on the order and marks it with a different status ('shipped', 'reserved', and so on). For example, even if only one of ten items is sent to the warehouse for picking, a printed invoice shows the status of all ten items, thereby preventing customer confusion. All lines that are shipped to the same address and marked as 'process together' are flagged as 'pick together' for the WMS. This functionality may vary when packs are involved depending upon whether the reservation is at the pack level or at the component level.

RCOM communicates with the warehouse using not the customer order ID, but the ship request ID. Note that this process is behind the scenes and cannot be seen by the user.

At times, it may be necessary to communicate shipping and handling instructions to warehouse personnel, direct-ship suppliers, or delivery personnel. Such instructions are entered at the ship-to address level or at the ship-to address/order line level.

Delivery messages and warehouse messages are associated with a ship-to-address. 'Process together' requests, gifting services, and holds can be entered for one or more order lines destined for the same ship-to address.

### Replacement-related message

A replacement is a scenario in which a customer makes a request to return an item for another of the same item. Replacements are handled as inventory adjustments and a new cost of goods (COGS) adjustment. The processing surrounding replacement differs depending upon the business scenario.

1 Scenario: The item is returned resellable.

   A customer calls into replace an item that arrived in good condition but for some reason does not meet the customer's expectations.

   When the item is received, RCOM sends an inventory adjustment to RMS to decrement the inventory at the virtual store. RCOM then sends a COGS adjustment to RMS. This COGS adjustment increments the inventory and decrements the COGS at the virtual store for the item. RCOM creates a PAID IN transaction with reason code of 'Replacement In' to increase liability/contra sales and sends it to a sales audit system where it will be booked to a general ledger account.

   Once the WMS sends a ship confirmation to RCOM, RCOM does the following:

   - Decrements the reservation system and increments RMS with the reservation at the virtual warehouse that shipped the item.
   - Sends a RIB message to transfer the item from the virtual warehouse (decrement stock on hand and customer reservations at the virtual warehouse).

- Increments the stock on hand at the store. A COGS adjustment will also be sent to RMS. This COGS adjustment will decrement the inventory and increment the COGS at the virtual store for the item.
- RCOM creates a PAID OUT transaction with reason code 'Replacement Out' to decrease liability/contra sales and sends it to a sales audit system where it will be booked to a general ledger account.

2 Scenario: The item is returned damaged.

A customer calls into replace an item that arrived in damaged condition, perhaps from being broken in the warehouse, and so on.

When the item is received, RCOM creates a PAID IN transaction with reason code of 'Replacement In' to increase liability/contra sales and send it to a sales audit system where it will be booked to a general ledger account.

When the replacement item is shipped, RCOM decrements the reservation system and increments RMS with the reservation at the virtual warehouse that shipped the item, sends a RIB message to transfer the item from the virtual warehouse (decrement stock on hand and customer reservations at the virtual warehouse) and increments the stock on hand at the store. A COGS adjustment is also sent to RMS. This COGS adjustment decrements the inventory and increments the COGS at the virtual store for the item.

RCOM also creates a PAID OUT transaction with a reason code of 'Replacement Out' to decrease customer liability and sends it to a sales audit system where it will be booked to a general ledger account.

3 Scenario: The item does not need to be returned.

A customer calls into replace an item that arrived damaged. The item is not required to be returned either because the value of the item is below the return required tolerance, or the user determines the item does not need to be returned.

RCOM records no liability because was returned.

When the replacement item is shipped, RCOM performs all of the following:

- Decrements the reservation system and increments RMS with the reservation at the virtual warehouse that shipped the item,
- Sends a RIB message to transfer the item from the virtual warehouse (decrement stock on hand and customer reservations at the virtual warehouse)
- Increments the stock on hand at the store.
- Sends a COGS adjustment to RMS. This COGS adjustment decrements the inventory and increments the COGS at the virtual store for the item.

4 Scenario: The item is returned, and the replacement is fulfilled immediately.

The item is required to be returned because the value of the item is above the return required tolerance, and the replacement item is sent out immediately to appease the customer.

RCOM sends a pick request to WMS for the replacement item.

When the WMS sends a ship confirmation to RCOM, RCOM performs the same steps as in the first scenario; only the order is changed. The sale portion of the processing is likely to occur before the return portion of the processing.

5 Scenario: The item is returned , and a cancellation occurs for the replacement item.

A customer calls into replace an item that arrived damaged. The item is required to be returned because the value of the item is above the return required tolerance. After the item is received into the DC, the replacement item would normally be released for fulfillment. However, if the item is on backorder, it is held until inventory becomes available. During this time the customer contacts the call center and cancels the replacement item, which in turn, triggers a refund. The replacement item can be in a status other than backorder, but all existing cancellation validation/restrictions still apply.

The processing surrounding this scenario is highly complex because so much processing involves undoing replacement-related transactions. The central piece of this processing is that RCOM reverses out the replacement transactions that have occurred, and RCOM turns the processing into a normal return sale transaction.

### The customer order subscription

RCOM subscribes to messages from the RIB for ship, container, and distribution confirmation data. The messages originate in the warehouse management system.

After ship confirmation data is received, RCOM updates its order line information. Ship confirmation data can be received for order lines in backordered (only if part of the quantity of the order line is in fulfilling), fulfilling, and pre-cancelled status. If ship confirmation data is received for an entire quantity of the order line, the order line status is updated to ‘shipped’. The shipped quantity is updated to reflect quantity shipped. When all the order lines have received a ship confirmation and the status of all order lines has been updated to shipped or are in cancelled status, the status of the order header is updated to closed.

If ship confirmation data is received for part of the order line quantity (and no pick exceptions are received), the order line status remains in its current status. The shipped quantity field is updated to the quantity shipped.

The WMS informs RCOM as to what shipped and how many containers were used during the shipping process (these seven items went into one container; these three went into another container, and so on).

RCOM subscribes to stock order status messages related to pick exception information that are originally published by the WMS. These messages are published when the WMS is unable to perform fulfillment. Some of the data included in the notification includes the ship request number, item number, exception quantity and the exception status. When a pick exception message is received, the corresponding order line is found (based on the ship request number and the inventory item SKU number). The reserved quantity is decremented by the exception quantity, and the backorder quantity on the order line is incremented by the exception quantity. A call is made to the Reservation module to decrement the reservation and increment the backorder reservation for the pick exception quantity. Also, a pick exception event is written to the order's history.

The stock order status message subscription is also used to process cancellation confirmations (successful deletes). Once the message is consumed, RCOM moves cancellations out of a pending status.

RCOM subscribes to data from the RIB that facilitates estimated customer delivery date (ECDD) recalculation. RCOM stores the data on a staging table which is used in a batch process that recalculates the estimated customer delivery date for backordered lines.

## Overview of the shipment confirmation process

1 The customer order is created.

2 The order line is reserved.

3 The order is submitted.

4 Order fulfillment processing runs automatically for all items that are ready for fulfillment when the order is submitted.

5 The order status is open, and the order line status is fulfilling.

6 The item is shipped.

7 The ship confirmation is built.

8 RCOM receives the ship confirmation from the RIB.

Note that the following information is updated in RCOM for the ship confirmation (not all of the fields are required to have a return value):

- Ship confirmation order number
- Ship confirmation order line number
- Ship confirmation inventory item number
- Ship confirmation shipment number
- Ship confirmation carrier/carrier service level/zone
- Ship confirmation tracking number
- Shipped quantity
- Ship confirmation date
- Ship confirmation ship location
- Ship confirmation return location

- Ship confirmation RMA number

## Capturing demand status for each order line that is cancelled

The system has the ability to capture demand status for each order line that is cancelled from order entry or order maintenance.

Demand status for each order line that is cancelled from order entry or order maintenance is also captured in the order line audit tables. Every order line created is populated on the AUDIT_ORDER_LINE table with a demand status (when an order line has a demand status).

The demand statuses descriptions and codes include the following in their hierarchical order:

- No longer available (NLA)
  During order entry, if an item is placed on an order and ATP returns No Longer Available quantity, a new order line is created (behind the scenes) with a status of cancelled. The order quantity for the NLA quantity is returned from the ATP module with a cancel reason of No Longer Available and a demand status of No Longer Available. Order lines that are cancelled with a status of No Longer Available are never displayed in Order Entry or Order Maintenance. Instead, they contain an order line type of 'NLA'. This method is used to exclude the order line from the order line validation and exclude these types of order lines from being displayed to the user.
- Order entry error (ORDENTYERR)
  When an order line is cancelled and the cancel reason that is chosen by the customer service representative is Order Entry Error, the Demand status is set to Order Entry Error and persisted with the order line.
- Fraud (FRD)
  When an order is pended because it failed fraud validation, it will have a pended reason code of Fraud. If an order line is cancelled, the demand status is set to Fraud and persisted with the order line. If an order is cancelled, the system checks all of the pend reasons for the order (active and inactive) and determines if there are any fraud pend reasons. If so, the order lines are assigned a fraud demand status.
- Credit decline (CDCLN)
  When an item is placed on an order and is reserved, if any of the payments for this order have a status of Declined and the customer does not offer any other payments for the order, the customer service representative can cancel the order. Upon such an occurrence, all associated order lines have a demand status of Credit Decline.
- Substitute cancel (SUBCNCL)
  Sometimes, an order line is applied (regardless of order line status) and a sub item is offered. If the user accepts the sub item instead of the original item, the original order line is systematically canceled with a demand status of Cancel Substitute. The Reason code is also be systematically populated on the original line.
- Backorder abandonment (BKORDABDNT)
  In order entry, if an item that is backordered is cancelled, the demand status is set to Backorder Abandonment.
- Backorder cancel (BKORDCNCL)
  In order maintenance or order entry, if an item that is ordered is placed on backorder and then the customer decides to cancel the order line, regardless of the reason, the demand status is set to Backorder Cancel.

- Tag along (TGALNG)
  In order maintenance, if an order has multiple order lines and the first order line is cancelled with a reason of Backorder Cancel, the rest of the items that are cancelled after the first contain a reason of Tag Along. Processing does not have to occur within the same instance of order maintenance.
- Customer cancel (CUSTCNCL)
  During order entry if the customer decides to cancel an item that is reserved for a reason other than Order Entry Error, the demand status is set to Customer Cancel.

## Quantities: requested, ordered, and chargeable

### Requested quantity

Requested quantity is dependent upon the state of the order line. See the below table:

| Order line status | Requested quantity calculation |
|---|---|
| New | 0 |
| Reserved | Reserved qty + virtual warehouse reserved qty + shipped quantity |
| Virtual Warehouse Reserved | Virtual warehouse reserved quantity + shipped quantity |
| Backordered | Backordered quantity + reserved quantity + virtual warehouse reserved quantity + shipped quantity |
| Pre-Cancelled | Reserved quantity + virtual warehouse reserved quantity + fulfilling quantity + backordered quantity + cancelled quantity + shipped quantity |
| Cancelled | 0 |
| Fulfilling | Fulfilling quantity + shipped quantity |

### Ordered quantity

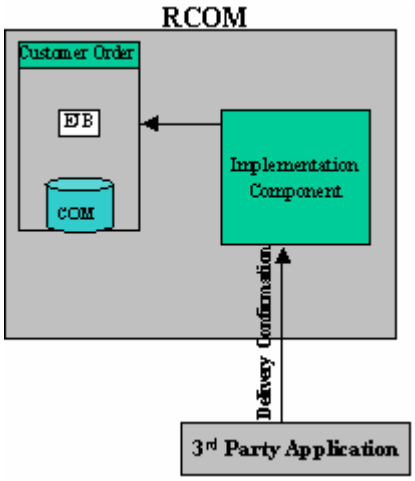
Ordered quantity always follows the following rules:

| Order Line Status | Requested quantity calculation |
|---|---|
| New | 0 |
| Pre-Cancelled | Shipped quantity + fulfilled quantity + cancelled quantity |
| All Others | Reserved quantity + shipped quantity + fulfilling quantity + backordered quantity + cancelled quantity + virtual warehouse reserved quantity |

### Chargeable quantity

The system uses the chargeable quantity to populate the original order line quantity in Order Maintenance. This quantity is all quantity minus cancelled quantity.

## Customer order component's interface with a 3rd party for delivery confirmation




RCOM must capture proof of delivery at the shipment level based on the delivery confirmation from the ship carrier. A delivery confirmation API provides an access point for 3rd party vendors to interface this information into RCOM.

Every container has a tracking number unique to the carrier that delivers it. RCOM accepts the carrier ID and tracking number in the delivery confirmation API. These fields allow RCOM to find the correct shipped container and update it with the delivery date/time.

ContainerDeliveryConfirmation is used for updating the delivery confirmation date on a shipped container from a 3rd party system. Every container has a tracking number unique to the carrier that delivers it. RCOM accepts the carrier ID and tracking number in the delivery confirmation API. These fields allow RCOM to find the correct shipped container and update it with the delivery date/time.

To read the ContainerDeliveryConfirmation and update the date against a tracking number, the following steps are performed:

**Note:** These steps are intended to be a guide rather than a direct procedure. Additional Java programming may be required to achieve desired results.

1 Get an instance of ContainerDeliveryConfirmationManager as following ContainerDeliveryConfirmationManager containerDlvryConfMgr = CustomerOrderManagerFactory.getContainerDeliveryConfirmationManager(getContext());

2 Read the Container by passing the proper carrierId and tracking number and is as follows ContainerDeliveryConfirmation foundContainer = ContainerDeliveryConfirmation foundContainer = containerDlvryConfMgr.findConfirmation(carrierId, carrierTrackingNumber);

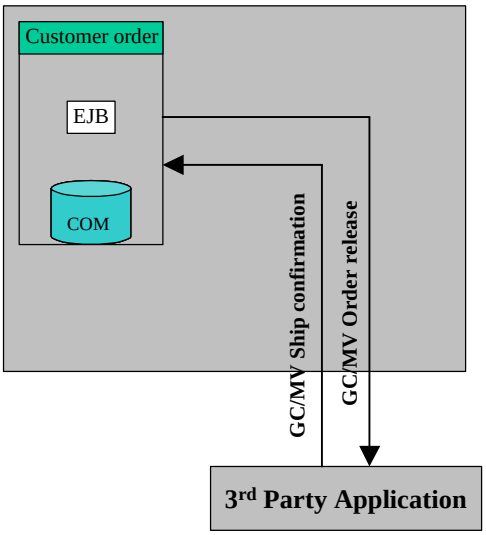
3 Create a delivery date of type RDate as following RDate deliveryDate = fill this with proper date and time of delivery.

4 Set this date on the container read in step 2 as follows foundContainer.setDeliveryConfirmationDate(deliveryDate);

5 Submit this change to the application as follows foundContainer.submit();

The process is a success if the Submit does not throw an exception.

## Customer order component's interface with a 3rd party for gift certificate fulfillment




### Gift certificate sales

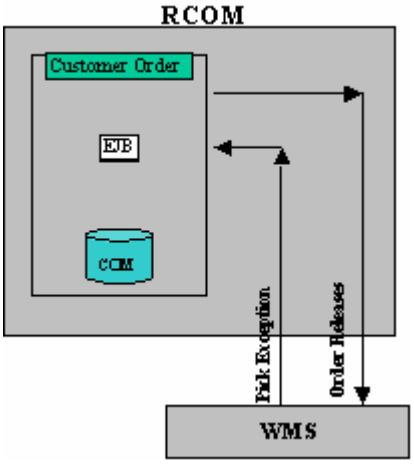
Customer orders are released through the RIB to the WMS. RCOM publishes the same RIB message as it publishes with a non-gift certificate order. If the order contains an order line(s) containing a gift certificate, then the RIB message indicates that a gift certificate is on this order line. The message is then picked up by the legacy system for gift certificate fulfillment. The WMS does not fulfill gift certificates.

A gift certificate item on an order generates a separate ship request from the rest of the non-gift certificate items.

### Ship confirmation of gift certificates

When a gift certificate has been shipped from the legacy system to the customer, the legacy system sends RCOM a ship confirmation message via the RIB. RCOM then finds all order line(s) associated to the shipment and updates the shipped quantity, order line status, and tracking information. The legacy system supplies RCOM with the voucher number and expiration date for the gift certificate. Only the voucher number is required.

## Customer order component's interface with a warehouse management system




### Customer order release

Customer orders are released through the RIB to the WMS. The message includes order priorities, which helps determine the fulfilling sequence when the order gets to the WMS.

### Pick exception

A pick exception from the WMS could occur when an order's item is one of the following:

- 'EX'–expired
- 'NI'–no inventory
- 'US'–update ship date (updates ship date from a direct ship supplier)

When the message is received into RCOM, the corresponding customer order line is found (based on the ship request number and the inventory item SKU number), the fulfilling quantity is decremented by the exception quantity and the backorder quantity on the order line is incremented by the exception quantity. The order line status is then set to 'Backordered'.

A call is also made to the Reservation module to decrement the reservation and increment the backorder reservation for the pick exception quantity.

### Cancel order line after release to WMS

When an order line has been released to the WMS and has a fulfilling quantity, it is possible to cancel the order line. The user will cancel the order line and the order line status will go to CANCEL REQUEST. RCOM sends a cancel order message to the WMS. If the WMS can successfully stop the item from getting shipped, the WMS sends a pick exception message back to RCOM with a status of 'SD' to represent a Successful Delete. RCOM then cancels the order line.

## The customer order packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's customer order component, see the following packages in the RCOM Javadoc:

- com.retek.component.customerorder
- com.retek.component.customerorder.batch
- com.retek.component.customerorder.integration.rib

## Customer order RIB integration

This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, ‘Subscribers mapping table’ and/or the ‘Publisher’s mapping table’ in “Chapter 3 – RCOM and the Retek Integration Bus (RIB)”. The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data’s entry into the application’s business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

## Customer order component batch processing

Java batch processing is associated with this component. For functional summaries about batch processing within RCOM, see the RCOM Operations Guide.

# Demand component

## Functional overview

The flash demand report displays hourly and historical demand data on a banner-level media by media basis. The demand component facilitates the processing necessary to ensure that flash demand report can be accessed from within the RCOM application.

The flash report provides information in two areas:

- Marketing: Summarizes demand at the order level
- Inventory: Summarizes demand at the order line level

Flash values are calculated hourly for a 24-hour period. Values are also rolled up to the day-to-date (DTD), week-to-date (WTD), and life-to-date (LTD) levels. Life to date refers to the life of the active or released media. Each day begins at midnight; each week begins on a predefined day. DTD refers to the previous day.

See the RCOM User Guide for definitions of the column values in the flash demand report (for example, # Orders, Total Order $, and so on).

Because the flash demand report must be available for all active and released media, a batch process runs on an hour to hour basis (the report will not be available on an ad hoc basis). For example, a user sees the same results at 1:15 p.m. as at 1:45 p.m., but at 2:00 p.m., the user sees new data.

A new day starts at midnight. At 12:00 a.m., the daily totals are reset to 0. At 1 a.m., the DTD totals are the values from 12 a.m. to 1 a.m.

A new week starts based upon a system parameter. The parameter holds numeric values. 1 = Sunday; 7 = Saturday.

Note the following:

- If an un-released order line is updated with increased quantity after the order has been initially persisted, the difference in the updated quantity is captured in flash demand.
- If an un-released order line is updated with decreased quantity after the order has been initially persisted, the difference in the updated quantity is not captured in flash demand.

## The demand packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's demand component, see the following packages in the RCOM Javadoc:

- com.retek.component.demand
- com.retek.component.demand.batch

## Demand component batch processing

Java batch processing is associated with this component. For functional summaries about batch processing within RCOM, see the RCOM Operations Guide.

# Direct ship order component

## Functional overview

Direct ship order lines are published upon the successful submission of a customer order using the same messages that are used for a warehouse-fulfilled item.

There is an additional indicator in the routing information for the RIB to use to route these order lines to a third party system (that is, to a supplier collaboration framework).

The third party has the responsibility of consolidating and of notifying the vendor of orders. Customer order lines that are put on event holds are also immediately published to the third party. Again, it is the responsibility of the third party to hold these orders and to release them to the vendor in time for them to be fulfilled.

The status updates associated with direct ship order lines follow the same process as that of warehouse-fulfilled items, except in the case of returns.

Direct ship supplier inventories are manually maintained in RCOM and are updated during the order creation, fulfillment, and shipment process.

The ship confirmation message is used to ship confirm both warehouse and direct-ship items. For a direct-ship, it is required that a cost for the item be provided. This data is used to record a receipt of goods at the virtual store that records the sale.

RCOM also publishes cancel requests to the third party, just as it does with regard to warehouse processing.

For more information about the direct-ship related publication process, see “Chapter 3 – RCOM and the Retek Integration Bus (RIB)”. The customer order component is responsible for publishing the direct-ship related data.

## The directshiporder packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM’s directshiporder component, see the following packages in the RCOM Javadoc:

- com.retek.component.directshiporder

# Event component

## Functional overview

Events (Christmas, Thanksgiving, and so on) play an important role in RCOM's functionality. 'Events' provide the ability to allow a user to place a customer hold on the order lines based on the event dates. For example, if an order line has an estimated customer delivery date of October 10th, and a 'Christmas' event hold is placed on it with the event customer delivery date of December 20th, the line's estimated customer delivery date is updated to reflect the new delivery date of December 20th.

Events can be created at the banner, supplier, and item/supplier levels in RMM. For example, if a Christmas event is set up, it can be associated to a banner, supplier or an item/supplier. In order to create an event at an item/supplier level, it has to first exist at the supplier level. Multiple events can be created at each level.

When a direct ship order line is selected, the system must first check to see if any event exceptions have been set up on the item/supplier level. If the item/supplier combination events are found, only those events are displayed. If an item exception event is not found, the system looks at the events set up for the supplier that is fulfilling the item. Only the events defined for this supplier are displayed. Finally, if no events for the supplier exist, the standard banner-level events are displayed. For all non-direct ship lines, only banner-level events are displayed.

# Geolocation component

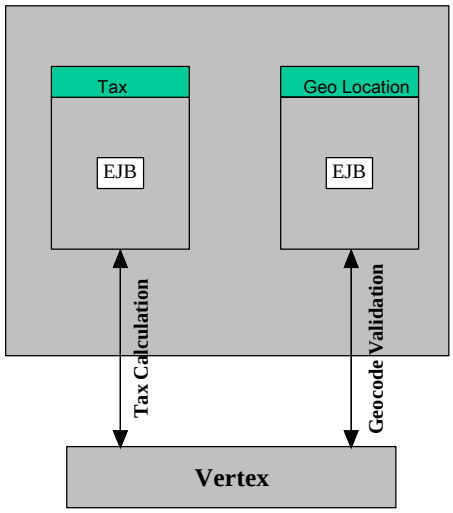
## Functional overview

Once RCOM passes city, state, county, zip code, and postal code data to the third party system (such as Vertex), it sends geo location data back to RCOM as a tax reference field. RCOM stores this tax reference code with the customer address so that when Vertex is called again to calculate taxes during order entry, the calculation is based on the geo location.

## Geolocation component's interface with a 3rd party tax application

The RCOM application uses Vertex as its 3rd party tax application. The diagram below indicates at a conceptual level the interface functionality between Vertex and the RCOM system.

In production, the database connection information held as a system parameter must be changed.




### Geocode

During order entry, the order entry system sends a postal code to the Vertex GEOCODE API. The Vertex GEOCODE API returns all the matching city/state/county combinations that are valid for that postal code. The matches are passed back into the order entry system. If an invalid postal code is sent, an error message is returned to RCOM. If the postal code cannot be validated by Vertex, the user has to manually capture the address from the customer, and it will not be validated at that time. It will be validated when the system becomes available.

## The geolocation package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's geolocation component, see the following package in the RCOM Javadoc:

- com.retek.component.geolocation

# History component

## Functional overview

Events are captured within RCOM and persisted within a history. When an order, or a customer, or an activity request, for example, undergoes a change, it is recorded in an applicable history. The type of information that RCOM captures for a history includes date/time, user, event type, event details, banner, and so on.

Historical information concerning orders and interactions with customers plays a critical role in facilitating enhanced customer service and in understanding the lifecycle of orders.

**Note:** Customer order payment history is processed in the customer order component. See the section, ‘Customer order component’ in this chapter.

### Customer order history

These events are captured as they occur and include the following, among others:

- Order holds
  Whenever an order is submitted and order lines have been placed on hold, a record is captured in order history.
- Create order
  When the order is created, a record is added to the order history.
- Cancel order
  If the entire order is canceled, a record is added to the order history.
- Cancel line item
  If an order line is cancelled, a record is added to the order history.
- Closed order
  When all order lines on an order are in closed (shipped or cancelled), a record is added to the order history.
- Return order line
  When an order line is returned, a record is added to order history.
- Replacement order line
  When an order line is replaced, a record is added to order history.
- Partial order line
  When a partial order line is created, a record is added to order history.
- Exchange order line
  When an exchange order line is created, a record is added to order history.
- Replacement order line
  When a replacement order line is created, a record is added to order history.
- Customer correspondence
  When an order or line is created, cancelled, returned or placed on backorder, a correspondence is created and sent to the customer, and a record is added to order history.

- Activity requests
  When activity requests are created and associated with an order, a record is added to order history.
- Order comments
  If the user creates an order comment, a record is added to order history.
- Customer accommodation
  When an accommodation is created for an order or an order line, a record is added to order history.
- Address update
  If an address is updated for an order, a record is added to order history.

### Customer history

These events are captured as they occur and include the following, among others:

- Catalog request
- Activity requests
- Submitted order data
- Associate comments
- Customer accommodation
  An associate entered an ad hoc discount to a customer order at the order level or the order line level.
- Customer created
  An associate entered a new customer.
- Customer preferences updated
  An associate updated the customer's preferred contact method and contact time.
- Customer updated
  An associate updated a customer's name, address, telephone number, or e-mail address.
- Manual release
  An associate investigated a pending order, resolved any issues, and released it.
- Order created
  An associate entered a customer order.

### Event history

RCOM receives historical information (such as history event ID, description, type [systematic verses external], and so on) from the following two sources:

- External system
  When an event comes from an external system, RCOM verifies that it is both an existing history event and that it is marked as ‘active’. If either of these proves to be false, RCOM logs an error, and no history event is captured. For example, the customer receives a promotional mailing from a 3rd party system. That system creates event history records for the customers that received the mailing and sends those to RCOM. RCOM validates that they are active, existing history events and places them on the applicable historical event table. This event type must be in the RCOM system for the system prior to receiving the event records via the API.
- Through the user interface (UI)
  The UI allows users to edit event type descriptions for all existing events (regardless of history event type – systematic verses external). This will be the only editable attribute for systematic history event types. For external history event types, users are able to mark them as inactive.

## The history package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM’s history component, see the following package in the RCOM Javadoc:

- com.retek.component.history

# Internet component

For information about the internet component, see “Chapter 6 – Internet/external APIs integration”.

# Inventory component (including the ATP module)

## Functional overview

Once the item is determined, the system follows these high-level business rules regarding its inventory status, that is, its ability to be fulfilled:

1 If the item can be warehouse fulfilled, the system utilizes the warehouse for fulfillment. To determine whether availability exists, the system uses the configurable ATP module, described below, which is responsible for the calculations in this step.

2 If the item cannot be warehouse fulfilled, the system determines whether the item is a direct ship item. If it is, the system checks the supplier quantity block to determine if the item is available.

3 If the item is not available in the supplier quantity block, the system checks to determine if the item can be backordered.

   a If the item can be backordered (an item's 'backorderability' is determined by specific rules), the system tries to determine the first possible day the item can be fulfilled by checking whether POs are incoming or in-transit quantities are incoming. These values, as applicable, are used to help calculate the estimated customer delivery date (ECDD).

   b If none of these values are incoming, the system uses today's date and adds a 'blanket' parameter to determine an estimated ship date. At order entry, the carrier service days are added.

4 For pack items, the pack inventory is checked first. If the requested quantity can be fulfilled, the pack is reserved. However, if enough inventory is not available, the component inventory is checked, and the pack is reserved at the component level. If a component is pick excepted, the component is rereserved and not the entire pack; this logic prevents the overshipping of goods to customers.

## The use of PO data

RCOM uses PO data in its available to promise (ATP) processing to determine future quantities. Thus, for backordered items, RCOM can provide approximate warehouse arrival dates that are based upon POs.

PO data is also used to reclass backordered items. If a PO changes, the system must reorganize POs and check dates to detect potential changes in estimated customer delivery dates (ECDD). When the estimated in stock date dates change for an item on associated POs, the system updates the ECDD. Based on parameters in the system, the system may also trigger the sending of a backorder notification to the customer.

## The available to promise (ATP) module

ATP module is responsible for determining inventory quantity-related calculations. For information about configuring the ATP module's values, see the RCOM Operations Guide.

ATP processing is integral to the process of taking an order. Behind the scenes, the ATP module comprises a major part of RCOM's inventory processing.

RCOM uses the ATP module for one of the following reasons:

- To place order line reservations and/or back orders.
- To release order lines because the merchandise has either been cancelled or shipped.

In other words, ATP facilitates the reservation of inventory, the reservation of a back order, or the release of a reservation or of a back order.

Within the ATP module, the Reservation module determines if it can reserve or backorder a customer order requested item quantity.

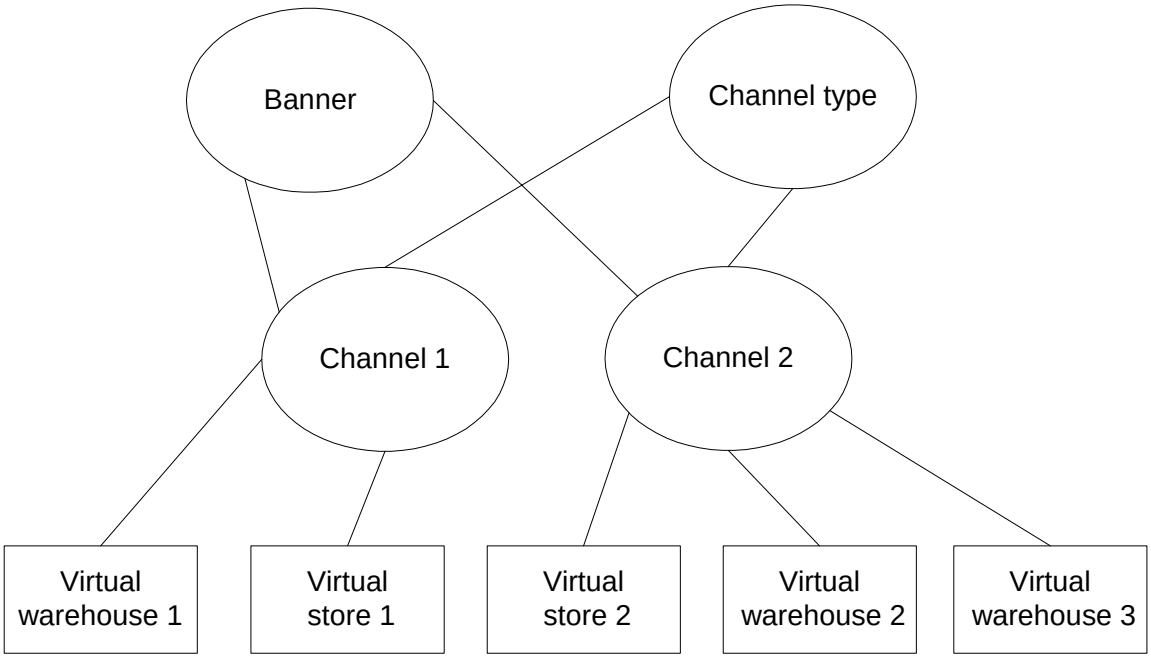
The available to promise (ATP) represents current inventory quantities at any time. ATP references data from the virtual warehouse level and rolls up available inventory to the concept and channel type to determine item availability. Note the following characteristics of the ATP module:

- The ATP-Reservation module enables RCOM to get the current inventory picture as well as the direct ship availability blocks.
- The ATP module prevents synchronization issues.
- The ATP module resides in the Inventory component.

### ATP in the context of the order process

When an order is taken in RCOM, the selling items are converted into inventory items and stored on the order. RCOM uses the inventory item to check ATP and perform the inventory reservation against.

Reservations are held at the banner-channel type level. When reservations are performed, the system knows the sum across the virtual stores that have the same banner-channel type. See the diagram below for a visual representation of the system's logic.




**ATP inventory concept at the banner and channel type level**

When payments are applied to an order in RCOM, the customer's credit card (if that is the form of payment used) is authorized for the amount of the order. The items are reserved in inventory and the order is then released from RCOM to be sent to the warehouse.

### Retail direct

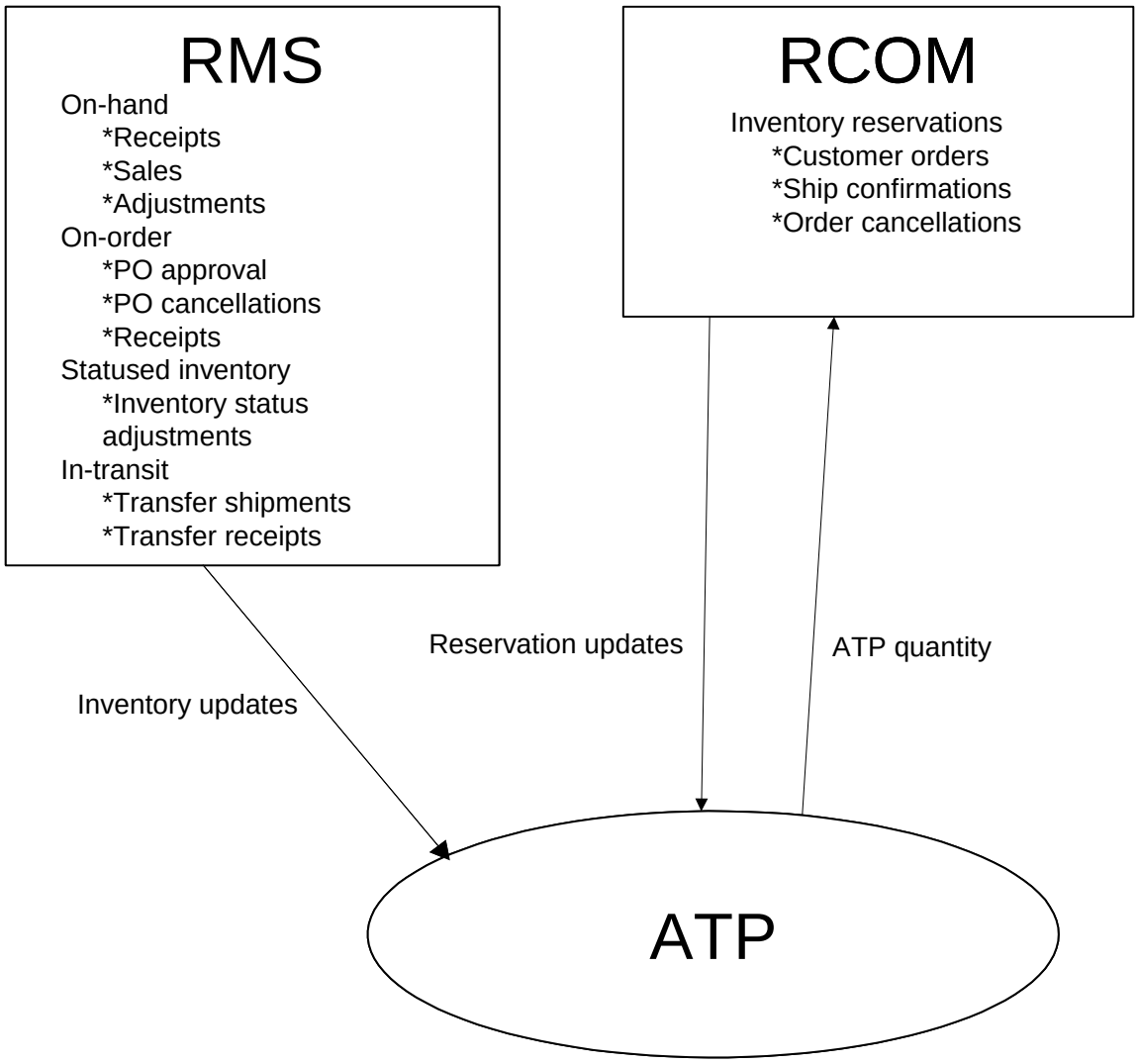
No matter where an order originates (whether from brick and morter, web, and so on), a system parameter in RCOM holds a default fulfillment channel type. This system parameter is used by ATP across the entire application. The value of this system parameter is a valid channel type.

ATP is based on all the channels that belong to the system parameter channel type and the current banner. This functionality allows all orders taken via RCOM to be fulfilled via valid warehouses for those channels.

### The ATP module and the merchandising system

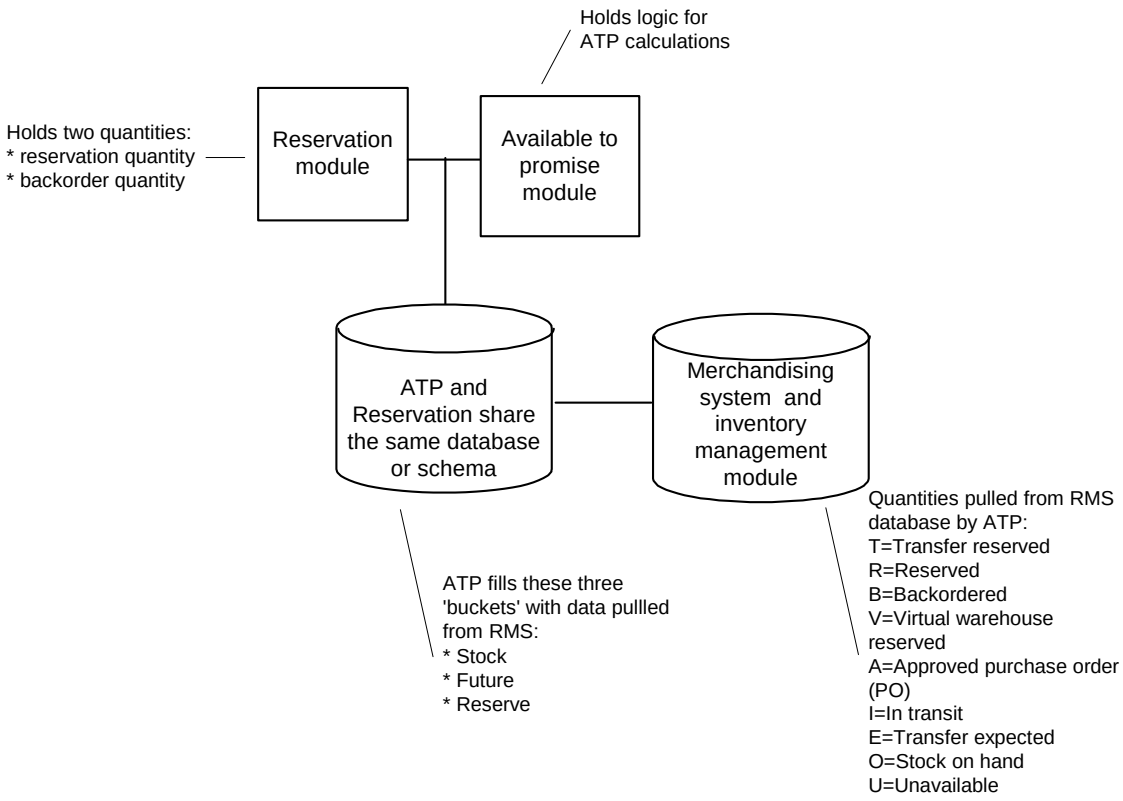
The following diagram provides a high-level view of what data passes among the merchandising system, the ATP module, and RCOM.

**Note:** Although the ATP module resides within the RCOM application, it is depicted here outside the application to better illustrate the logic of the ATP processing dataflow.




**High-level overview of the ATP module**

The ATP module is the only part of RCOM that accesses the merchandising system directly (rather than through the RIB) to get data. The module accesses the table ITEM_LOC_SOH and other inventory datasources for ATP. RCOM logs into the merchandising system behind the scenes.




**The ATP and Reservation modules and the merchandising system**

## The inventory interface

Within the inventory component, a generic object named InventoryDto holds the inventory quantities (the entire ‘picture’ of them) for the following:

- The sellable item-banner (inventory quantities are summed for all warehouses for all channels for this banner and system parameter channel-type)

A sellable item can be either a SKU or a pack.

The interface is InventoryDAO. Its method readInventory returns the inventory for for a banner based on the input parameter organization reference. That is, if a banner reference as the organization reference is passed in, the system returns the inventory of the sellable item at the banner level (actually banner/channel type level for all valid channel types for this banner).

If inventory quantities stem from a merchandising system other than RMS, a client would write a different implementation of InventoryDAO. A property file enables the substitution to occur.

Inventory at the banner level accounts for all of the channel types of that banner’s having that sellable item.

## Conversion of units of measure

Because different units of measure exist in the system, what is sold and what is stocked may represent two different units of measure. Processing within the inventory component converts the sellable unit of measure to the standard unit of measure when a reservation is made.

## The inventory package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's inventory component, see the following package in the RCOM Javadoc:

- com.retek.component.inventory

# Item component

## Functional overview

**Note**: Before an item can be successfully consumed by a subscribing application, the vendor, location, UDAs, diffs, channels, or banners that the item references must have already been successfully consumed by the same subscribing application.

RCOM subscribes to item-related messages from the RIB and stores the data within its system for its processing uses. The messages are published by the merchandising system. When RCOM components utilize item data, they leverage the data that has been persisted within RCOM. The only item data that is accessed directly in RMS is the quantity data that is handled by the ATP module within the Inventory component. See the section, 'Inventory component (including the ATP module)' in this chapter.

Valid values for item data must be kept in sync with the external system. All item data within RCOM is a reflection of the item data in the merchandising system.

A significant amount of business functionality within RCOM depends upon item subscriptions, including the following:

- The item's subscribing classes (injectors) facilitate the entry of item information from a merchandising system (such as RMS). The information should be imported from an external system and stored. This message is only used during the initial creation of an item and encompasses the item header message and possibly other messages relating to the item. The message serves as a 'wrapper' for the record so that all required information is there before the create message is sent. RCOM verifies that no more than one header is allowed for each item, and that an item has one header. RCOM subscribes to several item level, item-location level and item-supplier level attributes.
- Within the merchandising system, a default tax category has been added as an attribute on the class level of the merchandise hierarchy. Assigning a default tax category attribute at the class level allows the attribute to be inherited by the item during item setup. Items that are associated with the class inherit the tax category.
- RCOM subscribes to pack items. Packs are composed of one or more sellable or nonsellable components. The item pack data subscribing classes (injectors) facilitate the entry of item pack information from an external merchandising system (such as RMS). RCOM subscribes to this item pack information data and stores it with its association to the item. In RCOM, a pack number is a selling item SKU. Bill of material items (BOM) are items transformed to a finished good from a collection of sellable or nonsellable components. The components of a BOM are orderable items that can be either sellable or non-sellable. A pack item uses one item number to facilitate the buying and selling of both of the following:
  - Multiple units of a multiple component item
  - Multiple units of one component item

  Packs can be attached to a media, and they can be placed on order lines.

- RCOM subscribes to messages on the RIB related to the status of an item-location. The merchandising system publishes item-location data that illustrates whether an item-location is active, inactive, deleted, and so on. For example, in the summer, snow shovels at a given store might become 'inactive'. Within RCOM, the ATP module uses the status of an item-location to help it determine the item's availability. For more information about RCOM's ATP module, see the 'Inventory' section of this chapter.
- Four diffs can be associated with an item. Whenever RMS publishes item messages to the RIB, it can include all four diffs and their types. RCOM uses diff-related data to facilitate the ordering of items. When a selling item is entered into the system from the catalog, the system uses the information to return the diffs. A specific diff can then be selected that gives you the inventory item to be sold. In addition, searches can be performed using differentiator-related data.
- User defined attribute (UDA) injectors facilitate the entry of UDA header information from an external merchandising system (such as RMS). RCOM subscribes to this UDA data and stores it, associating it to the item. The attributes used for the product information functionality is derived from subscription-based item attributes/UDA data, or custom data entered directly into RCOM. Product information is used by the user to offer additional information to a customer about an item. Product information is held at the banner level and has a start and end date. There can be no more than one association between product information and item between the start and the end date.
- Item-supplier attribute related data subscribing classes (injectors) facilitate the entry of item-supplier level attributes from an external merchandising system (such as RMS). RCOM subscribes to these item-supplier level attributes and stores them with their association to the item. During the item definition, the suppliers from which the item are supplied are indicated. This information determines how the information is loaded into RCOM. Within the message, the primary direct ship supplier indicator informs the system that this supplier is the primary direct ship supplier for this item.
- The item-supplier-country attribute related data subscribing classes (injectors) facilitate the entry of item-supplier-country attributes from an external merchandising system (such as RMS). RCOM subscribes to these item-supplier-country attributes and stores them with their association to the item. The message allows RCOM to determine the cost to use when building a direct-ship purchase order.

## A note about the item levels that RCOM can receive

RCOM can receive the parents and the grandparents of items but cannot take the children of items. In other words, in its current version, RCOM can only take transaction level items and higher. In terms of the merchandise hierarchy, RCOM receives department, class, and subclass data through the item messages to which it subscribes.

## The item packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's item component, see the following packages in the RCOM Javadoc:

- com.retek.component.item
- com.retek.component.item.integration.rib

## Item component RIB integration

This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, ‘Subscribers mapping table’ and/or the ‘Publisher’s mapping table’ in “Chapter 3 – RCOM and the Retek Integration Bus (RIB)”. The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data’s entry into the application’s business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Location component

## Functional overview

Changes to store and warehouse data occur when organization features are added, modified or deleted in the merchandising system. RCOM receives these changes from the RIB.

Call centers are considered locations in RCOM. However, they are 'owned' by RCOM and do not pass through the RIB. For tax purposes, every user within the system is assigned to a call center. When an order is taken, the location of the person taking the order has repercussions in terms of tax calculations based upon tax jurisdictions.

Valid values for store must be kept in sync with the external merchandising system. Because items cannot be sold from the warehouse in the RMS, a 'virtual' store must be set up RCOM. A 'virtual store' simply refers to a store in the merchandising system that is set up as a non-stock keeping location.

Valid values for warehouse and the warehouse-attribute data must be kept in sync with the external system. All warehouse and warehouse attribute data within RCOM is a reflection of the warehouse and the warehouse-attribute data in the merchandising system.

Warehouse attributes data helps to regulate the types of merchandise that can be stocked at the warehouse. The definition of warehouses is divided into the following two entity types:

- Physical warehouses: The physical warehouses define the actual 'four-wall' warehousing, distribution and fulfillment facilities through which inventory is managed. The physical warehouse is a roll-up structure that holds one or more virtual warehouses. The physical warehouse attributes include the facility address, contact information and other facility level attributes. Transactions such as purchase orders and transfer orders that are sent to applications that do not use virtual warehouses are rolled up to the physical warehouse level (the vendor sees order quantities by physical warehouse).
- Virtual warehouse: A virtual warehouse represents the ownership divisions of inventory within the physical warehouses and the company as a whole. The virtual warehouse is defined within the maintenance of the physical warehouse and is assigned a Channel ID to classify the inventory ownership.

The organization functions provide data structures and screens to maintain the setup data within RCOM. Much of this information is defined in the merchandising system (such as RMS) and is integrated directly into the RCOM application. The information is primarily be setup data that is entered once when the application is installed and is most likely changed only when new organization features are added, changed, or deleted.

## The location packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's location component, see the following packages in the RCOM Javadoc:

- com.retek.component.location
- com.retek.component.location.integration.rib

## Location component RIB integration

**Note:** In a multi-channel environment, before a location can be successfully consumed by the subscribing application, any channels or banners that the location references must have already been successfully consumed by the same subscribing application.

This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, ‘Subscribers mapping table’ and/or the ‘Publisher’s mapping table’ in “Chapter 3 – RCOM and the Retek Integration Bus (RIB)”. The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data’s entry into the application’s business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Media component

## Functional overview

Processing within the media component allows for media setup within RCOM.

There are several attributes for the media, including the media type. A media type identifies the vehicle for communicating product offerings to customers. The media may be delivered in a print format, such as a catalog, or electronically, such as through an internet website.

A media may have different 'drops', subsets of a media. For example, a catalog might use a different cover, although the pricing structure within remains the same. A media may have more than one drop code.

Customers and prospects can be grouped by characteristics, such as demographics. This grouping takes place in a third-party marketing system. Source codes are used to uniquely identify such customer segments. Source codes are associated to drops, and drops are associated to the media. This data can lead to helpful analysis answering such questions as 'Which cover of the catalog inspired the customer to call?'

From the third party marketing system, RCOM imports the drop code and source code relationship. For example, drop one has these source codes associated to it; drop two has these source codes associated to it, and so on.

Once the media header is created based on this data, RCOM knows, for example, that this media has these four drops with source codes associated to each. RCOM generates a media code based the drop code-source code relationship data that it receives and sends that media code back to the third party system.

For each banner, consecutive media codes are generated. A channel is then associated to that media code. Thus, a banner's catalog and internet cannot have the same media code.

Shipping rate tables are associated to a media (for this range of money, the system charges this amount for shipping, and so on).

## Items and media

After the media header is created, an assortment of items can be added. A selling item number is generated for each item that you add to the media. The selling item number consists of the three-digit media code, a dash, and the parent item number or style number.

If one or more parent items or styles is combined, a dummy number is generated for the multi-style selling item. The media code precedes this dummy number also. The dummy number is generated based on a database sequence set up by a database administrator. Once the selling items are added to the media, one or more depictions can be defined for each selling item. Each selling item contains one or more selling SKUs.

Pricing is at the selling SKU level (for example, a large shirt could be priced less than an extra-large shirt). Items, with their associated differentiators, define a selling SKU and can thus be 'picked' to be represented and sold within a given catalog (orange sweaters for Halloween, red sweaters for Christmas, and so on).

Depictions can be set up, and a depiction code is comprised of a combination of the following:

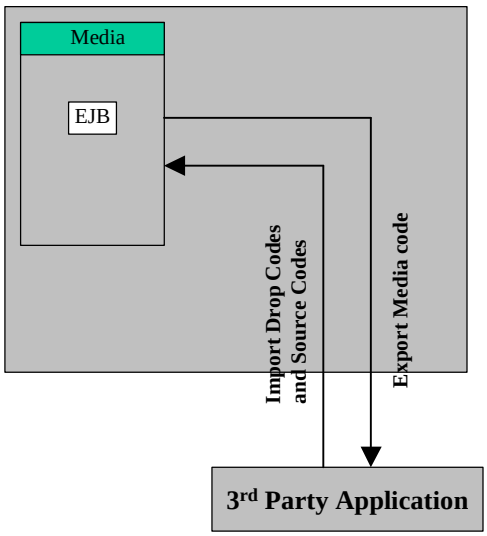
- Page spread: The page or spread of pages on which the depiction is placed.

- Key: An alphabetic or numeric code that is used to link the photograph or graphic with the copy text.

Other attributes include the % of page, the space cost, the placement, the total area of depiction and so on.

Value added services (VAS) can be associated at the inventory item level, so the consumer of the media can see that an item can be personalized (monogrammed, gift wrapped, and so on). Whether or not an item can be personalized or gift wrapped is determined by the merchandising system, but the media can show what is available. Additional setup must occur in RMM in order for the item personalization functionality to be available.

## Media component's interface with a 3rd party marketing application




From the third party marketing system, RCOM imports the drop code and source code relationship. For example, drop 1 has these source codes associated to it; drop 2 has these source codes associated to it, and so on. Once the media header is created based on this data, RCOM knows, for example, that this media has these 4 drops with source codes associated to each.

RCOM generates a media code based the drop code-source code relationship data that it receives and sends that media code back to the third party system.

## The media packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's media component, see the following packages in the RCOM Javadoc:

- com.retek.component.media
- com.retek.component.media.integration.rib
- com.retek.component.media.batch

## Media component batch processing

Java batch processing is associated with this component. For functional summaries about batch processing within RCOM, see the RCOM Operations Guide.

## Media component RIB integration

This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, ‘Subscribers mapping table’ and/or the ‘Publisher’s mapping table’ in “Chapter 3 – RCOM and the Retek Integration Bus (RIB)”. The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data’s entry into the application’s business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Message component

## Functional overview

The message component facilitates the functionality associated with the daily messages that are sent to specific call centers, corporate headquarters, or all locations. Although all associates may view messages, only those with the appropriate security level have permission to create, update, and delete daily messages.

By default, associates see the messages pertaining to their own location when they log on to RCOM.

The types of messages that appear in the message center may concern event notices, promotion reminders, and general information that management needs to convey to its associates.

Messages can have a start date and an end date, after which they are no longer visible.

## The message package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's message component, see the following package in the RCOM Javadoc:

- com.retek.component.message

# Payment component

## Functional overview

The payment component is responsible for processing related to payment types, credit card types, credit card validation attributes, and other payment-related data. Payment component processing addresses payment amounts, amounts that are authorized, authorization responses, payment in currencies, and settlements. The payment model is extended to include reward certificate payment functionality and stored value card (SVC) payment functionality.

RCOM processes tender types (credit card payments, gift certificate payments, merchandise certificates, and checks) according the logic that each requires. For example, when processing gift certificate payments, RCOM uses a gift certificate's control number but when handling a credit card payment, RCOM knows that a control number does not exist.

Tender type and tender type group codes and mappings are utilized in RCOM's export to a sales audit system because both systems must utilize common IDs. See the RCOM Operations Guide for more information.

The payment component also ensures that valid payments occur at different levels. For example, sometimes tender types are not valid for certain order sources (for example, a web-based store would not accept a check).

The credit card number rule is the logic that uses numeric prefixes and lengths to automatically determine a credit card type.

Once the payment vendor processes and matches the customer name and address information from RCOM, the payment vendor returns an address verification system (AVS) code. The payment component processes this data and stores it within the system.

The payment settlement process is used to communicate between the order entry application and the integration component. A batch process (located in the customer order component) runs to settle all credit card payments. The system will start the standard settlement process for orders once the item is ship confirmed. Settlements occur only for the payment amount that is equal to the sum of the merchandise amount, shipping costs, taxes and value added services of the item(s) being shipped. Each shipment will be settled once.

If an order is only partially shipped, the payments will be settled according to the payment settlement order of the payment types; for example, cash is settled before credit cards. The system will use the payment settlement order as defined by the banner/channel/payment type.

### Authorizations

The payment types currently supported by the RCOM application include the following:

- Gift certificates
- Gift cards (type of stored value card)
- Merchandise vouchers
- Merchandise cards (type of stored value card)
- Reward certificates
- Credit cards
- Checks
- Cash

For all payment types except credit cards, gift cards and merchandise cards a manual authorization within the application is currently required and the order status is therefore left in a status of pending. For credit card payments, the payment component uses the external authorization interface to authorize the payment. Gift cards and merchandise cards are authorized in real-time with a third-party via the authorization API.

The payment authorization API communicates synchronously. However, the API supports a request API and a response API for the integration layer. When the authorization request API is called, it makes a blocking call, forcing the UI to wait for a response from the response API. This processing allows the integration team to continue to build its interfaces.

The payment authorization API was modified to pass routing numbers and check numbers. The only required fields that need to be passed to the authorization API are tender type and payment amount. The system has the ability to capture and store check authorization information from the third party authorization application.

The integration API for credit card authorization is documented in the Javadoc for the package com.retek.component.payment.integration.creditcard.

### Settlements

The payment settlement process is used to communicate between the order entry application and the integration component. A batch process runs to settle credit card payments.

The batch process creates the flat file that is output from RCOM to the third party authorization application when settlement occurs. This settlement file is only created when the payment type is credit card. The batch process also publishes RIB messages for refunds.

The system starts the standard settlement process for orders once the item is ship confirmed. Settlements occur only for the payment amount that is equal to the sum of the merchandise amount, shipping costs, taxes and value added services of the item(s) being shipped. Each shipment is settled once.

If an order is only partially shipped, the payments are settled according to the payment settlement order of the payment types, that is, cash is settled before credit cards. The system uses the payment settlement order defined by the banner/channel/payment type.

Gift certificates and merchandise vouchers are settled upon ship confirmation but are not run through the batch process. When a ship confirmation is received, the system runs through the settlement order for the order's payments. If the gift certificate and/or merchandise voucher amount can be used to cover the ship container amount, the payment line is settled.

### Functional reasons for the settlement-related RIB publication

The settlement process can result in publication requests for check and merchandise voucher refunds. The message is RefdPayStlmtDesc.dtd.

The settlement (refund)-related publication is processed from within the Customer order component.

For more information about this publication processing, see the 'Customer order component' section of this chapter.

## Encryption strategy

RCOM facilitates the encryption of credit card numbers using an encryption strategy implementation that uses the Java Cryptography Standard Extenstion implementation of the DES algorithm to return base 64 encoded encrypted representations of credit card information.

## Payment component's interface with a 3rd party credit application system

RCOM sends credit application data to a third party system to allow it to set up a customer on a private label credit card (PLCC).

### Customer is already pre-approved

#### Scenario

This scenario outlines the most basic credit card application. An existing customer has been pre-approved for a credit card for the selected concept, and the scenario is triggered when an order is placed. The customer is informed of the pre-approval and asked if he or she would like to apply for the credit card. If the customer proceeds with the application, the third party application processes the application and returns an application status, account number and credit limit (if status is approved) for the new credit card.

### Customer is not already pre-approved

#### Scenario

This scenario outlines the situation when an existing customer has not yet been pre-approved for a credit card for the selected concept. This scenario is referred to as quick credit application, and the scenario is triggered manually by the user. The customer fills out the same application as in the 'Customer is already pre-approved' scenario but for the fact that the pre-approval code is blank. The third party system first checks to determine whether this customer is pre-approved for a credit card for the selected tender type. If the customer has *not* been pre-approved, the application process is returned with a status of 'Denied'. If the customer has been pre-approved, the application process proceeds as described in the 'Customer is already pre-approved' scenario.

The third party system returns an application status, account number and credit limit (if status is approved) for the new credit card.

### Interface process

1 The integration layer must define a class that implements the SynchronousCreditApplicationApplier interface (handles the applying of a single credit application).

2 The applyForCredit() method on this class is the entry point for triggering the credit application. Its only argument is an instance of CreditApplicationRequest which is defined as:

| Attribute name | Required? | Notes |
|---|---|---|
| ssn | Yes | Last 4 digits of customer social security number. |
| preApprovalCode | No | |
| dateOfBirth | Yes | |
| tenderTypeCode | Yes | |
| orderTotal | No | |
| firstName | Yes | |
| lastName | Yes | |
| middleInitial | No | |
| suffixCode | No | |
| addressLine1 | Yes | |
| addressLine2 | No | |
| addressLine3 | No | |
| city | Yes | |
| state | Yes | |
| postalCode | Yes | |
| postalCodePlusFour | No | Zip plus four |
| telephoneNumber | Yes | |

3 The applyForCredit() method should return an instance of CreditApplicationResponse which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| status | Yes | Whether application was approved or declined |
| accountNumber | No | |
| creditLimit | No | |

4 If any sort of system exception or fatal error occurs during credit SynchronousCreditApplicationApplier#applyForCredit, a CreditAppProcessingException should be thrown.

### Timeout or empty response from the third party system

#### Scenario

This scenario outlines a credit card application when the response from the third party system is 'timeout'. This is defined by one of these scenarios:

- The third party system never receives the request.
- The third party system receives the request, but does not respond within a certain time frame (defined in RCOM by a system parameter).
- The third party system receives the request, but returns an empty response.

In all of these cases, the status in the CreditApplicationResponse should be set to 'Reapply Later'.

## Payment component's interface with a 3rd party credit card authorization system

The Third Party Credit Card Authorization System revolves around two objects, The CreditCardAuthorizationRequest (CCREQ) bean and the CreditCardAuthorizationResponse (CCRESP) bean. RCOM sends a CCREQ to a registered SynchronousCreditCardAuthorizer via the SynchronousCreditCardAuthorizer.authorize method.

RCOM finds which SynchronousCreditCardAuthorizer to load and execute by looking at the file: com/retek/component/payment/integration/creditcard/integration.properties (rcom-rmm\build\conf\com\retek\component\payment\integration\creditcard\integration.properties) under the authorizer.synchronous key.

- SynchronousCreditCardAuthorizer.authorize(…)

| Attribute name | Required? | Notes |
|---|---|---|
| request | Yes | This CreditCardAuthorizationRequest Object holds all the information needed for the 3rd party system to extract and make an authorization request. (See CreditCardAuthorizatinRequest) |

If the third party integrators decide to validate that all needed information is on the request bean, they should throw a CreditCardProcessingException if required information is missing. Third party system should not return a declined CCRESP if pre validation fails.

The authorize(…) method should return an instance of CreditCardAuthorizationResponse which is defined as:

- CreditCardAuthorizationResponse

| Attribute Name | Required? | Notes |
|---|---|---|
| status | Yes | Either approved or declined. |

| Attribute Name | Required? | Notes |
|---|---|---|
| transactionId | Yes | This is the transaction number returned by Visa gateway system. |
| systemAuthorizationCode | Yes | This is the authorization code returned by visa. |
| avsCode | No | Please note that Rcom does not use this to determine whether or not the request was authorized. Rcom uses the status to verify whether or not a transaction is authorized. |
| authorizationDate | Yes | AVS code returned by visa. |
| cvvResponseCode | No | CVV Response code returned by Visa. |
| requestId | No | Id of the original CCREQ sent to Third party system by Rcom that caused this response. Will be used when asynchronous authorization is in place. |
| referenceField1-referenceField10 | No | These fields are here for future needs and can be used by the Settlement system. |

If any sort of system exception or fatal error occurs during SynchronousCreditCardAuthorizer.authorize(…), a CreditCardProcessingException should be thrown.

- CreditCardAuthorization Request Object

| Attribute Name | Required? | Notes |
|---|---|---|
| amount | Yes | The value to be authorized. |
| cardNumber | Yes | The card number to be authorized. |
| expirationMonth | Yes | The month the card expires. |
| expirationYear | Yes | The year the card expires. |
| cardVerificationValue | No | This is the card verification value entered by the User. |
| firstName | No | This is the first name on the orders primary customer. Note: This is required for address verification. |
| middleInit | No | This is the middle initial on the orders primary customer. |
| lastName | No | This is the last name on the orders primary customer. Note: This is required for address verification. |
| addressLine1 | No | This is the address line 1 of the orders primary bill to address. Note: This is required for address verification |

| Attribute Name | Required? | Notes |
|---|---|---|
| zipCode | No | This is the zip code (first 5 digits) of the orders primary bill to address. Note: This is required for address verification. |
| transactionDate | Yes | Day the credit card authorize request is generated. This is a standard Java Date object representing a snap shot of the system date on the server (includes time information). |
| requestId | No | This is some Rcom generated information that will be useful when synchronized authorized is implemented. |

## Reward certificate authorization processing

### Interface process

1 The integration layer must define a class that implements the SynchronousRewardCertificateAuthorizer interface (handles the authorization of a single reward certificate).

2 The authorize() method on this class is the entry point for triggering the reward certificate authorization. Its only argument is an instance of RewardCertificateAuthorizationRequest which is defined as:

| Attribute name | Required? | Notes |
|---|---|---|
| amount | Yes | The requested amount for the reward certificate |
| controlNumber | Yes | The reward certificate's account number |

3 The authorize() method should return an instance of RewardCertificateAuthorizationResponse which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| status | Yes | Indicates authorization was approved, invalid, expired, or redeemed |
| certificateValue | Yes | Reward certificate value |
| authorizationDate | Yes | Date the authorization attempt took place |

4 If any sort of system exception or fatal error occurs during SynchronousRewardCertificateAuthorizer.authorize(), a RewardCertificateProcessingException should be thrown.

**Note:** Retek has provided an implementation of this framework to authorize reward certificates against a sales audit system. See com.retek.component.payment.impl.remote.SyncronousRewardCertificateAuthroizerService for an implementation of SynchronousRewardCertificateAuthorizer

## Stored value card (SVC) integration

The integration layer must define a class that implements the SynchronousStoredValueCardProcessor interface (the API for all interaction with the stored value card system). When processing stored value cards (SVC), the system looks for com/retek/component/payment/integration/storedvaluecard/integration.properties (currently in the conf directory) in the class path and load the class named in the property storedvaluecard.synchronous.

The SynchronousStoredValueCardProcessor has four methods that are explained below:

- SynchronousStoredValueCardProcessor.authorize(…)

This method handles the authorization of a single stored value card. If the authorization is successful, the stored value card should be debited in the external system.

The only argument for this method is an instance of StoredValueCardAuthorizationRequest, which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| accountNumber | Yes | The stored value card's account number. |
| pinNumber | Yes | The stored value card's pin number. |
| requestedAmount | Yes | The requested amount to authorize. |

Both the accountNumber and pinNumber must match for the authorization to be valid. Failure to match the account number and pin number should result in an invalid response returned.

The authorize(…) method should return an instance of StoredValueCardAuthorizationResponse which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| authorizationDate | Yes | Date the authorization attempt took place. Populated even if the authorization fails. |
| authorizedAmount | Yes | The actual amount that was authorized for the stored value card. This value may be less then the requestedAmount. |
| availableBalance | Yes | Remaining card balance after the debit from the current transaction. |
| cardType | Yes | Either gift card or merchandise card |
| status | Yes | Either approved or invalid |

If any sort of system exception or fatal error occurs during SynchronousStoredValueCardProcessor.authorize(…), a StoredValueCardProcessingException should be thrown.

- SynchronousStoredValueCardProcessor.inquiry(…)

This method handles the inquiry of a single stored value card. No changes should be made in the stored value card system because of this call, it is strictly a "read" method.

The only argument for this method will be an instance of StoredValueCardInquiryRequest, which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| accountNumber | Yes | The stored value card's account number. |

The inquiry(…) method should return an instance of StoredValueCardInquiryResponse that is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| availableBalance | Yes | The stored value card's available balance. |
| bannerDisplayCode | Yes | The banner display code associated with the stored value card. |
| cardType | Yes | Either gift card or merchandise card |
| inquiryDate | Yes | Date the inquiry takes place. This is the one field that should be set even if a stored value card is not found for the requested account number. |

The attributes should be set to null (except inquiry date) if a stored value card is not found for the requested account number.

If any sort of system exception or fatal error occurs during SynchronousStoredValueCardProcessor.inquiry(…), a StoredValueCardProcessingException should be thrown.

- SynchronousStoredValueCardProcessor.cashout (…)

This method handles the cashout of a single stored value card. If the method call is successful, the entire balance should be removed from the card in the stored value card system. SynchronousStoredValueCardProcessor.cashout(…) is responsible for enforcing the business rule that a merchandise card cannot be cashed out. It is also responsible for checking the pin number supplied against the pin number of the card.

The only argument for this method will be an instance of StoredValueCardCashoutRequest, which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| accountNumber | Yes | The stored value card's account number. |
| pinNumber | Yes | The stored value card's pin number. |

The cashout(…) method should return an instance of StoredValueCardCashoutResponse that is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| cashoutAmount | Yes | The amount available to cashout (the stored value card's available balance) |
| cashoutDate | Yes | Date the cashout takes place. This is the one field that should be set even if a stored value card is not found for the requested account number. |
| bannerDisplayCode | Yes | The banner display code associated with the stored value card. |
| cardType | Yes | Either gift card or merchandise card |

The attributes should be set to null (except cashout date) if a stored value card is not found for the requested account number or if the card found is a merchandise card or if the card found does not have a pin number matching the supplied number.

If any sort of system exception or fatal error occurs during SynchronousStoredValueCardProcessor.cashout(…), a StoredValueCardProcessingException should be thrown.

- SynchronousStoredValueCardProcessor.voidAuthorization(…)

This method handles the process of voiding out an authorization made on a stored value card. If the method call is successful, the requested amount should be added back to the stored value card system.

The only argument for this method will be an instance of StoredValueCardAuthorizationRequest, which is defined as:

| Attribute Name | Required? | Notes |
|---|---|---|
| accountNumber | Yes | The stored value card's account number. |
| pinNumber | Yes | The stored value card's pin number. |
| requestedAmount | Yes | The requested amount to added back to the stored value card system. This amount will always be a positive number. |

Both the accountNumber and pinNumber need to match for this operation to be valid.

The voidAuthorization(…) method does not return a value. If the requested stored value card was not found (or an invalid pin number was supplied), an instance of StoredValueCardProcessingException should be thrown. The operation was successful if no exceptions were thrown from the processing of the method.

## Sample output settlement flat file from RCOM

This is a sample flat file that would be output from RCOM to the third party authorization application when settlement occurs. This settlement file layout definition .XML file defines the current implementation of the file specification.

```
<?xml version="1.0" ?>
- <fixed-width-format>
- <record-format bean="com.retek.component.payment.FheadRecord">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  <format>number</format>
  </field>
- <field start="16" end="29" required="Y">
  <property>fileCreateDate</property>
  <format>date,yyyyMMddHHmmss</format>
  </field>
- <field start="30" end="41" required="Y" justify="L" pad="">
  <property>settlementFileNumber</property>
  </field>
  </record-format>
- <record-format bean="com.retek.component.payment.FtailRecord">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  </field>
- <field start="16" end="25" required="Y" justify="R" pad="0">
  <property>fileRecordCount</property>
  </field>
  </record-format>
- <record-format bean="com.retek.component.payment.impl.TheadRecordImpl">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  </field>
```

```
- <field start="16" end="16" required="Y" justify="L" pad="">
  <property>transactionCode</property>
  </field>
- <field start="17" end="30" required="Y">
  <property>transactionDate</property>
  <format>date,yyyyMMddHHmmss</format>
  </field>
- <field start="31" end="42" required="Y" justify="L" pad="">
  <property>bannerCode</property>
  </field>
- <field start="43" end="48" required="Y" justify="L" pad="">
  <property>channelTypeCode</property>
  </field>
- <field start="49" end="78" required="N" justify="L" pad="">
  <property>orderNumber</property>
  </field>
- <field start="79" end="84" required="Y" justify="R" pad="0">
  <property>itemCount</property>
  </field>
  </record-format>
- <record-format bean="com.retek.component.payment.impl.TtailRecordImpl">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  </field>
- <field start="16" end="25" required="Y" justify="R" pad="0">
  <property>transactionRecordCount</property>
  </field>
  </record-format>
- <record-format bean="com.retek.component.payment.impl.TcustRecordImpl">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  </field>
- <field start="16" end="31" required="Y" justify="L" pad="">
  <property>customerId</property>
```

```
  </field>
- <field start="32" end="71" required="N" justify="L" pad="">
  <property>customerName</property>
  </field>
- <field start="72" end="111" required="N" justify="L" pad="">
  <property>address1</property>
  </field>
- <field start="112" end="151" required="N" justify="L" pad="">
  <property>address2</property>
  </field>
- <field start="152" end="191" required="N" justify="L" pad="">
  <property>address3</property>
  </field>
- <field start="192" end="221" required="N" justify="L" pad="">
  <property>city</property>
  </field>
- <field start="222" end="224" required="N" justify="L" pad="">
  <property>state</property>
  </field>
- <field start="225" end="234" required="N" justify="L" pad="">
  <property>zipCode</property>
  </field>
- <field start="235" end="237" required="N" justify="L" pad="">
  <property>countryCode</property>
  </field>
  </record-format>
- <record-format bean="com.retek.component.payment.impl.TpymtRecordImpl">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  </field>
- <field start="16" end="27" required="Y" justify="L" pad="">
  <property>paymentId</property>
  </field>
- <field start="28" end="39" required="Y" justify="L" pad="">
  <property>paymentTypeId</property>
  </field>
- <field start="40" end="79" required="Y" justify="L" pad="">
```

```
  <property>paymentTypeDescription</property>
  </field>
- <field start="80" end="95" required="Y" justify="L" pad="">
  <property>creditCardNumber</property>
  </field>
- <field start="96" end="115" required="N" justify="R" pad="0">
  <property>amount</property>
  <format>number,0.0000</format>
  </field>
- <field start="116" end="129" required="N">
  <property>dateOfPurchase</property>
  <format>date,yyyyMMddHHmmss</format>
  </field>
- <field start="130" end="141" required="N" justify="L" pad="">
  <property>authorizationCode</property>
  </field>
- <field start="142" end="155" required="N" justify="L" pad="">
  <property>expirationDate</property>
  <format>date,yyyyMMddHHmmss</format>
  </field>
- <field start="156" end="165" required="N" justify="L" pad="">
  <property>paymentPlanCode</property>
  </field>
- <field start="166" end="185" required="N" justify="L" pad="">
  <property>referenceField1</property>
  </field>
- <field start="186" end="205" required="N" justify="L" pad="">
  <property>referenceField2</property>
  </field>
- <field start="206" end="225" required="N" justify="L" pad="">
  <property>referenceField3</property>
  </field>
- <field start="226" end="245" required="N" justify="L" pad="">
  <property>referenceField4</property>
  </field>
- <field start="246" end="265" required="N" justify="L" pad="">
  <property>referenceField5</property>
  </field>
- <field start="266" end="285" required="N" justify="L" pad="">
  <property>referenceField6</property>
```

```
  </field>
- <field start="286" end="305" required="N" justify="L" pad="">
  <property>referenceField7</property>
  </field>
- <field start="306" end="325" required="N" justify="L" pad="">
  <property>referenceField8</property>
  </field>
- <field start="326" end="345" required="N" justify="L" pad="">
  <property>referenceField9</property>
  </field>
- <field start="346" end="365" required="N" justify="L" pad="">
  <property>referenceField10</property>
  </field>
- <field start="366" end="385" required="N" justify="R" pad="0">
  <property>originalAuthorizedAmount</property>
  <format>number,0.0000</format>
  </field>
  </record-format>
- <record-format bean="com.retek.component.payment.impl.TitemRecordImpl">
- <field start="1" end="5" required="Y" justify="L" pad="">
  <property>recordType</property>
  </field>
- <field start="6" end="15" required="Y" justify="R" pad="0">
  <property>fileLineNumber</property>
  </field>
- <field start="16" end="25" required="N" justify="L" pad="">
  <property>itemDepartment</property>
  </field>
- <field start="26" end="35" required="N" justify="L" pad="">
  <property>itemClass</property>
  </field>
  </record-format>
  </fixed-width-format>
```

## The payment packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's payment component, see the following packages in the RCOM Javadoc:

- com.retek.component.payment
- com.retek.component.payment.integration.creditcard

- com.retek.component.payment.integration.creditapp
- com.retek.component.payment.integration.rewardcertificate
- com.retek.component.payment.integration.storedvaluecard
- com.retek.component.payment.integration.xml

# Pend component

## Functional overview

To detect potential fraud, rules and parameters are established in RCOM at the banner level. Validation occurs when a customer order is submitted by an associate. If a customer order fails the validation check, it is automatically pended for further review. Pended customer orders can be reviewed, and a determination can be made whether to release or to cancel the order. The pend component facilitates processing that systematically validates customer orders or automatically pends them and sends them to the 'worklist' bucket. When a rule is created, it is assigned a task for routing purposes. See the section 'Task component' later in this chapter.

Pend rules are assigned a 'pend rule level'. This level dictates the following:

- Which pend rule criteria is valid when creating/editing the rule
- The level at which the rule is evaluated

A table within RCOM contains level/criteria relationships. The four levels are the following:

1 Order header

2 Order line

3 Return line

4 Accommodations

All areas that evaluate the pend rules upon an order submit evaluate rules at the correct level. For example, only pend rules created at the order header level are used to evaluate order header totals/attributes; only pend rules created at the order line level are used to evaluate order lines; and so on.

Each pend rule is treated as an OR condition. If a pend rule has multiple criteria, each criterion within the pend rule is treated as an AND condition.

The negative file contains records of customer, customer contact, and order-related information that were determined to be fraudulent. Records are added systematically when an associate cancels a customer order because of a specific reason code or selects the applicable add to file check box from the worklist.

A customer order may fail for one of the following reasons, among others:

- Payment type
  If the selected payment type is used to make a payment on a customer order, the customer order is pended.
- Department
  If one or more items are sourced from the selected department, the customer order is pended.
- Order line quantity
  If the number of units entered for one or more order lines is equal to or greater than the quantity entered, the customer order is pended.
- Order total tolerance
  If the grand total of a customer order is equal to or greater than the amount entered, the customer order is pended.
- High risk zip codes
  The zip codes on the bill-to and ship-to addresses are compared to predefined, high-risk zip codes. If there is a match, the customer order is pended.
- Negative file match
  The customer information is compared to predefined customer information in the negative file. If there is a match, the customer order is pended.
- Different bill-to and ship-to
  The customer order is pended if the bill-to and ship-to addresses differ in any way.
- Pend accommodations
  The customer order is pended if it contains any customer accommodations.

## The pend package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's fraud component, see the following package in the RCOM Javadoc:

- com.retek.component.pend

# Promotion component

## Functional overview

The promotion component facilitates the set up and the application of a general discount (50% off in percentage terms or $50.00 off in dollar terms, for example) that can be applied to order totals, order lines or to service totals. It also includes the setup and application threshold level discounts. Threshold level discounts incorporate purchasing a specific currency or quantity amount of an item or group of items in order to receive a discount off the purchase price, a discounted or free item, a discounted or free gift certificate, or a new PLCC plan code.

The general discount promotion is triggered by source codes, alternative selling lists or offer codes. There is only one source code associated to an order, and if that source code is associated to a promotion or promotions, the system applies the discount(s). (For more information about source codes, see the 'Media component' section of this chapter.) When an alternative selling list is triggered on an order (by either media code or selling item), and that alternative selling list is associated to a promotion or promotions, the system applies the discount(s). When an offer code is applied to an order, and if that offer code is associated to a promotion or promotions, the system applies the discount(s). The percent off or dollar off amount is established during the promotion setup.

If the promotion is set up to provide free or discounted services that are associated with the order header, the dollar off or percent off amount is applied to the service (such as shipping and handling) associated with the order. An order total or service total that is 'free' (such as free shipping and handling, for example) would be defined in the system as a 100% off general discount.

The threshold promotion is triggered by source codes, offer codes or tender types. If the order an order contains one of these triggers and meets specific criteria established during promotion setup, the system applies the discount(s).

Promotions are held at the banner/channel type level and occur within a valid date range.

## The promotion package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's promotion component, see the following package in the RCOM Javadoc:

- com.retek.component.promotion

# Security component

## Functional overview

The security framework within RCOM has been created to be responsible for the following two pieces of functionality, which are described in this section:

- Authenticating a user who is logging onto the system.
- Authorizing a user through the user interface (UI) to have access to specific business functions applicable to his or her role.

## The authentication of users

To facilitate the authentication of users, RCOM is integrated with a 3rd party directory service application: Microsoft's Active Directory®. Microsoft's website describes this product as having a "single-logon capability and a central repository for information for your entire infrastructure, vastly simplifying user and computer management and providing superior access to networked resources." RCOM's security solution is implemented with LDAP, which allows RCOM to 'talk' with Active Directory.

Data about users is stored in Active Directory. Such data could include, for example, username, password, location, supervisor, and so on. Active Directory does not contain mappings of users to roles or roles to permissions. RCOM provides the mappings between users and roles and roles and permissions.

 **Note:** RCOM never writes data to Active Directory.

An RCOM batch process runs and pulls new and/or modified user-related data from Active Directory and, after a validation step, persists the data within RCOM (thus ensuring that the two systems are in sync). The user's address information is the call center ID. API method calls verify that the call center that is imported from Active Directory is a valid call center in the RCOM system.

If the data is *not* valid, the user's data is not submitted to RCOM. Rather, the data is written to an output file, which can be specified as an argument in the command line when the batch process is run. The output file serves as a reference for errors, a log of 'bad' users. Before the batch process is run again, applicable corrections must be made to the 'bad' user's data in Active Directory.

### LDAP

LDAP stands for Light Directory Access Protocol. The LDAP standard defines a network protocol for accessing information in a directory. For additional information about LDAP, see the following website:

- http://www.openldap.org/
  This site contains the OpenLDAP main page. This site contains introduction, downloads, and documentation.

### Bootstrap user

The security framework includes a user that is does not originate in Active Directory. This user is not authenticated via the normal process. This user was created to engage in initial security functions and is intentionally prevented from being removed from the system (thus, this user is not visible from the front end).

### System user

Throughout RCOM's operation, the system performs automated tasks which are logged (in auditing tables, in history tables, and so on). System user is the name under which these automated tasks are logged.

## The authorization of role-based access for users

Security is used to grant permission to users to perform the tasks in RCOM and RMM that are required for their jobs.

One or more users must have permission to maintain security. That user, or security administrator, is responsible for setting up roles and assigning the appropriate roles to all other users. Roles are assigned to users, and permissions are assigned to roles.

Once the security administrator assigns one or more roles to a user, the user can access the application (RCOM and/or RMM) and perform the tasks for which permissions were assigned.

## Security component's interface with a 3rd party security-related system

RCOM uses Active Directory (AD) for two purposes:

- As the master repository of user information
- As a third-party authentication service

In the first case, all user information that is needed for RCOM is copied from AD into the RCOM repository via a batch process. Currently, the batch process only supports adding and updating users; there is no way to delete or inactivate users.

In the second case, RCOM authenticates users by connecting to AD as the user who is attempting to log in to RCOM. The user's password is never stored in RCOM; it is passed along when RCOM tries to connect to AD. If the connection to AD succeeds, then the user is considered authenticated in RCOM.

If RCOM cannot connect to AD; the user is not able to log in.

In both cases, RCOM connects to AD via LDAP. No Microsoft-specific enhancements are utilized.

## An overview of the security process

The following numbered steps illustrate the security process from a high-level perspective using a call center as an example. The steps illustrate a scenario in which a new user is added to Active Directory.

1 A new customer service representative (CSR) is hired.

2 The CSR user data is entered into Active Directory.

3 RCOM's batch process, SecurityUserUpdateBatch, is run, and the user's data is validated and persisted in the RCOM schema.

4 An RCOM security administrator assigns the user a role that provides access to applicable business processes.

5 The user can log on to the system, having undergone authentication and a role assignment.

## Security.properties

Security configurations for RCOM are located in the security.properties file.

Note that within the property file, a # sign that proceeds a value in the properties file signifies that what follows is a comment and is not being utilized as a setting.

### authentication.loginmodule

This setting specifies the process that is used to authenticate a user. For production purposes, this setting should always be set to the value below, which assumes the use of Active Directory:

```
authentication.loginmodule=com.retek.component.security.auth.LdapLog
inModule
```

### ldap.initialcontextfactory

This internal Java-specific setting should not change from its initial value.

### For LDAP authentication

These values are used for the configuration of the authentication process as it is run through LDAP. For example, in a production environment, the setting below would contain the client's address for its Active Directory:

```
# For LDAP authentication.
ldap.authenticationprovider.url=ldap://64.238.67.60:389/
```

### For batch user update

These values are used for the configuration of the authentication process with regard to the SecurityUserUpdateBatch process. For example, the setting below specifies where the system starts to look in the directory service for users:

```
ldap.batchuser.dn=cn=Administrator,cn=Users,dc=rcomad,dc=local
```

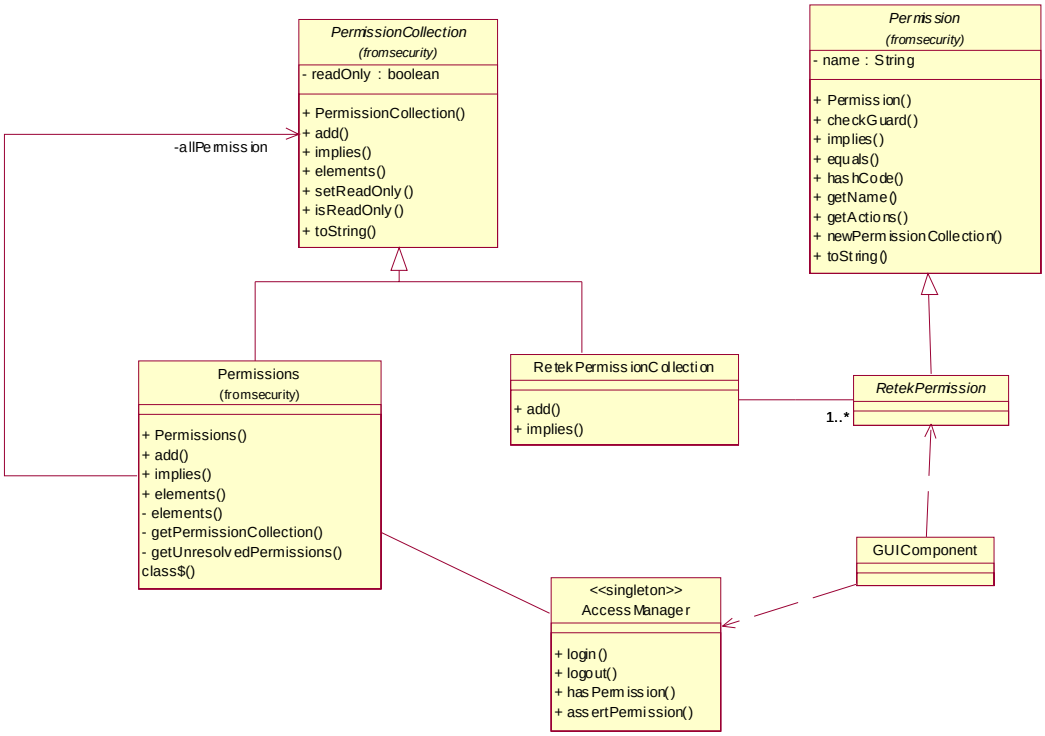
### For mapping LDAP to RCOM schema

These values are used for the configuration of Active Directory attribute names. For example:

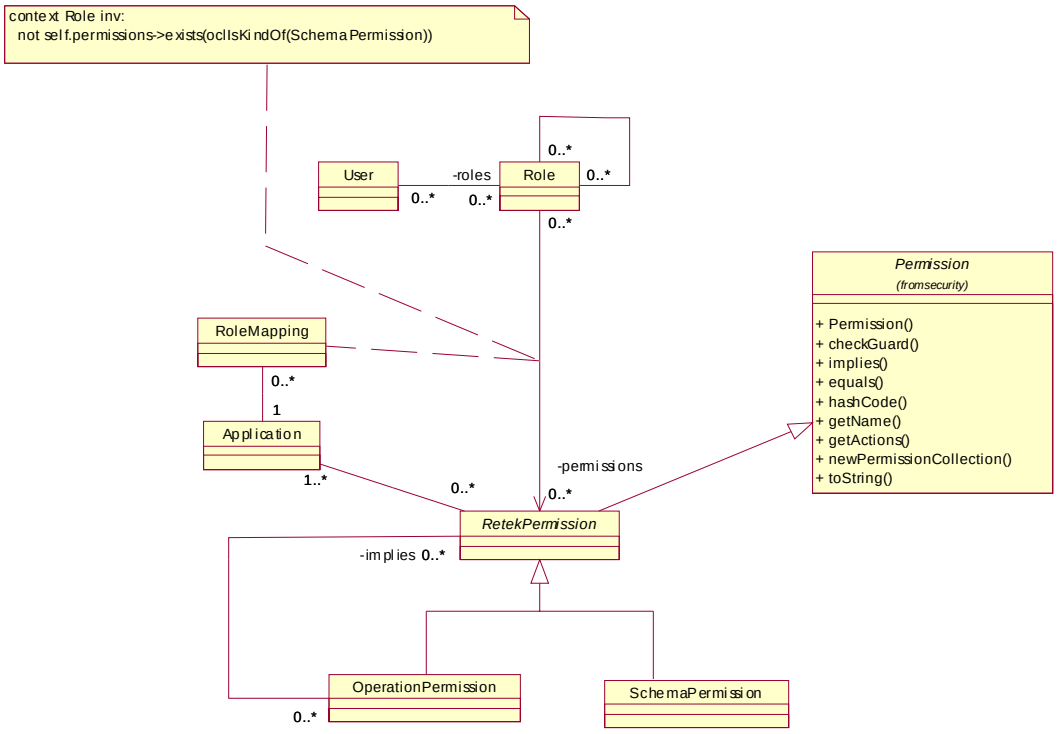
```
ldap.country.attrname=c
ldap.county.attrname=county
ldap.employeenum.attrname=employeeID
```

## Security model diagrams

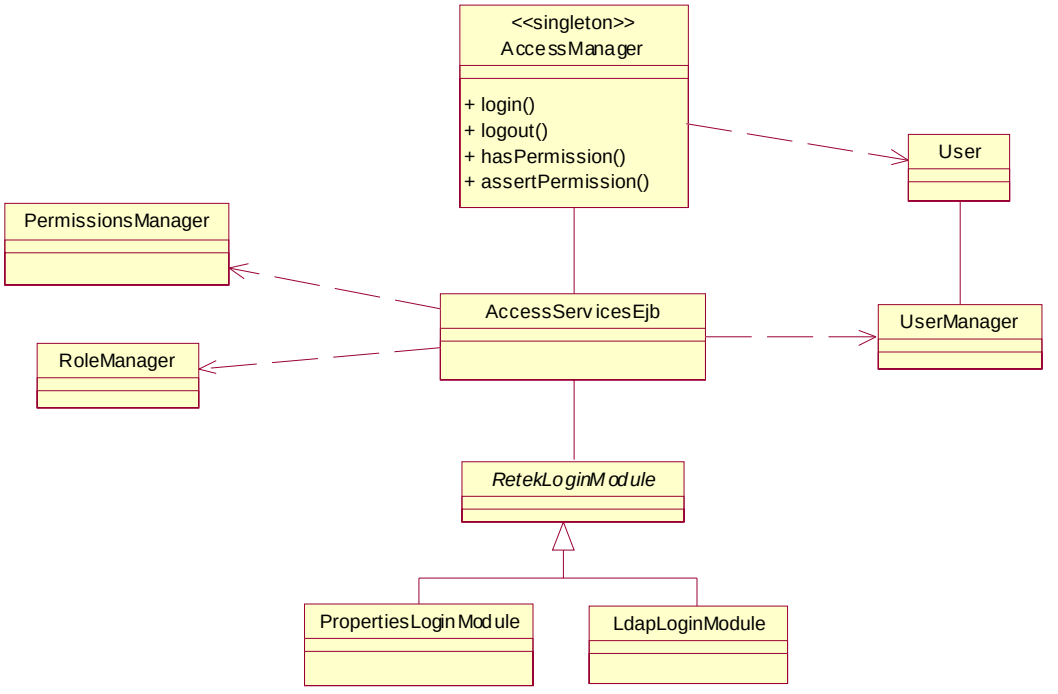
The following diagrams provide a high level view of the security model. They illustrate the conceptual logic behind security processing:




**Permission model**




**Permission mapping**




**Authentication model**

## The security package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's security component, see the following packages in the RCOM Javadoc:

- com.retek.component.security
- com.retek.component.security.auth
- com.retek.component.security.batch

## Security component batch processing

Java batch processing is associated with this component. For functional summaries about batch processing within RCOM, see the RCOM Operations Guide.

# Shipping component

## Functional overview

This component engages in processing related to carriers, shipping methods (carrier services), and ship restriction data, all of which are imported into RCOM. This component associates the carriers (UPS, Federal Express, and so on) to the ship methods (overnight, next day, and so on). Items that are set up in RMM include a default shipping method.

Some of the processing within this component includes the following:

- Adding and/or removing a specified carrier to the list of carriers that provide a shipping method.
- Setting/returning a list of carriers that provide a shipping method.
- Setting/returning a default carrier for a shipping method.
- Setting/returning the number of days it takes to deliver an item via a shipping method. These values, as applicable, are used in conjunction with the estimated ship date, which is the result of ATP calculations. To arrive at an estimated customer delivery date (ECDD), processing elsewhere in the system adds the ATP-generated estimated ship date to the shipping-generated delivery days data.
- Setting/returning a description of a shipping method.
- Getting/returning a shipment tracking carrier website URL for a given shipment tracking number (from a shipment container).

The following attributes are held at the carrier level:

- Carrier (existing and non editable)
- Valid carrier(s) used for return pickups
- Default carrier to be used for pickups
- Account number used for the billing of pickups for that carrier
- Tracking URL (existing)

RCOM subscribes to the RIB to retrieve the carriers and shipping methods (carrier services). See the ‘Codes component’ section of this chapter.

## The shipping package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's shipping component, see the following package in the RCOM Javadoc:

- com.retek.component.shipping
- com.retek.component.shipping.integration.rib

## Shipping component RIB integration

The shipping-related subscription is processed from within the Codes component. For information about this component and the RIB, see the sections, 'Subscribers mapping table' and/or the 'Publisher's mapping table' in "Chapter 3 – RCOM and the Retek Integration Bus (RIB)". The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data's entry into the application's business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# Supplier component

## Functional overview

Supplier and supplier address-related data is imported from the RIB and originates within the merchandising system. Valid values for the supplier and supplier address data must be kept in sync with the external system. All supplier-related data within RCOM is a reflection of the supplier-related data in the merchandising system. The message serves as a 'wrapper' for the record so that all required information is there before the create message is sent.

Processing within this component facilitates the following business processes.

- Personalization service: RCOM processes this data because personalization differs among suppliers.
- Monogramming service: RCOM processes this data because personalization differs among suppliers.
- Direct ship order addresses are used for tax calculations during ship confirmation.
- The supplier font and color descriptions originate from an external supplier's system. The supplier component uses this data for personalization and monogramming. The data includes applicable ID values. For the color Red, for example, supplier A red might use a value of 'R', and supplier B might use a value of 'Red'.

## The supplier packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's supplier component, see the following packages in the RCOM Javadoc:

- com.retek.component.supplier
- com.retek.component.supplier.integration.rib

## Supplier component RIB integration

This component is involved in RIB-related processing. For information about this component and the RIB, see the sections, 'Subscribers mapping table' and/or the 'Publisher's mapping table' in "Chapter 3 – RCOM and the Retek Integration Bus (RIB)". The publishers table illustrates the relationship among the message family, the message type, and the DTD/payload object. The subscribers table includes the message family and message type name, the document type definition (DTD) that describes the XML message, the component, and the subscribing classes that facilitate the data's entry into the application's business object layer. For additional information, see the latest Retek Integration Guide and other RIB documentation.

# System parameter component (including system parameters)

## Functional overview

Every system parameter has a pre-established (default) value.

System-level parameters are configured according to the client's needs during initial implementation. They reside on the following RCOM table:

- SYS_SYSTEM_PARAMETER

As a reference, they are described below. See the RCOM Installation Guide for more information about how and in what order to enter them into the system.

| Parameter Name | Description | Values |
|---|---|---|
| Point of Presence -- tax.pointOfPresence | This parameter is used to determine which organization addresses are used to determine Point of Presence for Tax calculations. Set up as a system parameter. | Values are 'Banner' or 'Global', Parameter Type = 'T' |
| System Parameters to connect to the tax service<br>Vertex.databaseUrl<br>Vertex.databaseUser<br>Vertex.databasePassword | These variables are used only when connecting to Vertex. Set up as system parameters. | URL – Holds the connection to a 3rd party tax application (Vertex)<br>USER – Holds the user name used to enter into a 3rd party tax application (Vertex)<br>Password – Holds the password to enter into a 3rd party tax application (Vertex)<br>Parameter Type = 'X' |
| In Transit Day Factor -- atp.inTransitDayFactor | This parameter specifies the number of days to be added to the date an order line is backordered (due to in-transit inventory), in order to calculate the Estimated Customer Delivery Date. Set up as a system parameter. | Number (in days), Parameter Type = 'A' |

| Parameter Name | Description | Values |
|---|---|---|
| System Parameters for Hotkeys | Ability to use short cut codes when entering text on the personalization and ship-to processing secondary tabs. (User types SHIFT + ALT + one of the codes). Set up as a system parameter. | Values:<br>HB – Happy Birthday<br>MC – Merry Christmas<br>HA – Happy Anniversary<br>HH – Happy Hanukkah<br>HW – Happy Halloween<br>HM – Happy Mother's Day<br>HF – Happy Father's Day<br>NP – No Packing Peanuts<br>BD – Beware of Dog<br>LN – Leave with Neighbor<br>Parameter Type = 'V' |
| Match Code System Parameter - matchCode.isFunctionalityActive | This parameter determines when match code functionality is active or inactive. Set up as a system parameter. | Values are 'true or 'false', Parameter Type = 'M' |
| Shipping & Handling Tax Code -- tax.shippingAndHandlingTaxCode | This parameter is used to identify tax codes for product codes. Set up as a system parameter. | Number, Parameter Type = 'T' |
| Gifting Tax Code -- tax.giftingTaxCode | This parameter is used to identify tax codes for product codes. Set up as a system parameter. | Number, Parameter Type = 'T' |
| Personalization Tax Code -- tax.personalizationTaxCode | This parameter is used to identify tax codes for product codes. Set up as a system parameter. | Number, Parameter Type = 'T' |
| Fraud Cancel Reason -- fraud.cancelReason | This parameter is used to hold the value(s) of the fraud cancel reason that generates a negative fraud file. The parameter can hold multiple cancel reason ids. Set up as a system parameter. | Number, Parameter Type = 'F' |
| Flash Demand Week Start Day system.flashReportWeekResetDaykey | For flash demand reporting, a new week will begin on the specified day. The parameter will hold numeric values. 1 = Sunday, 7= Saturday | Parameter Type = 'S' |

| Parameter Name | Description | Values |
|---|---|---|
| Time Out In Seconds<br>system.parameterTimeoutInSeconds | The number of seconds the application will cache the system parameter values in memory before going back to the database to read the values again. Number (in seconds) | Parameter Type = 'S' |
| Backorder Pending Hours<br>system.backorderPendingHoursKey | The number of hours a backorder notification will be held in the pending file before it is published to the RIB. This will give users the ability to go in and cancel a notification if it is deemed no longer needed to be sent. - Number (in hours) | Parameter Type = 'S' |
| Return Disposition Code for Disposed Inventory<br>system.defaultReturnDispositionCode | This parameter is used to denote the return disposition code that results in disposed inventory transactions. | Parameter Type = 'S' |
| Pick Not After Days --<br>System.pickNotAfter | The pick_not_after_date field is required for the RIB message, but RCOM items do not have this kind of information. This parameter will be added onto the transaction date in the ship request message to populate the pick_not_after_date on the message. The parameter's value is in days. | Parameter Type = 'S' |
| Return Tax Credit Reason --<br>tax.returnTaxCreditReasonCode | This code represents why a tax credit is being issued in the case of returns. | Parameter Type = 'T' |
| Accommodation Tax Credit Reason --<br>tax.accommodationTaxCreditReasonCode | This code represents why a tax credit is being issued in the case of a general accommodation. | Parameter Type = 'T' |
| Tax Accommodation Tax Credit Reason --<br>tax.taxAccommodationTaxCreditReasonCode | This code represents why a tax credit is being issued in the case of a tax accommodation. | Parameter Type = 'T' |

| Parameter Name | Description | Values |
|---|---|---|
| Sunday Contact Day -- system.sundayContactDay | This value informs the system what day is Sunday. To guarantee synchronicity between systems, this day must be identical to that set up in the merchandising system (such as RMS). | Parameter Type = 'S' |
| Default Client Browser Path -- system.defaultClientBrowserPath | This parameter represents the default path to a browser that resides on the client system. The browser is used in conjunction with shipment tracking functionality. | Parameter Type = 'S' |
| ATP Fulfillment Channel Type -- atp.fulfillmentChannelTypeMod | This parameter must be a channel type used for ATP inventory lookup. | Parameter Type = 'A' |
| Max Days Compare Dups - - salesAudit.maxDaysCompareDups | This is the number of days before the transaction numbers will be reset to 1.<br>To guarantee synchronicity between systems, this day must be identical to that set up in the merchandising system (such as RMS). | Parameter Value = 1<br>Parameter Type = U |
| Default Destination ID -- system.defaultDestinationId | The parameter represents the default destination in Warehouse management system for customer orders. | Parameter Value = Retailer defined<br>Parameter Type = S |

| Parameter Name | Description | Values |
|---|---|---|
| Additional Field Editable -- system.additionalFieldsEditable | This parameter determines if the user will have access to the following RMM tabs:<br>Inventory Item<br>Item Supplier<br>Item Supplier Personalization<br>Item Location<br>Events<br>Maintain Events<br>Banner Events<br>Supplier Events<br>Item Events<br>Admin<br>Organization<br>Supplier<br>Warehouse<br>Ship Restriction Codes | Parameter Values = true, or false<br>Parameter Type = S |
| Last Transaction Reset Date - - salesAudit.lastTransactionResetDate | Internal date used for sales audit batch process. No initial setup is required. | Parameter Value = “MM-DD-YYYY”<br>Parameter Type = S |
| Unknown RMA Number Code -- system.unknownRmaNumberCode | In return confirmation, if this value is encountered in the RMA number field in the CustRetDesc payload, then triggers the unknown RMA processing. | Values = "UNKNOWN" or the value that is placed in the RMA field from the Warehouse Management System in the case of no RMA. |

## The system parameter package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's system parameter component, see the following package in the RCOM Javadoc:

- com.retek.component.systemparameter

# Task component

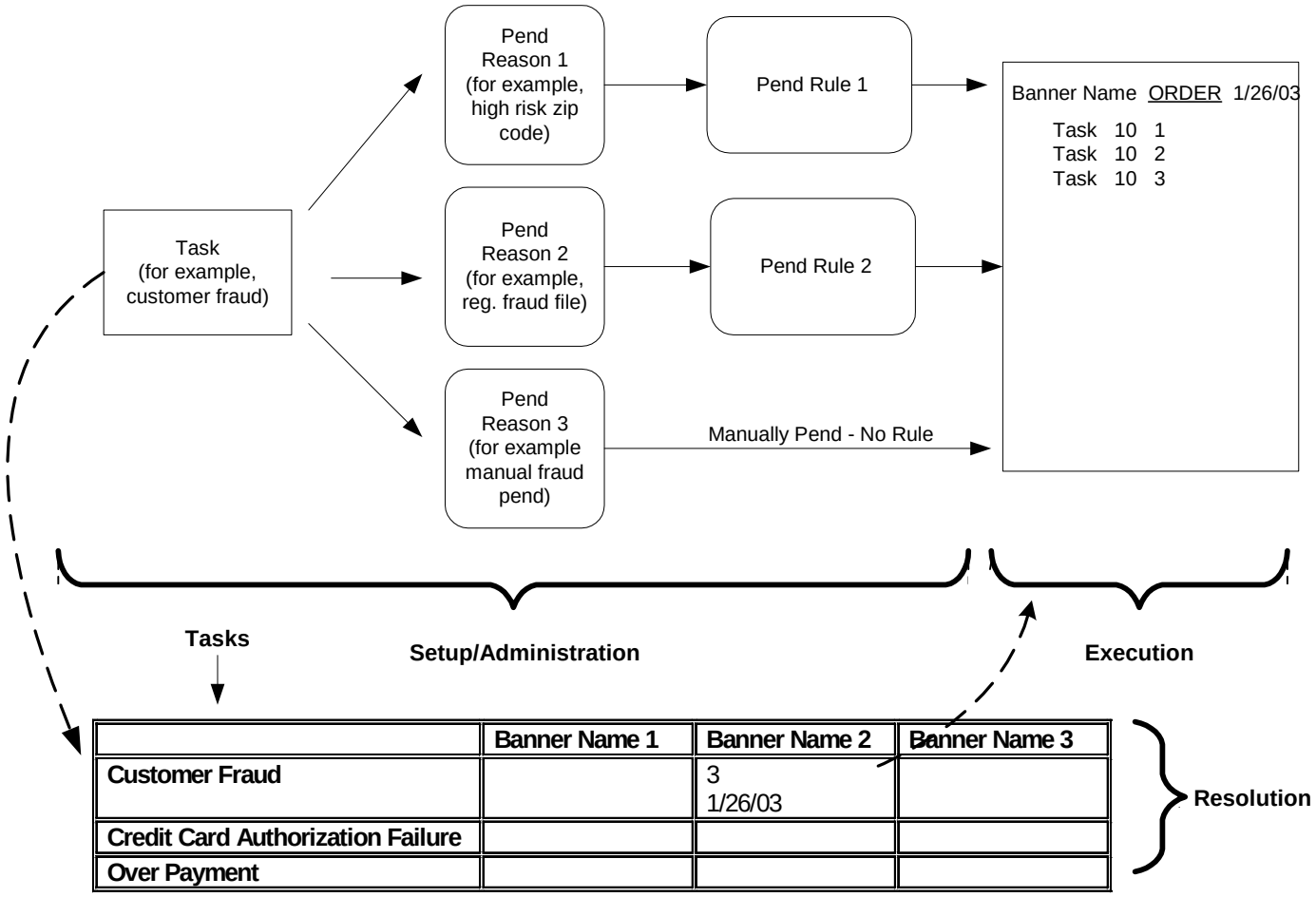
## Functional overview

This component addresses the variety of customer-service tasks that do *not* fall under the strict category of order capture or maintenance. These tasks typically cannot be addressed immediately (for example, while on the phone with the customer). They are managed as a set of offline activities with their own priorities and resource requirements.

The diagram below (along with the following passage) describes the logic behind task processing. Any time an order is pended, a task is created and routed to a worklist. New tasks are associated to the reasons that orders are pended (for example, underpayment, credit card authorization failed, and so on). An order pended for three reasons would have three tasks. Tasks are a required attribute for every pend reason in the system.

The user responsible for addressing a task can access it on a worklist. For example, the user in charge of credit card authorization failures could access a worklist of tasks related to that pended reason; the user responsible for fraud issues could access a worklist of fraud tasks; and so on. When a user double clicks on a task, he or she is taken to an order.

To release a reason, a user with supervisor privileges completes a task. For example, let’s say the user has a credit card authorization failure on his or her task list. He or she navigates to the applicable order, reauthorizes the credit card, calls the customer, and so on. After rectifying the credit card authorization issue, the user can release the reason on the order. If the order is pended for additional reasons, additional tasks would have to be addressed before the reasons could be released on an order.




**The setup, execution, and resolution of tasks**

### Print file tasks associated to activity requests

Tasks associated with activity requests can be routed to a print file. Using the print file, specific users within the system have the ability to print these tasks on an ad-hoc basis.

For example, let's say a user wishes to have shipping labels printed in order to mail back a box. The user creates an activity request related to printing shipping labels and associated to a task related to shipping label printing. The task is routed to the print file (as opposed to a worklist). Once the print screen is accessed, the user has the choice of what type of activity requests should be printed to a file. The system prints all of the labels (all of the information) associated to all of the activity requests under the task. For example, if a hundred people are waiting for labels, the user could select all of the activity requests, and the system would, based on the task, create a flat print file that could be later merged and printed using a predetermined template in either Word or Excel.

### Personal reminders

Personal reminders are tasks associated to a specific user ID within the system; these reminders are visible only to the assigned user. The user can access the list of personal reminders at his or her discretion. These tasks are routed to neither a worklist nor a print file.

If a personal reminder is created when the user is within an activity request or an order, the personal reminder is associated to the activity request or order number. Note that the association to a specific order or activity request is independent of any existing tasks on an order or on an activity request. If a personal reminder contains an association to an activity request or an order, the user can use the personal reminder to navigate to the order or the activity request.

To resolve a personal reminder, the user opens the personal reminder and enters the applicable 'resolved' command. The system marks the task as resolved and associates the correct date/time to the task. The task becomes no longer viewable on the user's personal reminder primary view.

The user can edit existing personal tasks by selecting the task. The only editable fields include task description and content. The 'last updated date' and 'create date' data is populated systematically.

## The task package in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's task component, see the following package in the RCOM Javadoc:

- com.retek.component.task

# Tax component

## Functional overview

From an order entry perspective, RCOM calls the third party tax system to calculate estimated taxes for applicable amounts. These taxes are persisted with the customer order, and they represent the amount that is charged to the customer.

During the ship confirmation process, a call to the third-party tax system must occur to document the actual tax in the third-party tax system application. The actual tax is the tax calculated on the day of shipment and may be different than the quoted tax charged to the customer. RCOM is responsible for facilitating the persistence (within the third-party tax system) of the RCOM-generated invoice number along with the shipment record. The invoice number is also persisted within RCOM in case the need arises to reconcile the two systems' processing. If there is a difference between the quoted and the actual tax, the customer is charged the lesser of the two values. Within RCOM, the quoted tax is not altered, and the actual and the charged values are also held.

Because addresses play an important role in tax processing, RCOM includes functionality (in various components) to ensure that the following address data, sent to the third-party tax system, is valid and complete:

- Order acceptance
- Ship-to
- Ship-from

For example, RCOM processing steps ensure that customers must have county information as part of their address.

## Tax component's interface with a 3rd party tax application

**Note:** The tax codes held in the system parameter table must match the codes contained in the tax engine.

RCOM includes processing to call the third-party tax system to perform the following:

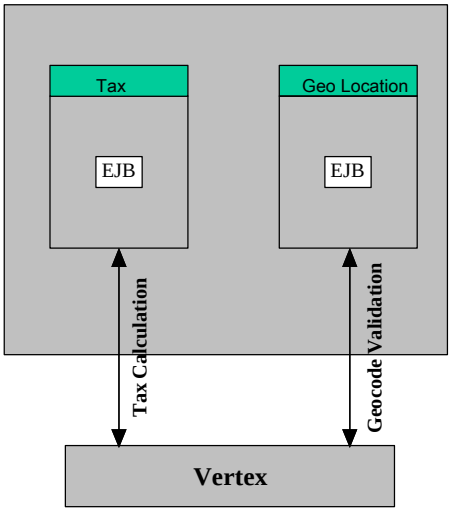
- Calculate following types of taxes for an order line:
  - VAS tax (for gifting and personalization value added services, for example)
  - Net merchandise tax
  - Shipping and handling tax (including additional delivery charge tax)
- Determine whether a tax credit applies to a previous tax calculation for an order.
- Determine whether a customer is tax exempt.
- Create a tax credit transaction for the total calculated tax amount from the created invoice. The total tax amount would be negative in cases where the taxes collected to reverse. The third-party tax system can accept both positive and negative total tax amounts. RCOM assigns the credits a tax credit accommodation type. Returns-related credits update tax liability. When creating tax credit invoices, the system uses the original order line create date.

- Update the actual taxes owed to the tax jurisdictions.
- Pass error codes to the order entry system.
- Send RCOM valid GEOCODE(s) (matching city-state-county combinations) for a given postal code that is passed from RCOM.

## Vertex overview

The RCOM application uses Vertex 2.1.4 as its 3rd party tax application. The diagram below indicates at a conceptual level the interface functionality between Vertex and the RCOM system.

Vertex has already been implemented. In production, the database connection information held as a system parameter must be changed.




## Tax processing

To calculate tax on a customer order, RCOM creates a TaxCalculatorInvoice by calling the method buildTaxCalculatorInvoiceInstance in TaxManager. Each order line in RCOM needs to be converted to TaxInvoiceLineItem. To make the conversion, RCOM calls buildTaxInvoiceLineItemInstance in TaxManager and then sets all the parameters from the customer order line number to the merchandise amount. These steps are repeated for all order lines. These TaxInvoiceLineItems are added to the TaxCalculatorInvoice.

At this point, the TaxCalculator is built by calling the buildTaxCalculatorInstance in TaxManager by passing in the Tax CalculatorInvoice. To trigger the calculation, the calculate method is called in the TaxCalculator. There is no need to call this calculate method explicitly. Calling the getMerchandiseTax on the TaxCalculator triggers this action.

The shipping and handling tax for the entire order is retrieved by setting the shipping and handling amount on the TaxCalculatorInvoice. To get the extended shipping and handling tax on the order line, a shipping and handling product code is added to the TaxInvoiceLine. To get the value added service tax (personalization and gifting cost tax) on the order line, a personalization or gifting product code is added to the TaxInvoiceLine before triggering the calculate method. RCOM is responsible for summing up the two value added service tax amounts and storing them as only one field in the system.

If Vertex becomes unavailable, RCOM continues to process the order without tax information. During order submit, RCOM again validates whether or not Vertex is available. If it is unavailable, RCOM systematically pends the order; otherwise, the tax is calculated and the order submitted normally.

### Geocode processing

During order entry, the order entry system sends a postal code to the Vertex GEOCODE API. The Vertex GEOCODE API returns all the matching city/state/county combinations that are valid for that postal code. The matches are passed back into the order entry system. If an invalid postal code is sent, an error message is returned to RCOM. If the postal code cannot be validated by Vertex, the user has to manually capture the address from the customer, and it will not be validated at that time. It will be validated when the system becomes available.

## The tax packages in Javadoc

To better understand the method-level implementation that would be required to leverage RCOM's tax component, see the following packages in the RCOM Javadoc:

- com.retek.component.tax
- com.retek.component.tax.integration.taxengine

# Chapter 6 – Internet/external APIs integration

## Functional overview

This component facilitates the use of customer order management functionality through a custom user interface (such as an internet e-commerce website). Because of processing within this component, RCOM exposes a portion of its existing functionality to an external system. Through this exposure of its APIs, RCOM provides functionality for the following three areas and more:

- Customer creation, modification and search
- Customer order summary, order creation and submission.
- Catalog request creation

To ensure that the system offers rapid performance throughout a visitor's session to the custom interface (such as an internet website), an RCOM batch process exports a media and all of its items to an XML file for the use of the external system. To access the item-related data within the XML file, the third party system does not have to make a real-time call to RCOM (which would have an adverse impact upon performance, given the volume of item-related information).

Note that any orders created via the internet have an 'internet' order source.

## Internet component batch processing

Java batch processing is associated with this component. For functional summaries about batch processing within RCOM, see the RCOM Operations Guide.

# Processing through a custom user interface (such as the internet)

The steps below provide an example of how RCOM is used in conjunction with a custom interface (such as an internet e-commerce website). During internet processing, two primary RCOM calls occur through APIs. The first call is to 'calculate my shopping cart'. This call can be made as many times as necessary. Each time this call occurs, RCOM can look up the order again, apply changes, perform recalculations, and so on. The second call is to create the order and submit the order. Both primary calls are shown in the steps below. Note that other smaller APIs are used in other calls (related to ATP, customer creation, and so on).

1 A website is built with a custom user interface.

2 Through the website, a user inserts customer information and item information.

3 The information is conveyed in a direct call to RCOM through its APIs.

4 RCOM's functionality performs the following and informs the user of the results:

- Creates the customer.
- Uses ATP functionality to determine whether or not the items are in stock.
- Calculates taxes, prices and so on.

5 RCOM sends this data back in real time to the user as a pended order, that is, a 'shopping cart' on its way to checkout. A shopping cart is the minimum amount of information needed to create an order. Note that an external system may request ATP information or create customers at any time. However, a pended order is only created once the third party provides a minimum set of data (as outlined in the APIs).

6 The user performs one of the following:

- Adds and/or removes items from the order.

OR

- Inserts payment information and submit the data ('order submit' is the equivalent command in the RCOM application).

7 The information is conveyed in a call to RCOM through the APIs, and RCOM performs one of the following and informs the user:

- Creates and submits the order, returning an order confirmation.

OR

- Returns an error requesting more information from the external system.

## A guide to using RCOM's external APIs (such as for the internet)

This design addresses the need to integrate existing client e-commerce web sites with the customer and order management functionality implemented by RCOM. This external API will henceforth be called **ExA**, for convenience and brevity.

The external API was implemented primarily to deal with conditions presented by the use case shown in Figure 1, which represents a high-level view of a web ordering facility.

Given that the RCOM functionality is strongly oriented to the call center purposes that it fulfills, this design strives to provide, in a seamless fashion, a stateless representation of RCOM order creation with the following related functions:

- Obtain all selling SKUs for a given selling item
- Obtain current stock and delivery information for a given selling SKU
- Obtain all customers and associated customer data for a given set of search criteria, including customer number, name, and address information
- Obtain all existing orders and associated order data for a given set of search criteria, including order number, customer number, and various customer information
- Obtain all history events for customer and orders for a given set of search criteria
- Create and maintain customer emails, telephone numbers, addresses and preferences
- Request a catalog, for a given concept and subconcept, to be sent to a specified customer address
- Request pricing, shipping, and tax information, and associated totals, for a given set of selling SKUs (with quantities)
- Create an order, paid with any payment type, and have designated items on the order shipped to specified locations
- Create an order, and pend it for completion by conversation with a call center representative
- Recall an order by order number and modify line quantity, cancel the order, cancel lines, or cancel payments on the order
- Create order returns, exchanges and view container information

The overriding philosophy observed in implementing ExA was simplicity. RCOM is a very robust and functionally complete system, and to utilize the native APIs from an external application environment would be very difficult. ExA attempts to shield the external application as much as possible, from this complexity. Likewise, to immunize clients of this API from changes to the implementation, all interaction with the API is via established Java interfaces.

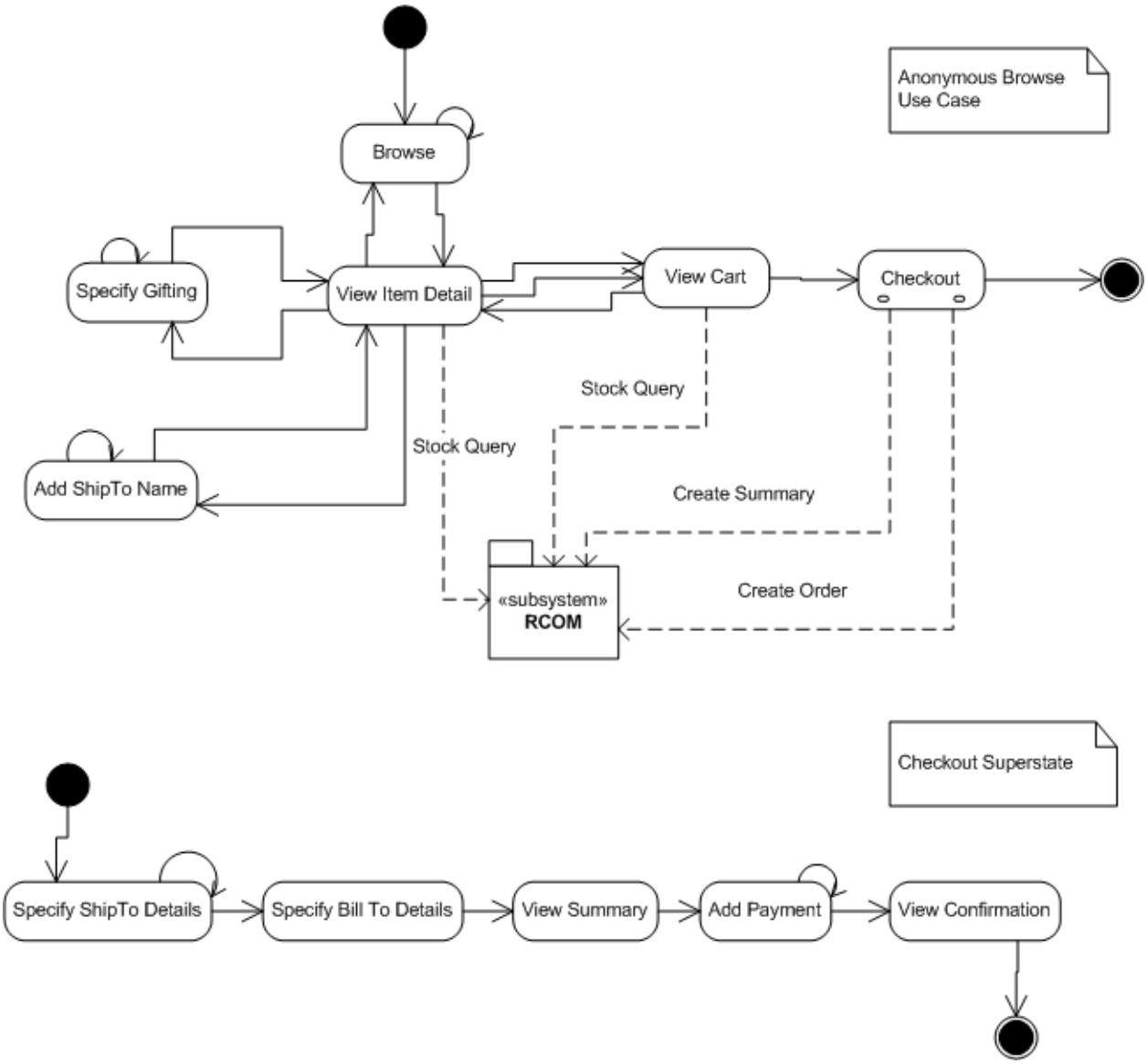

Anonymous Browse
Use Case
Browse
Specify Gifting
View Item Detail
View Cart
Checkout
Stock Query
Stock Query
Add ShipTo Name
Create Summary
«subsystem»
RCOM
Create Order
Checkout Superstate
Specify ShipTo Details
Specify Bill To Details
View Summary
Add Payment
View Confirmation

**Figure 1.**

# Information sources

The information used to drive any ExA function is obtained from one of three sources

1 Customer or third party sources

2 ExA queries

3 RCOM data extract

Customer-provided information (via the web) or third party sources are entirely beyond the scope of this document. It is assumed that this information will be provided by the integrator.

ExA queries can provide information on selling items, selling SKUs, item availability, item delivery, existing customers, and existing orders.

Periodic data extractions are exported from RCOM to facilitate querying and ordering from outside the native RCOM environment via ExA. This extract will include information about all concepts, media, selling items, selling skus, gifting, color and font data, etc.

Annotations will be included wherever data from the extract is required for ExA. See “Appendix A – Batch file layout specifications” in the RCOM Operations Guide appendix.

# Usage philosophy

ExA consists of two managers, the request manager and the API manager, and many classes that support manager operations. Each manager is instantiated by invoking a specific method on the manager factory class ***ExAManagerFactory*** and each will be discussed separately in this section.

The request manager, ***ExARequestManager***, is specifically intended to create objects observing request interfaces (primarily “setter” methods), which are to be used by the API manager. Some of these objects include:

- ExACatalogRequest
- ExAOrderLineRequest
- ExABannerRequest
- ExAMediaRequest
- ExASellingItemRequest
- ExASellingSkuRequest
- ExAPackSellingSkuRequest
- ExACustomerRequest
- ExAOrderCustomerRequest
- ExACatalogCustomerRequest
- ExAShipToLabelRequest
- ExAOrderRequest
- ExAPersonalizationRequest
- ExAPersonalizationTextLineRequest

- ExAMonogrammingRequest
- ExAGiftingSeasonRequest
- ExAGiftWrapRequest
- ExAGiftCardRequest
- ExACreditCardPaymentRequest

The API manager, ***ExAManager***, uses instances of the above classes, in combination with various Java primitives, to accomplish the desired objectives. The methods implementing those aims are:

- findSellingItem()
- findStockStatus()
- findCustomers()
- findOrders()
- findOrderCancelReasons()
- createCatalogRequest()
- createSummary()
- createOrder()
- createPendedOrder()
- updateOrder()
- createCustomer()
- updateCustomer()

This section illustrates some common e-commerce use cases and example code that might be used to implement ExA functionality to support that use case.

### Exception Handling

The API methods have been implemented with a few essential assumptions about the client's usage. The following code illustrates one of these assumptions:

```
try {
   //
   //  API invocation
   //
} catch (ExABusinessException rbe) {
   //
   //  Exception exhibiting business rule inconsistency
   //
} catch (ExASystemException rse) {
   //
```

```
      //  Exception exhibiting system inconsistency
   //
   }
```

Encounters with the ***ExABusinessException*** are largely recoverable and represent business rule violations.

The ***ExASystemException***, on the other hand, is probably not immediately recoverable and probably represents a condition in the remote server that needs to be addressed.

Native Java exceptions that are thrown will include, for example, ***IllegalArgumentException***, that represent method invocation errors.

# Obtain selling item info

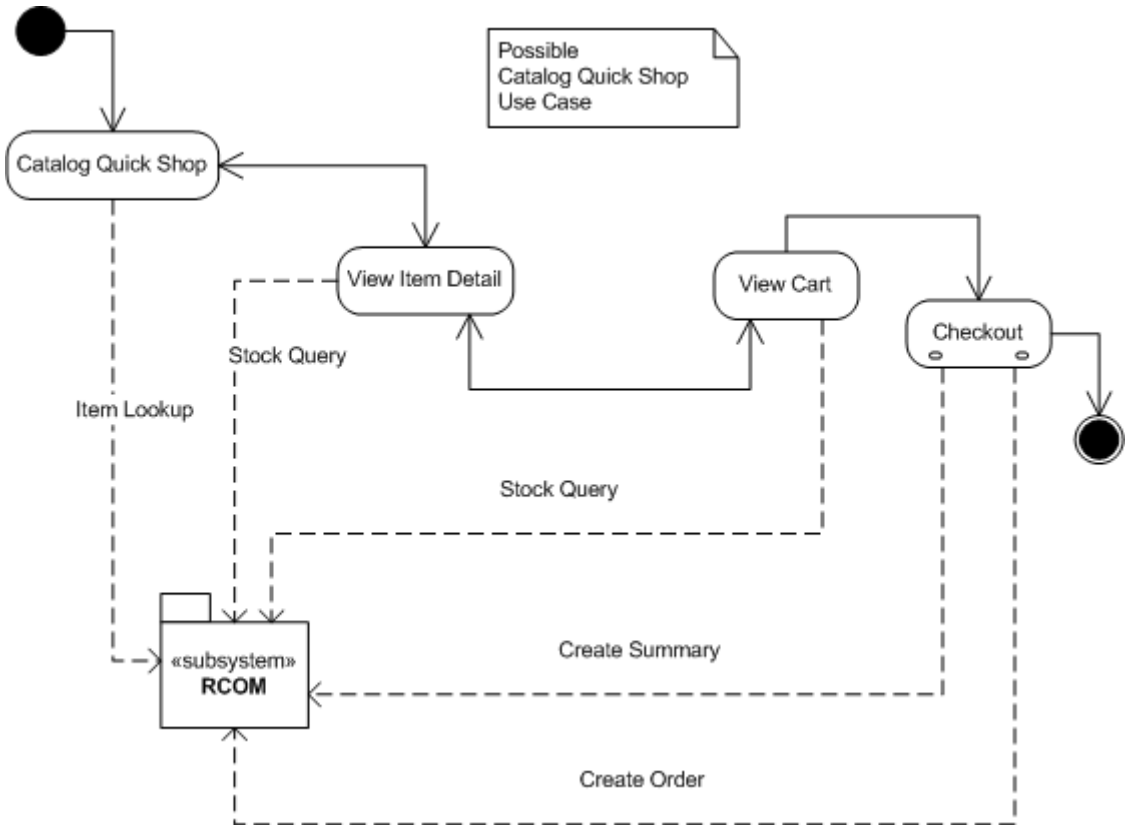



**Figure 2.**

Given that the RCOM data extract contains all pertinent information about selling items in extracted media, this might seem, at first glance, to be superfluous. However, if a requested selling item is not from a currently extracted media, information may only be available from the native database. The ***findSellingItem()*** method supplies access to that information. Figure 2 shows a *Catalog Quick Shop* feature that allows ad hoc entry of published catalog selling item information. The *Item Lookup* function corresponds to this method.

The following code excerpt shows query and retrieval of a uniquely-designated selling item:

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getRequestManager();


// The banner number is from the RCOM data extract.
String bannerNumber = “9900”;


// The media number may originate from either the RCOM data extract,
or from
// a UI field prompting for a catalog identifier.
```

```
String mediaNumber = “001”;


// The media object needed by the findSellingItem() API must first
be constructed,

// and it requires the prior existence of a banner instance.

ExABanner banner = (ExABanner)
reqMgr.buildExABannerRequest(bannerNumber);

ExAMedia media = (ExAMedia) reqMgr.buildExAMediaRequest(banner,
mediaNumber);


// The selling item number will very likely come from a UI field
which prompts

// for a catalog entry. Or it, too, may originate in the RCOM data
extract.

String sellingItemNumber = “100006666”;


ExASellingItem sellingItem = mgr.findSellingItem(media,
sellingItemNumber);
```

The *bannerNumber*, *mediaNumber*, and *sellingItemNumber* are data from the RCOM data extract or synthesized from customer or third party information. The returned ***ExASellingItem*** instance identifies various characteristics of the selling item as well as all selling SKUs, the actual, orderable item, encompassed by the selling item.

# Obtain stock status

Stock and delivery information for a single selling SKU may be determined immediately using the ***findStockStatus()*** method. Figure 2 depicts the ability to *View Item Detail* by utilizing a function shown as *Stock Status*, which corresponds to this method.

The following code shows query and retrieval of a uniquely-designated selling SKU:

```
ExAManager mgr = ExAManagerFactory.getExAManager();


// The selling sku number will very likely originate in the RCOM
data extract. It’s also

// possible that it came from a request to obtain selling item
information.

String sellingSkuNumber = “100166286”;

BigDecimal quantity = new BigDecimal(3);

ExASellingSkuStockStatus stockStatus =
mgr.findStockStatus(sellingSkuNumber, quantity);


If (stockStatus.getReservableQuantity().compareTo(quantity) == -1) {

   //

   //  Supplier will be unable to meet the given quantity without
backordering

   //

}
```

The sellingSkuNumber may originate in either the data extract or from an ExA query. The specified quantity is at the client’s prerogative. The returned instance of ***ExASellingSkuStockStatus*** reveals how much of the given item is currently on hand. It also shows the expected customer deliver date of anything that is immediately ordered.

# Obtain customer information

Customer information may be obtained by querying the remote RCOM server with the ***findCustomers()*** method. A specific customer may be sought by using a known customer number (say, from a previously submitted query or order), or a set of customers may be obtained by requesting customers matching a search criteria.

**Note:** This general search capability must be used with caution to avoid exposing confidential customer information inappropriately.

The first code example shows querying for customer information based on a known customer number. The client can expect either a single, unique customer on return, or nothing if the customer doesn't exist.

```
ExAManager mgr = ExAManagerFactory.getExAManager();

ExACustomerSearchCriteria criteria = new
ExACustomerSearchCriteria();

criteria.setCustomerNumber("1045993");

Set customers = mgr.findCustomers(criteria);

If (customers.size() == 0) {
   //
   //  No such customer exists
   //
} else {
   //
   //  Requested customer was returned
   //
}
```

The returned set of customers for a successful find, in the above scenario, will consist of a single instance of an ***ExACustomer*** object.

The following example implements a fuzzier search utilizing other fields in the ***ExACustomerSearchCriteria*** class.

```
ExAManager mgr = ExAManagerFactory.getExAManager();

ExACustomerSearchCriteria criteria = new
ExACustomerSearchCriteria();

criteria.setFirstName("John");

criteria.setLastName("Smith");

Set customers = mgr.findCustomers(criteria);

If (customers.size() == 0) {
   //
   //  No such customer exists
   //
```

```
} else {
   //
   //  Matching customer record(s) were returned
   //
}
```

In this case, the returned set for a successful find may well consist of more than one ExACustomer object. It is up to the client to responsibly deal with multiple ***ExACustomer*** information.

# Obtain history event information

History Event information may be obtained by querying the remote RCOM server with the ***findHistoryEvents()*** method. History events bay be sought by using a known customer number (from a previously submitted customer query or request) or by using a known order number. Here is some example code that shows querying history events based on customer information.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExAHistoryEventSearchCriteria criteria = new
ExAHistoryEventSearchCriteria();
criteria.setCustomerNumber(“1045993”);
criteria.setVisibility(“B”);
Set historyEvents = mgr.findHistoryEvents(criteria);
If (historyEvents.size() == 0) {
   //
   //  No history events exist for customer
   //
} else {
   //
   //  Requested history events were returned
   //
}
```

There are three visibility codes that can be set on the criteria: “C” returns only customer history events, “O” return only order history events and “B” return both types of history events for the criteria.

# Create customer

The ***createCustomer()*** method requires a number of things to complete:

1 An instance of ***ExAUser*** representing the legitimate ExA user identifier

2 An instance of ***ExACustomerRequest*** representing a new customer with fully specified demographics, address, etc.

Some minimal amount of information is required to create a customer. A name and a bill to address are the minimum amount required. The following example illustrates creating a customer:

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();


// The user name is provided as part of the integration work and
must be a legitimate
// user name with the RCOM application. This is required for all
transactions that may
// persist information.
ExAUser user = new ExAUser();
user.setUserName(“SpecialInternetUserName”);


ExACustomerRequest cust = reqMgr.buildExACustomerRequest();


// Assign name information
cust.setFirstName(“John”);
cust.setMiddleInitial(“J”);
cust.setLastName(“Smith”);


// Create and assign email
ExAEmail email = new ExAEmail();
email.setEmailAddress(“john.smith@yahoo.com”);
cust.addEmailAddress(email);


// Create and assign telephone
ExATelephoneNumber telephone = new ExATelephoneNumber();
telephone.setTelephoneNumber(“4155556789”);
cust.addTelephoneNumber(telephone);


// Create and assign bill to address
ExAAddress billToAddress = new ExAAddress();
```

```
billToAddress.setFirstName(“John”);
billToAddress.setMiddleInitial(“Q”);
billToAddress.setAddressLine1(“Smith”);
billToAddress.setCity(“Airedale”);
billToAddress.setCounty(“Froofroo”);
billToAddress.setState(“AK”);
billToAddress.setPostalCode(“67890”);
billToAddress.setCountry(“USA”);
billToAddress.setDayPhone(“1234567890”);
billToAddress.setEveningPhone(“1234567890”);

cust.setBillToAddress(billToAddress);

// Assign user
cust.setCreatedBy(user);

// Finally, request that the customer be created
ExaCustomer customer = reqMgr.createCustomer(cust);
```

The “SpecialInternetUserName” should be replaced with the unique user identifier provided as a consequence of the integration effort. It must be a legitimate RCOM user name and should be distinct, probably on a site basis.

# Modify customer

The ***updateCustomer()*** method is used to update customer information. An instance of ***ExAUser*** representing the legitimate ExA user identifier is required. The first step of the process of modifying a customer is to retrieve an existing customer using the Obtain Customer Info steps. Once an ExACustomer object has been obtained, all of the set() and add() methods may be used to modify data.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();


// The user name is provided as part of the integration work and
must be a legitimate
// user name with the RCOM application. This is required for all
transactions that may
// persist information.
ExAUser user = new ExAUser();
user.setUserName(“SpecialInternetUserName”);


ExACustomerSearchCriteria criteria = new
ExACustomerSearchCriteria();
criteria.setCustomerNumber(“1045993”);
Set customers = mgr.findCustomers(criteria);
ExaCustomer cust = null;
If (customers.size() == 0) {
   //
   //  No such customer exists
   //
} else {
   cust = (ExaCustomer) customers.iterator().next();
}
// Create and add new email
ExAEmail email = new ExAEmail();
email.setEmailAddress(“jsmith@busi.com”);
cust.addEmailAddress(email);
cust.getBillToAddress().setCity(“Cleveland”);
cust.setUpdatedBy(user);


ExaCustomer customer = reqMgr.updateCustomer(cust);
```

Exceptions may occur during this process.

# Modify customer preferences

The ***updateCustomer()*** method is used to update customer preference information. An instance of ***ExAUser*** representing the legitimate ExA user identifier is required. The first step of the process of modifying a customer is to retrieve an existing customer using the Obtain Customer Info steps. Once an ExACustomer object has been obtained, preferences can be retrieved and modified. Preferences are unique by banner display code, so if only a single banner preference is being altered, the banner display code is needed to identify the proper object.

```
ExAManager mgr = ExAManagerFactory.getExAManager();

ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();


// The user name is provided as part of the integration work and
must be a legitimate

// user name with the RCOM application. This is required for all
transactions that may

// persist information.

ExAUser user = new ExAUser();

user.setUserName(“SpecialInternetUserName”);


ExACustomerSearchCriteria criteria = new
ExACustomerSearchCriteria();

criteria.setCustomerNumber(“1045993”);

Set customers = mgr.findCustomers(criteria);

ExaCustomer cust = null;

If (customers.size() == 0) {

   //

   //  No such customer exists

   //

} else {

   cust = (ExaCustomer) customers.iterator().next();

}

Set preferenceSet = cust.getCustomerPreferences();

for (Iterator iterator = preferenceSet.iterator();
iterator.hasNext();) {

   ExACustomerPreference preference = (ExACustomerPreference)
iterator.next();

   preference.setDoNotCall(true);

}

cust.setUpdatedBy(user);


ExaCustomer customer = reqMgr.updateCustomer(cust);
```

Exceptions may occur during this process.

# Obtain order info

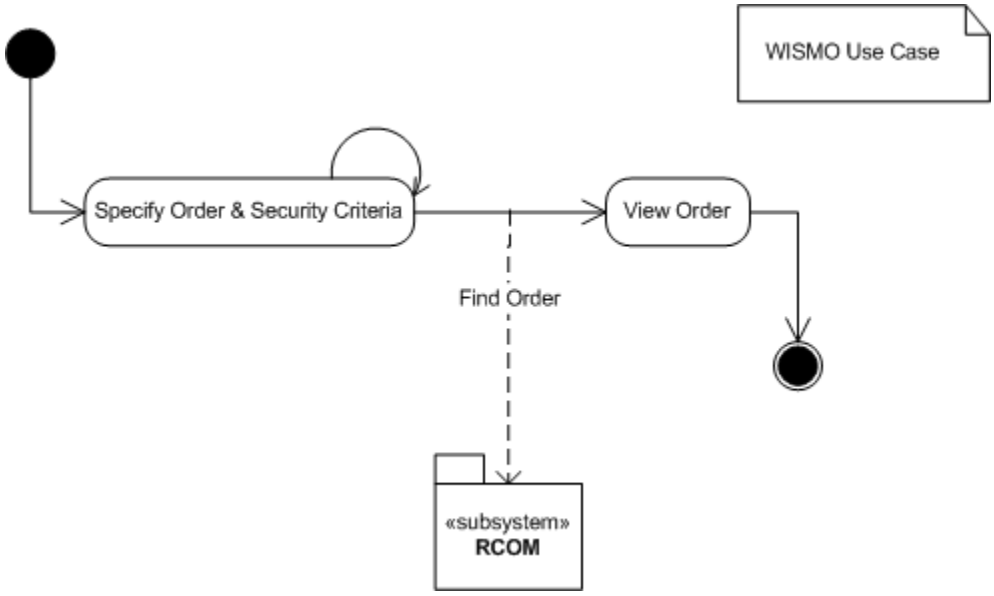



**Figure 3.**

Order information may be obtained by querying the remote RCOM server with the ***findOrders()*** method, as shown in Figure 3. A specific order may be sought by using a known order number (say, from the response to a previously submitted order), or a set of orders may be obtained by requesting orders matching a search criteria.

**Note:** This general search capability must be used with caution to avoid exposing confidential order information inappropriately.

The first code example shows querying for order information based on a known order number. The client can expect either a single, unique order on return, or nothing if the order does not exist.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExAOrderSearchCriteria criteria = new ExAOrderSearchCriteria();


// The order number may have originated from a previous order placed
over a web UI,
// or an order placed via the RCOM call center application.
criteria.setOrderNumber(“99703”);


// Though the order number is sufficient to identify a unique order,
additional information
// might be solicited to provide a customer security cross-check.
criteria.setBillToPostalCode(“55337”);
Set orders = mgr.findOrders(criteria);
If (orders.size() == 0) {
```

```
      //
      //  No such order exists
      //
   } else {
      //
      //  Requested order was returned
      //
   }
```

The returned set for a successful find, in the above scenario, will consist of a single instance of the ***ExAOrder*** object.

The state of the order represented by the returned object can be in a number of states:

| State | Description |
|---|---|
| ExAOrder.NEW_STATE | The order is in the process of being set up; no existing order should ever appear in this state. |
| ExAOrder.PENDING_STATE | The order, for one or more specific reasons, has been “pended”; that is, all order processing has been suspended until the reasons no longer apply. |
| ExAOrder.OPEN_STATE | The order is progressing through its natural fulfillment and shipping phases. |
| ExAOrder.CLOSED_STATE | All activity on the order has been naturally completed. No further activity is currently indicated, though returns and exchanges may resurrect order activity. |
| ExAOrder.CANCELLED_STATE | All activity on the order has ceased because of cancellation. No further activity is indicated. |

The following example implements a fuzzier search utilizing other fields in the ***ExAOrderSearchCriteria*** class.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExAOrderSearchCriteria criteria = new ExAOrderSearchCriteria();
criteria.setCustomerNumber(“1045993”);
Set orders = mgr.findOrders(criteria);
If (orders.size() == 0) {
   //
   //  No such order exists
   //
} else {
   //
   //  Matching order record(s) were returned
   //
```

```
}
```

In this case, the returned set for a successful find may well consist of more than one ***ExAOrder*** object. It is up to the client to responsibly deal with multiple ***ExAOrder*** information.

# Request a catalog

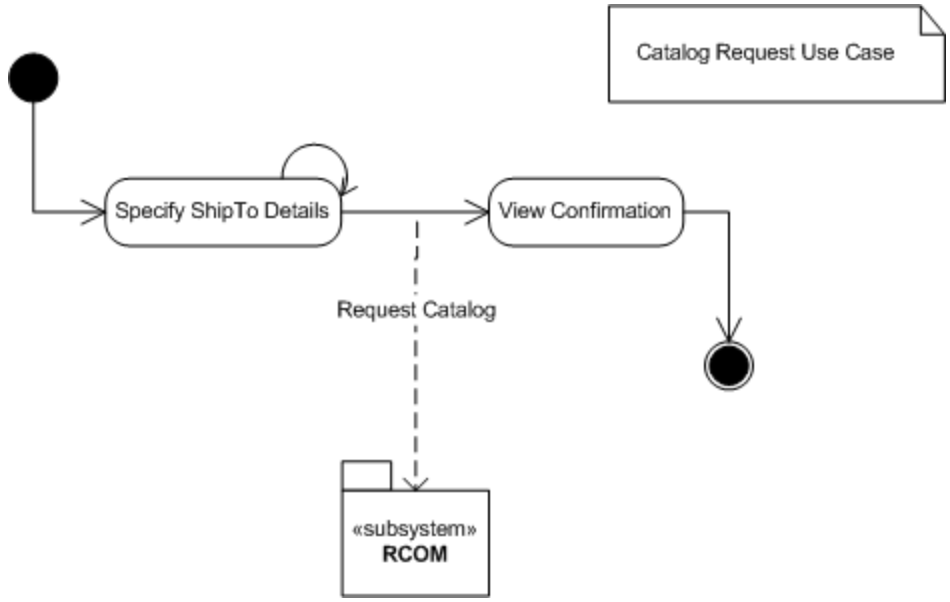



**Figure 4.**

The ***createCatalogRequest()*** method is a bit more complex than those in preceding sections. The catalog request requires a number of things to complete:

1 An instance of ***ExAUser*** representing the legitimate ExA user identifier

2 An instance of ***ExACatalogCustomerRequest*** representing either an existing customer (identified by the customer number) or a new customer with fully specified demographics, address, etc.

3 A banner number representing the numeric code of the concept from which the catalog is to be ordered

4 A subconcept identifier, which may qualify the banner number; a missing subconcept will uniquely identify only a concept with no subconcepts, and may not provide sufficient information to order a catalog successfully

Figure 4 shows a *Catalog Request* feature that allows a published catalog to be remotely ordered.

The following example illustrates an existing customer (whose number is known from, for example, a previous order) requesting a catalog.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();


// The user name is provided as part of the integration work and
must be a legitimate
// user name with the RCOM application. This is required for all
transactions that may
// persist information.
ExAUser user = new ExAUser();
```

```
user.setUserName(“SpecialInternetUserName”);


ExACatalogCustomerRequest cust =
reqMgr.buildExACatalogCustomerRequest();
ExACatalogRequest catRequest = reqMgr.buildExACatalogRequest();


// The customer number might be entered from the UI by an attentive,
existing customer.
cust.setCustomerNumber(“1045993”);


catRequest.setCustomer(cust);


// The banner number is from the RCOM data extract.
catRequest.setBannerNumber(“138”);


// The subconcept id is from the RCOM data extract.
catRequest.setSubConceptId(“7740”);
catRequest.setCreatedBy(user);


ExACatalogRequestResponse response =
mgr.createCatalogRequest(catRequest);
```

The “SpecialInternetUserName” should be replaced with the unique user identifier provided as a consequence of the integration effort. It must be a legitimate RCOM user name and should be distinct, probably on a site basis.

The banner number is the code from the data extract representing a specific concept. The subconcept id is also a code from the data extract, and it represents the specific subconcept within a concept.

The returned ***ExACatalogRequestResponse*** instance reveals the estimated delivery date of the catalog, a description of the catalog type ordered, and the customer number.

# Request order summary

The order summary returned by the ***createSummary()*** method represents an order that has been submitted to the server for pricing, tax, and shipping evaluation, all of which, including totals, are reported to the requesting client upon completion. The order thus submitted to the server is in a state not yet amenable to completion (since it does not yet have a payment associated with it).

This is one of the more complex methods. The order summary request requires a number of things to complete:

1 An instance of ***ExAUser*** representing the legitimate RCOM user identifier

2 An instance of ***ExAOrderCustomerRequest*** representing either an existing customer (identified by the customer number) or a new customer with fully specified demographics, address, etc.

3 At least one instance of ***ExAOrderLineRequest***, representing an order line

4 A banner number representing the numeric code of the concept from which the catalog is to be ordered

5 An order source code representing either generic internet (***ExAOrder.INTERNET_SOURCE***) source or gift registry (***ExAOrder.GIFT_REGISTRY_SOURCE***) source

6 A legitimate, fully-specified bill-to address

```
ExAManager mgr = ExAManagerFactory.getExAManager();

ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();


// The selling sku number will very likely originate in the RCOM
data extract. It’s also

// possible that it came from a request to obtain selling item
information.

String sellingSkuNumber = “100166286”;


BigDecimal requestedQuantity = new BigDecimal(3);


// The user name is provided as part of the integration work and
must be a legitimate

// user name with the RCOM application. This is required for all
transactions that may

// persist information.

ExAUser user = new ExAUser();

User.setUserName(“SpecialInternetUserName”);


ExACustomerRequest cust = reqMgr.buildExAOrderCustomerRequest();

ExAOrderRequest orderRequest = reqMgr.buildExAOrderRequest();
```

```
// The customer number might be entered from the UI by an attentive,
existing customer.
cust.setCustomerNumber(“1045993”);

Set shipTos = new HashSet();
ExAShipToLabelRequest shipTo = reqMgr.buildExAShipToLabelRequest();
shipTos.addShipToLabelRequest(shipTo);
Set orderLines= new HashSet();
shipTo.setOrderLines(orderLines);
ExAOrderLineRequest orderLine = reqMgr.buildExAOrderLineRequest();
orderLines.addOrderLine(orderLine);
orderLine.setSellingSku(sellingSkuNumber);
orderLine.setRequestedQuantity(requestedQuantity);

ExAAddress shipToAddress = new ExAAddress();
shipTo.setAddress(shipToAddress);
shipToAddress.setFirstName(“John”);
shipToAddress.setMiddleInitial(“Q”);
shipToAddress.setAddressLine1(“Smith”);
shipToAddress.setCity(“Airedale”);
shipToAddress.setCounty(“Froofroo”);
shipToAddress.setState(“AK”);
shipToAddress.setPostalCode(“67890”);
shipToAddress.setCountry(“USA”);
shipToAddress.setDayPhone(“1234567890”);
shipToAddress.setEveningPhone(“1234567890”);

ExAAddress billToAddress = new ExAAddress();
billToAddress.setFirstName(“John”);
billToAddress.setMiddleInitial(“Q”);
billToAddress.setAddressLine1(“Smith”);
billToAddress.setCity(“Airedale”);
billToAddress.setCounty(“Froofroo”);
billToAddress.setState(“AK”);
billToAddress.setPostalCode(“67890”);
billToAddress.setCountry(“USA”);
billToAddress.setDayPhone(“1234567890”);
```

```
billToAddress.setEveningPhone(“1234567890”);


orderRequest.setBannerNumber(“138”);
orderRequest.setOrderSourceCode(“I”);
orderRequest.setCustomerRequest(cust);
orderRequest.setShipToLabelSet(shipTos);
orderRequest.setBillToAddress(billTo);
orderRequest.setCreatedBy(user);

ExAOrder order = mgr.createSummary(orderRequest);
```

Upon completion of this method, the order summary has correctly calculated all prices, taxes, shipping, and associated totals, and the order has been systematically pended with a “Pended to Generate Internet Summary” pend reason. This state is intended to be highly transitory. When an order has been pended with this reason, and if it remains in this state, its life span is determined by the banner parameter “Internet Summary Pend Cancel Days” in RCOM’s ORG_PREFERENCE database table. After the number of days specified in this parameter, the aforementioned pended order is subject to systematic cancellation by CancelPendedOrderBatch.

For efficiency this summary, with the included order number, should be used to complete the order using the ***updateOrder()*** method. Once the order has been so completed, it moves to a state of “Open”, and is no longer subject to systematic pended-order cancellation.

# Create normal order

**Note**: This method has been deprecated and should not be used (use **updateOrder()** instead).

When an order is to be finally committed, the summary corresponding to that order, with all requisite payment information, must be submitted using the ***createOrder()*** method. Any payment methods or payment method combinations can be used to pay for the order. A credit card payment is used here as an example.

Upon successful commitment of the order, the reported order state should be ***ExAOrder.OPEN_STATE***. If the reported state reflects any flavor of ***ExAOrder.PENDING_STATE***, as it might with credit card problems or when defined ordering limits have been exceeded, the user placing the order should be urged to call the appropriate call center to rectify any problems.

# Create pended order

It is possible to create and commit an order in a ***ExAOrder.PENDING_STATE*** state. The rationale for doing so may vary greatly, but a succinct example is the user who does not wish to divulge credit card information over the web. The order will be created and pended, and a quick phone conversation (and the appropriate order number) with a call center representative can complete the order with payment information.

Following is an example code sequence of this scenario.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();
   //
   //  creation of order for summary
   //
ExAOrder order = mgr.createSummary(orderRequest);


ExAOrder orderResponse = mgr.createPendedOrder(order);
```

This is not to say that a pended order may not have payments applied to it. In fact, the order may be 100% complete and correct and still be pended for only user-known reasons.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();
   //
   //  creation of order for summary
   //
ExAOrder order = mgr.createSummary(orderRequest);


// The account number and optional PIN number will likely be
specified by UI fields.
```

```
String accountNumber = “4000000000000001”;
String pinNumber = “0123”;


// The tender type code is from the RCOM data extract. It must
correspond to the
// credit card type (e.g., Visa, Discover, MasterCard, etc.) that is
usually
// specified within a UI field.
String tenderTypeCode = “3000”;


Date expiryDate = new Date(2003, 11, 16);


Set payments = new HashSet();
ExACreditCardPaymentRequest payment =
reqMgr.buildExACreditCardPaymentRequest();
payments.addPayment(payment);
payment.setAccountNumber(accountNumber);
payment.setCardVerificationValue(pinNumber);
payment.setExpirationDate(expiryDate);
payment.setTenderTypeCode(tenderTypeCode);


ExAOrder orderResponse = mgr.createPendedOrder(order, payments);
```

Upon completion, the order will have been pended on the RCOM system with a “Internet Manually Pended” pend reason, and the user placing the order should be urged to call the appropriate call center to move the order through its normal life cycle.

While a manually pended cannot remain so indefinitely, it is no longer governed by the “Internet Summary Pend Cancel Days” in RCOM’s ORG_PREFERENCE database table, as are orders pended with the “Pended to Generate Internet Summary” pend reason.

# Modifying an order

It is possible to modify an order after calling createOrder, createSummary, pendOrder, or updateOrder. An order can have additional payments applied to it; the order can be cancelled; individual lines can have quantity changed; individual lines can be cancelled; individual payments can be cancelled; and change an address for a ship to label.

Following is an example code sequence of this scenario.

```
ExAManager mgr = ExAManagerFactory.getExAManager();
ExARequestManager reqMgr = ExAManagerFactory.getExARequestManager();


ExAOrderSearchCriteria orderSearchCriteria = new
ExAOrderSearchCriteria();
OrderSearchCriteria.setOrderNumber(“12345”);
Set foundOrderSet = mgr.findOrders(orderSearchCriteria);


If (foundOrderSet.isEmpty()) {
   // do something about missing order
}


ExAOrder preExistingOrder = (ExAOrder)
foundOrderSet.iterator().next();
Set allOrderCancelReasons = mgr.findOrderCancelReasons();
Iterator iter = allOrderCancelReasons.iterator();
ExAOrderCancelReason cancelReason = (ExAOrderCancelReason)
iter.next();


// cancel the order
exAOrder.cancel(cancelReason);


OrderSearchCriteria.setOrderNumber(“654321”);
foundOrderSet = mgr.findOrders(orderSearchCriteria);


If (foundOrderSet.isEmpty()) {
   // do something about missing second order
}


ExAOrder orderForOtherMods = (ExAOrder)
foundOrderSet.iterator().next();
Set allOrderCancelReasons = mgr.findOrderCancelReasons();
```

```
Iterator iter = allOrderCancelReasons.iterator();
ExAOrderCancelReason cancelReason = (ExAOrderCancelReason)
iter.next();
Iterator orderLineIter =
orderForOtherMods.getOrderLines().iterator();
ExAOrderLine orderLine = (ExAOrderLine)orderLineIter.next();
// cancel an order line
OrderLine.cancel(cancelReason);
// modify an order line’s quantity
ExAOrderLine orderLine2 = (ExAOrderLine) orderLineIter.next();
OrderLine2.setRequestedQuantity(new BigDecimal(5));


// canceling a payment
ExAPayment payment = (ExAPayment)
order.getPayments().iterator().next();
Payment.cancel();


// modify an address
ExAShipToLabel shipToLabel = (ExAShiptToLabel)
order.getShipToLabels().iterator().next();
ExAAddress address = shipToLabel.getAddress();
Address.setLine1(“something new”);


// now that we’ve modified the order, let’s submit it
mgr.modifyOrder(order);
```

# Create order returns

It is possible to return order lines which have been shipped completely or partially. In order to do so the order needs to have at least one order line with a shipped quantity of 1 or more.

Following is an example code sequence of this scenario for an example order. Order “888” has been created with one order line. Order was partially shipped.

```
ExAManager mgr = ExAManagerFactory.getExAManager();


//  find an existing order
ExAOrderSearchCriteria criteria = new ExAOrderSearchCriteria();


criteria.setOrderNumber(“888”);
ExAOrder order = mgr.findOrders(criteria).iterator().next();


// Get the line to be returned.
ExAOrderLine line = (ExAOrderLine)
order.getOrderLines().iterator().next();


// verify line is returnable.
boolean returnable = line.isReturnable();


// Create a return line. The request manager method needs to take in
the orderLine for
// which the return is being created.
ExAManager requestMgr = ExAManagerFactory. getExARequestManager ();
ExAReturnLine returnLine =  requestMgr.buildExAReturnLine(line);


// if returnable, get the maximum Quantity that can be returned.
BigDecimal maxReturnableQty = returnLine.getReturnableQuantity();
// Set the return quantity
returnLine.setExpectedReturnQty(maxReturnableQty);


// Select a return reason. In this example we select the first
reason.
Set returnReasons = mgr.findReturnReasons();
ExAReturnReason returnReason = (ExAReturnReason)
returnReasons.iterator().next();


// Set the Return Reason
```

```
returnLine.setReturnReason(returnReason);


// Add exchange line to order
order.addReturnLine(exaLine);

// Update order
mgr.updateOrder(order);
```

Upon completion, the order will have been saved on the RCOM system.

# Create order line exchanges

Order line exchanges can be created provided there is at least one line is being returned.

Following is an example code sequence of this scenario for an example order. Order “888” has been created with one order line that was shipped.

```
ExAManager mgr = ExAManagerFactory.getExAManager();


//  find an existing order
ExAOrderSearchCriteria criteria = new ExAOrderSearchCriteria();


criteria.setOrderNumber(“888”);
ExAOrder order = mgr.findOrders(criteria).iterator().next();


// Get the line to be returned.
ExAOrderLine line = (ExAOrderLine) order.getOrderLines().iterator().next();


// verify line is returnable.
boolean returnable = line.isReturnable();


// Create a return line. The request manager method needs to take in
the orderLine for
// which the return is being created.
ExAManager requestMgr = ExAManagerFactory. getExARequestManager ();
ExAReturnLine returnLine =  requestMgr.buildExAReturnLine(line);


// if returnable, get the maximum Quantity that can be returned.
BigDecimal maxReturnableQty = returnLine.getReturnableQuantity();


// Select a return reason
Set returnReasons = mgr.findReturnReasons();
ExAReturnReason returnReason = (ExAReturnReason)
returnReasons.iterator().next();


// Set the return quantity
returnLine.setExpectedReturnQty(maxReturnableQty);


// Set the Return Reason
returnLine.setReturnReason(returnReason);
```

```
// Add return line to order
order.addReturnLine(exaLine);



// Create exchange line
ExAOrderLineRequest exchangeLine =
requestMgr.buildExAExchangeLineRequest();


// Find a sellking Sku and set on exchange line
exchangeLine.setSellingSku(sellingSk);
exchangeLine.setRequestedQuantity(new BigDecimal(100));


// Update order
mgr.updateOrder(order);
```

Upon completion, the order will have been saved on the RCOM system with the return and exchange lines. If the addition of exchange lines creates an order balance (that is, cost of lines exchanged exceeds that of the returned) then the order will be pended.

Payments can be made to the order by reading the order and making payments based on the order balance.

```
ExAOrderSearchCriteria criteria = new ExAOrderSearchCriteria();


criteria.setOrderNumber(“888”);
ExAOrder order = mgr.findOrders(criteria).iterator().next();
BigDecimal orderBalance = order.getBalance();


// Create and Add payment to order
ExACreditCardPaymentRequest payment =
reqMgr.buildExACreditCardPaymentRequest();
payment.setAmount(orderBalance);
payment.setAccountNumber(“.. “);
payment.setExpirationDate(“06/09.. “);
payment.setTenderTypeCode(“..“);


order.addPayment(payment);


// Update order
mgr.updateOrder(order);
```

Upon completion, the order will have been saved on the RCOM system and should be in open status.

# Order line container information

The Order API will have methods to call and retrieve all container information, which includes ship to addresses, container line information, etc.

Following is an example of obtaining all the container and its associated information.

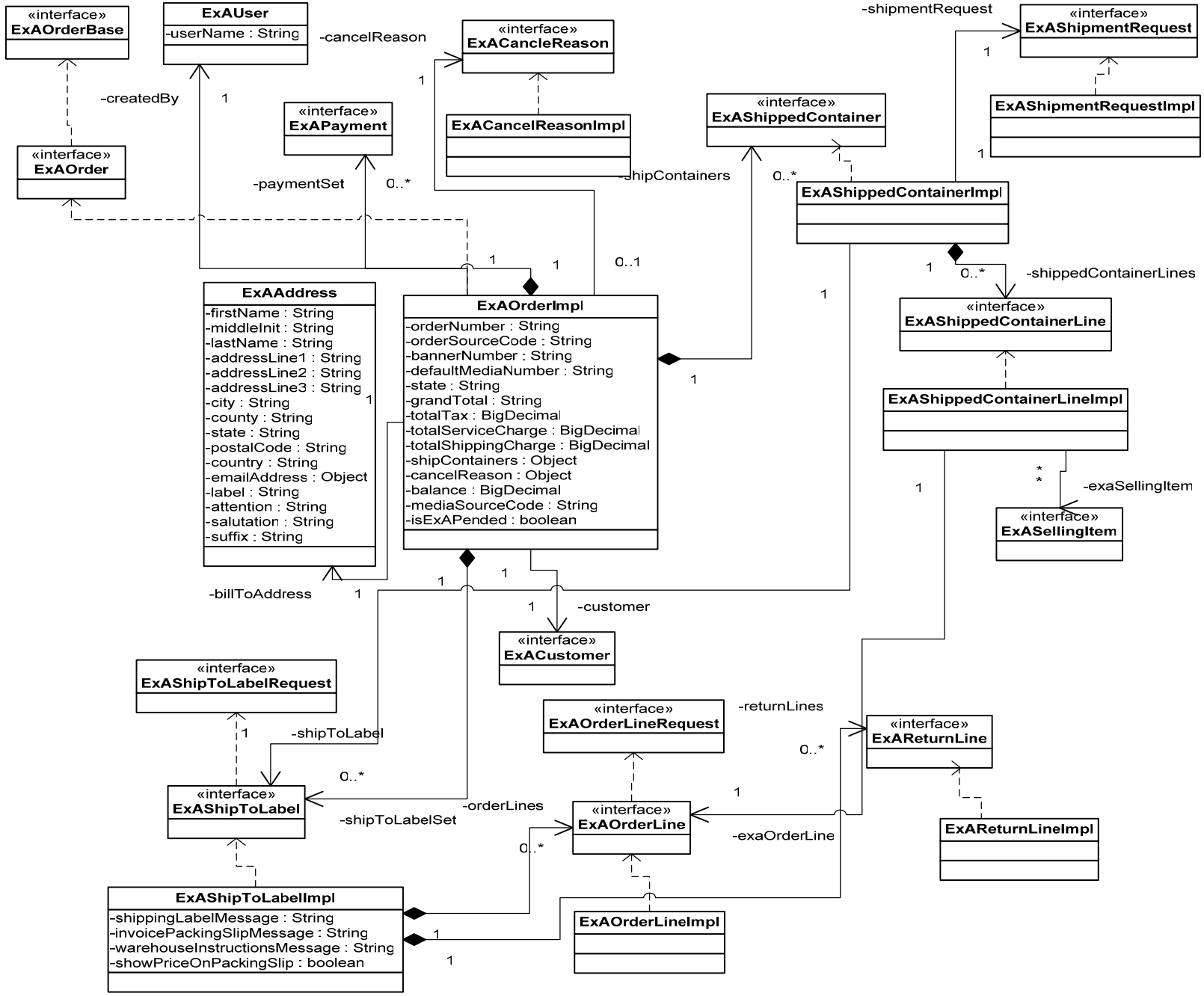
```
ExAManager mgr = ExAManagerFactory.getExAManager();


//  find an existing order
ExAOrderSearchCriteria criteria = new ExAOrderSearchCriteria();


criteria.setOrderNumber(“888”);
ExAOrder order = mgr.findOrders(criteria).iterator().next();
Set containers = exOrder.getShipContainers();
ExAShippedContainer container = (ExAShippedContainer)
containers.iterator().next();


// Get container information…
container.getRmaNumber();


container.getShippedContainerNumber();


container.getShippedDate();


container.getEstimatedArrivalDate();


// Get container Lines and container line information
Set containerLines = container.getShippedContainerLines();


ExAShippedContainerLine containerLine = (ExAShippedContainerLine)


containerLines.iterator().next();



ExaORderLine orderLine = containerLine.getOrderLine();


BigDecimal totalServiceCharge = exaline.getTotalServiceCharge());
```

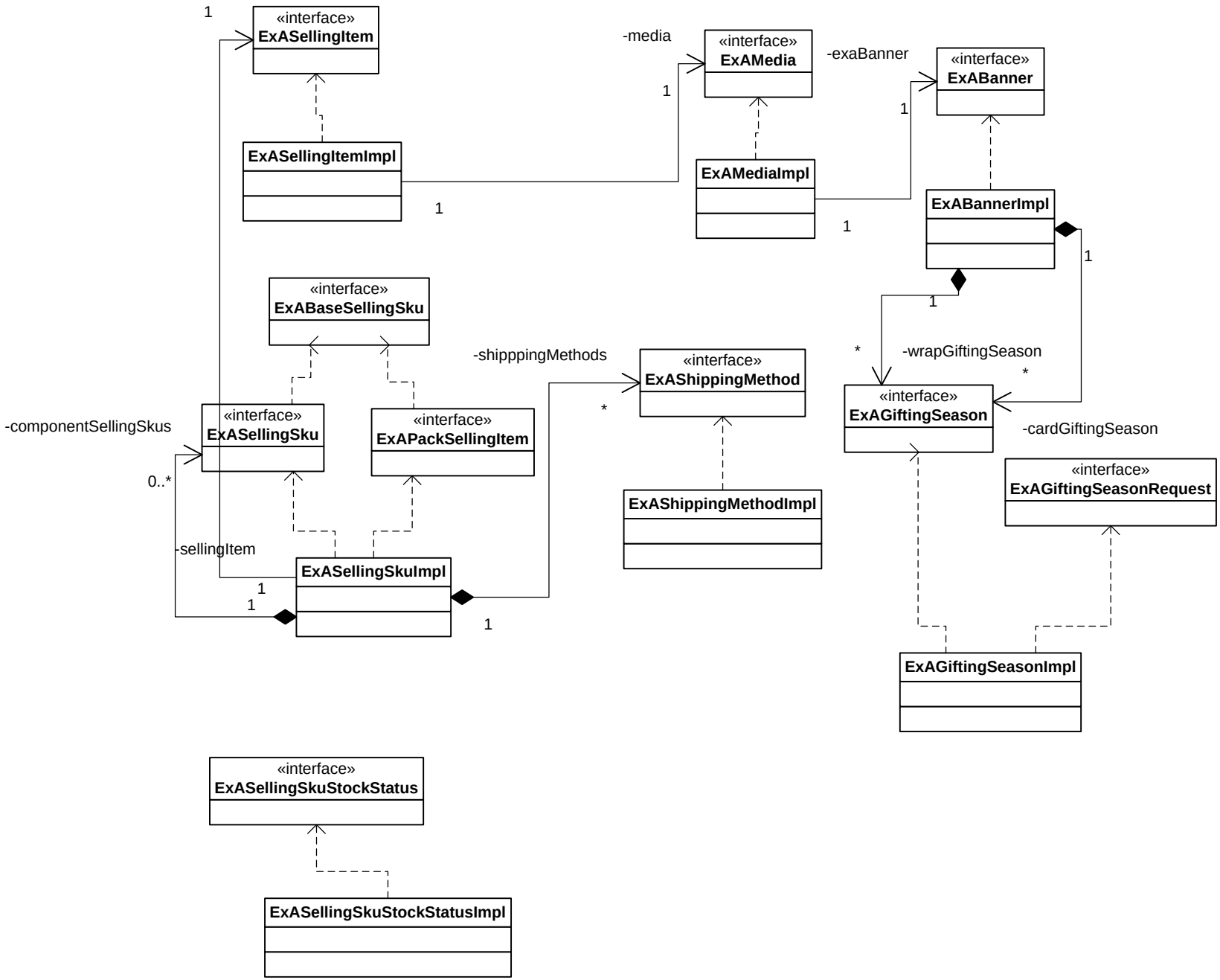
```
BigDecimal totalShippingAndDeliveryCharges =

line.getTotalShippingAndDeliveryCharges();

// Get Ship To Address for container Line
ExAShipToLabel exaShipTo = container.getShipTo();

exaShipTo.getInvoicePackingSlipMessage();

exaShipTo.getWarehouseInstructionsMessage();
// ..
```

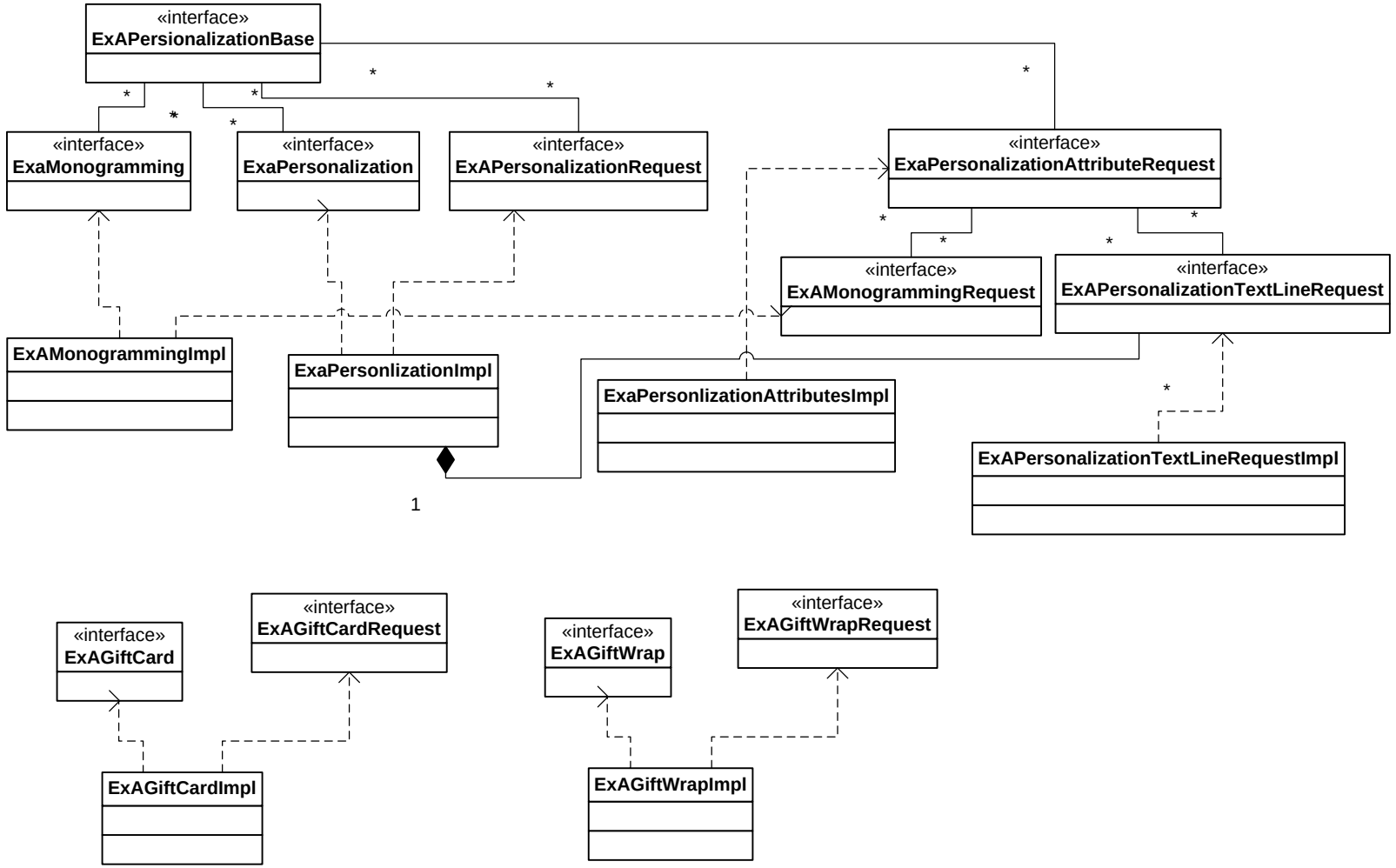
# Class diagrams

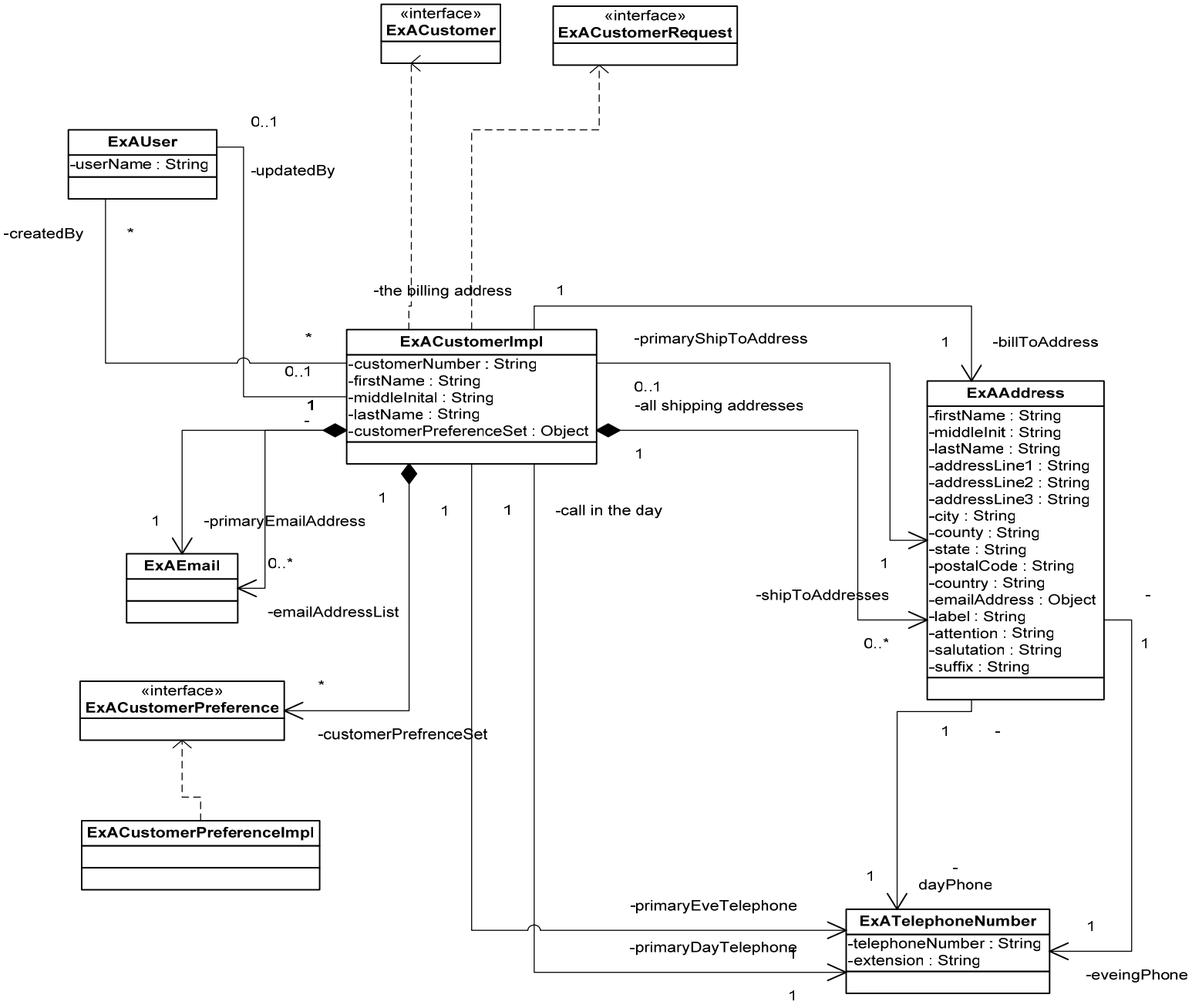
The following objects models are in this section in this order:

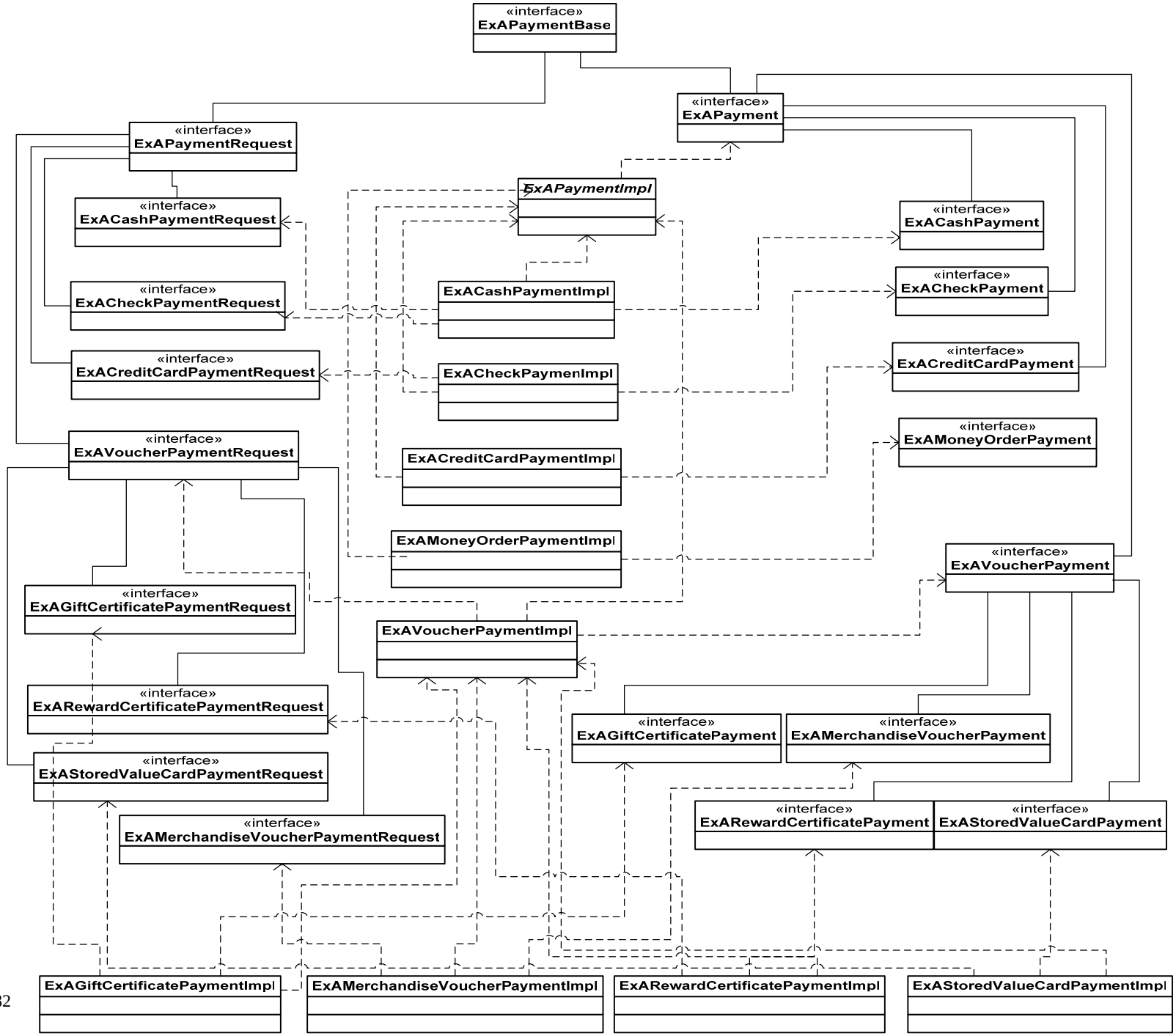
- Order
- Item
- Services
- Customer
- Payment


«interface»
ExAOrderBase
ExAUser
-userName : String
-cancelReason
«interface»
ExACancleReason
-shipmentRequest
«interface»
ExAShipmentRequest
-createdBy
1
1
1
«interface»
ExAPayment
ExACancelReasonImpl
ExAShipmentRequestImpl
1
«interface»
ExAOrder
«interface»
ExAShippedContainer
-shipContainers
0..*
ExAShippedContainerImpl
-paymentSet
0..*
1
1
0..1
1
0..*
-shippedContainerLines
ExAAddress
-firstName : String
-middleInit : String
-lastName : String
-addressLine1 : String
-addressLine2 : String
-addressLine3 : String
-city : String
-county : String
-state : String
-postalCode : String
-country : String
-emailAddress : Object
-label : String
-attention : String
-salutation : String
-suffix : String
ExAOrderImpl
-orderNumber : String
-orderSourceCode : String
-bannerNumber : String
-defaultMediaNumber : String
-state : String
-grandTotal : String
-totalTax : BigDecimal
-totalServiceCharge : BigDecimal
-totalShippingCharge : BigDecimal
-shipContainers : Object
-cancelReason : Object
-balance : BigDecimal
-mediaSourceCode : String
-isExAPended : boolean
1
«interface»
ExAShippedContainerLine
ExAShippedContainerLineImpl
1
*
*
-exaSellingItem
«interface»
ExASellingItem
1
1
-billToAddress
1
1
1
-customer
«interface»
ExACustomer
«interface»
ExAShipToLabelRequest
«interface»
ExAOrderLineRequest
-returnLines
«interface»
ExAReturnLine
1
-shipToLabel
0..*
0..*
«interface»
ExAShipToLabel
-orderLines
-shipToLabelSet
«interface»
ExAOrderLine
1
-exaOrderLine
ExAReturnLineImpl
0..*
ExAShipToLabelImpl
-shippingLabelMessage : String
-invoicePackingSlipMessage : String
-warehouseInstructionsMessage : String
-showPriceOnPackingSlip : boolean
ExAOrderLineImpl
1
1


«interface»
ExASellingItem
ExASellingItemImpl
-media
«interface»
ExAMedia
ExAMediaImpl
-exaBanner
«interface»
ExABanner
ExABannerImpl
«interface»
ExABaseSellingSku
-componentSellingSkus
«interface»
ExASellingSku
«interface»
ExAPackSellingItem
-shipppingMethods
«interface»
ExAShippingMethod
ExAShippingMethodImpl
-wrapGiftingSeason
«interface»
ExAGiftingSeason
-cardGiftingSeason
«interface»
ExAGiftingSeasonRequest
ExAGiftingSeasonImpl
-sellingItem
ExASellingSkuImpl
«interface»
ExASellingSkuStockStatus
ExASellingSkuStockStatusImpl


«interface»
ExAPersionalizationBase
«interface»
ExaMonogramming
«interface»
ExaPersonalization
«interface»
ExAPersonalizationRequest
«interface»
ExaPersonalizationAttributeRequest
«interface»
ExAMonogrammingRequest
«interface»
ExAPersonalizationTextLineRequest
ExAMonogrammingImpl
ExaPersonlizationImpl
ExaPersonlizationAttributesImpl
ExAPersonalizationTextLineRequestImpl
1
«interface»
ExAGiftCard
«interface»
ExAGiftCardRequest
«interface»
ExAGiftWrap
«interface»
ExAGiftWrapRequest
ExAGiftCardImpl
ExAGiftWrapImpl

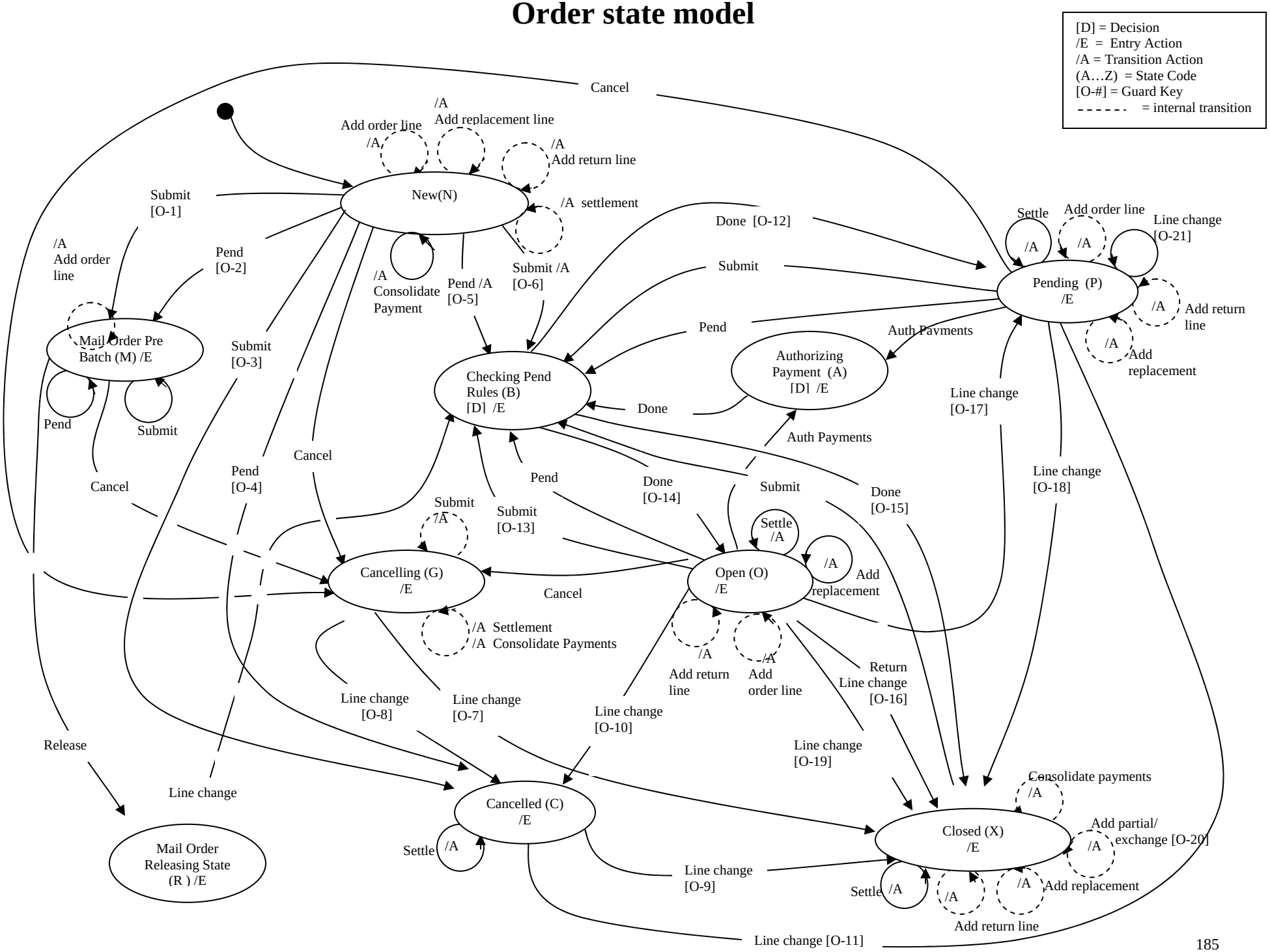

«interface»
ExACustomer
«interface»
ExACustomerRequest
ExAUser
-userName : String
0..1
-updatedBy
-createdBy
*
-the billing address
1
*
ExACustomerImpl
-customerNumber : String
-firstName : String
-middleInital : String
-lastName : String
-customerPreferenceSet : Object
0..1
1
-primaryShipToAddress
1
-billToAddress
0..1
-all shipping addresses
1
1
1
1
-call in the day
1
-primaryEmailAddress
ExAEmail
0..*
-emailAddressList
1
-shipToAddresses
0..*
ExAAddress
-firstName : String
-middleInit : String
-lastName : String
-addressLine1 : String
-addressLine2 : String
-addressLine3 : String
-city : String
-county : String
-state : String
-postalCode : String
-country : String
-emailAddress : Object
-label : String
-attention : String
-salutation : String
-suffix : String
-
1
«interface»
ExACustomerPreference
*
-customerPrefrenceSet
1
-
ExACustomerPreferenceImpl
1
-
dayPhone
-primaryEveTelephone
-primaryDayTelephone
ExATelephoneNumber
-telephoneNumber : String
-extension : String
1
1
1
-eveingPhone


«interface»
ExAPaymentBase
«interface»
ExAPayment
«interface»
ExAPaymentRequest
ExAPaymentImpl
«interface»
ExACashPaymentRequest
«interface»
ExACashPayment
«interface»
ExACheckPaymentRequest
ExACashPaymentImpl
«interface»
ExACheckPayment
«interface»
ExACreditCardPaymentRequest
ExACheckPaymentImpl
«interface»
ExACreditCardPayment
«interface»
ExAVoucherPaymentRequest
ExACreditCardPaymentImpl
«interface»
ExAMoneyOrderPayment
ExAMoneyOrderPaymentImpl
«interface»
ExAVoucherPayment
«interface»
ExAGiftCertificatePaymentRequest
ExAVoucherPaymentImpl
«interface»
ExARewardCertificatePaymentRequest
«interface»
ExAGiftCertificatePayment
«interface»
ExAMerchandiseVoucherPayment
«interface»
ExAStoredValueCardPaymentRequest
«interface»
ExAMerchandiseVoucherPaymentRequest
«interface»
ExARewardCertificatePayment
«interface»
ExAStoredValueCardPayment
ExAGiftCertificatePaymentImpl
ExAMerchandiseVoucherPaymentImpl
ExARewardCertificatePaymentImpl
ExAStoredValueCardPaymentImpl

# Appendix A – State model diagrams

For four functional areas, order, order line, payment, and return line, the diagrams in this appendix describe states of RCOM objects. Guards, which are Boolean expressions, control transitions between states. The Guard Key that follows each state model diagram maps to the preceding diagram.

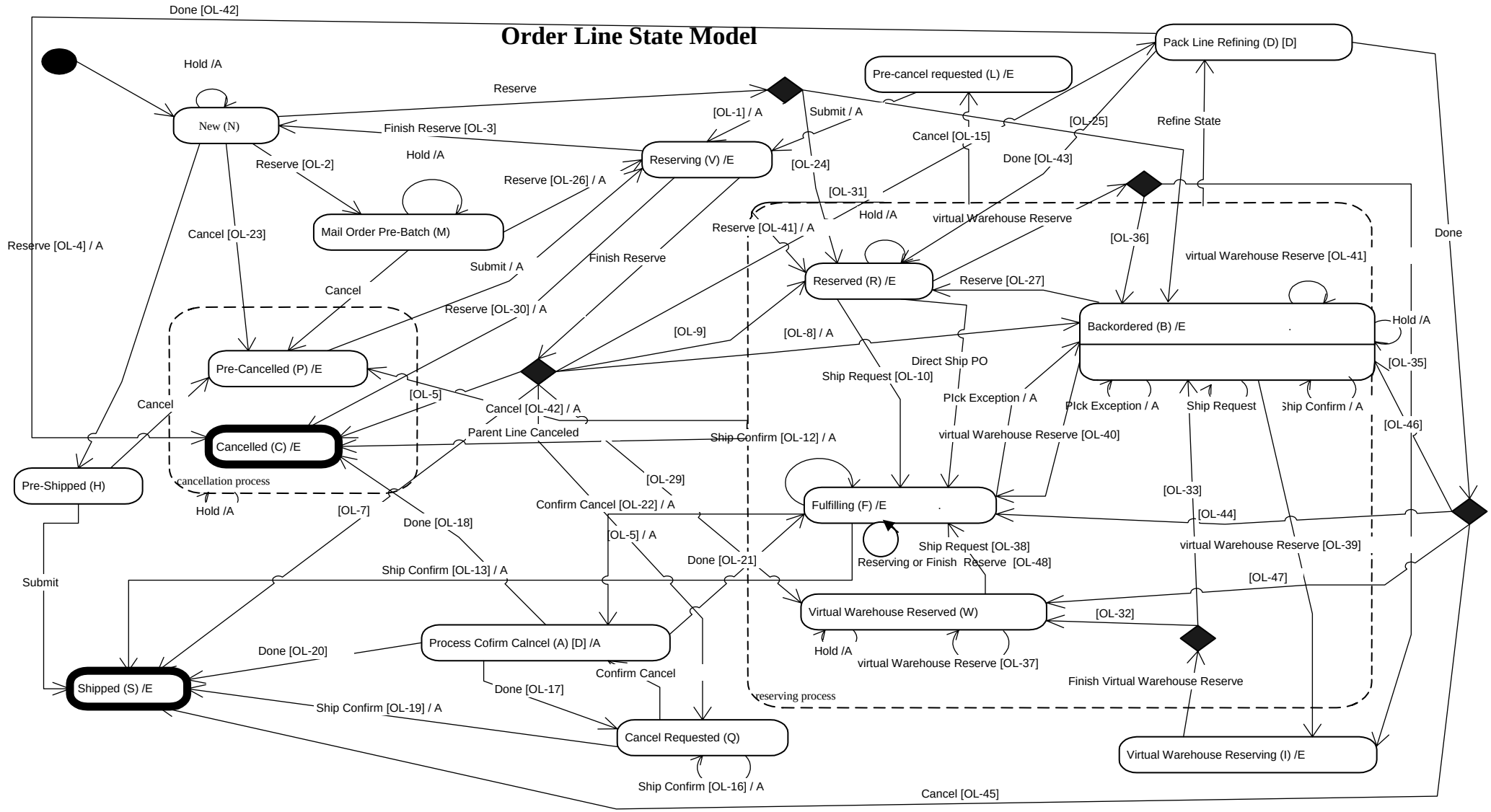
# Order state model



**Guard key**

| | |
|---|---|
| O-1 | NOT ALL ORDER LINES CANCELLED AND ORDER TYPE IS MAIL ORDER |
| O-2 | Not all order lines cancelled and order type is mail order |
| O-3 | All order lines cancelled |
| O-4 | All order lines cancelled |
| O-5 | Not all order lines cancelled and order type is not mail order |
| O-6 | Not all order lines cancelled and order type is not mail order |
| O-7 | All order line shipped or cancelled and one or more lines has a shipped quantity > 0 |
| O-8 | All order lines cancelled and no lines have a shipped quantity > 0 |
| O-9 | At least one order line has a shipped quantity > 0 |
| O-10 | All order lines cancelled and no lines have a shipped quantity > 0 |
| O-11 | All order lines cancelled and no lines have a shipped quantity > 0 |
| O-12 | Order has active pend reasons |
| O-13 | Not all order lines are cancelled |
| O-14 | Order does not have active pend reasons and not (All order lines are shipped or cancelled and all return lines are not new or pending return and all partial lines are shipped or cancelled) |
| O-15 | Order does not have active pend reasons and all order lines are shipped or cancelled and all return lines are not new or pending return and all partial lines are shipped or cancelled |
| O-16 | All order lines are cancelled or shipped and one or more lines has a shipped quantity > 0 and all return lines are returned or cancelled |
| O-17 | All order lines fulfilling or reserved and sall payments are cancelled, approved, or settled |
| O-18 | All order line shipped or cancelled and order does not have active pend reasons and one or more lines has a shipped quantity > 0 |
| O-19 | All order line shipped or cancelled and one or more lines has a shipped quantity > 0 |
| O-20 | Order line type is partial or exchanged |

| | |
|---|---|
| O-21 | Not all order lines shipped or cancelled |


Order Line State Model
Done [OL-42]
Hold /A
New (N)
Reserve
Finish Reserve [OL-3]
Reserve [OL-2]
Hold /A
Reserving (V) /E
[OL-1] / A
Submit / A
[OL-24]
Pre-cancel requested (L) /E
Cancel [OL-15]
[OL-25]
Refine State
Pack Line Refining (D) [D]
Done [OL-43]
Reserve [OL-26] / A
Mail Order Pre-Batch (M)
Cancel [OL-23]
Reserve [OL-4] / A
Submit / A
Finish Reserve
Cancel
Reserve [OL-30] / A
[OL-31]
Hold /A
virtual Warehouse Reserve
Reserve [OL-41] / A
[OL-36]
virtual Warehouse Reserve [OL-41]
Done
Reserved (R) /E
Reserve [OL-27]
Backordered (B) /E
Hold /A
[OL-35]
[OL-9]
[OL-8] / A
Direct Ship PO
Ship Request [OL-10]
Pick Exception / A
Pick Exception / A
Ship Request
Ship Confirm / A
Pre-Cancelled (P) /E
[OL-5]
Cancel
Cancel [OL-42] / A
Parent Line Canceled
Cancelled (C) /E
Ship Confirm [OL-12] / A
virtual Warehouse Reserve [OL-40]
[OL-46]
cancellation process
Pre-Shipped (H)
Hold /A
[OL-7]
Done [OL-18]
[OL-29]
Confirm Cancel [OL-22] / A
[OL-5] / A
Fulfilling (F) /E
[OL-33]
[OL-44]
Ship Request [OL-38]
Reserving or Finish Reserve [OL-48]
virtual Warehouse Reserve [OL-39]
Done [OL-21]
Ship Confirm [OL-13] / A
Submit
[OL-47]
Virtual Warehouse Reserved (W)
[OL-32]
Process Cofirm Calncel (A) [D] /A
Done [OL-20]
Hold /A
virtual Warehouse Reserve [OL-37]
Confirm Cancel
Finish Virtual Warehouse Reserve
Done [OL-17]
Shipped (S) /E
Ship Confirm [OL-19] / A
reserving process
Cancel Requested (Q)
Virtual Warehouse Reserving (I) /E
Ship Confirm [OL-16] / A
Cancel [OL-45]

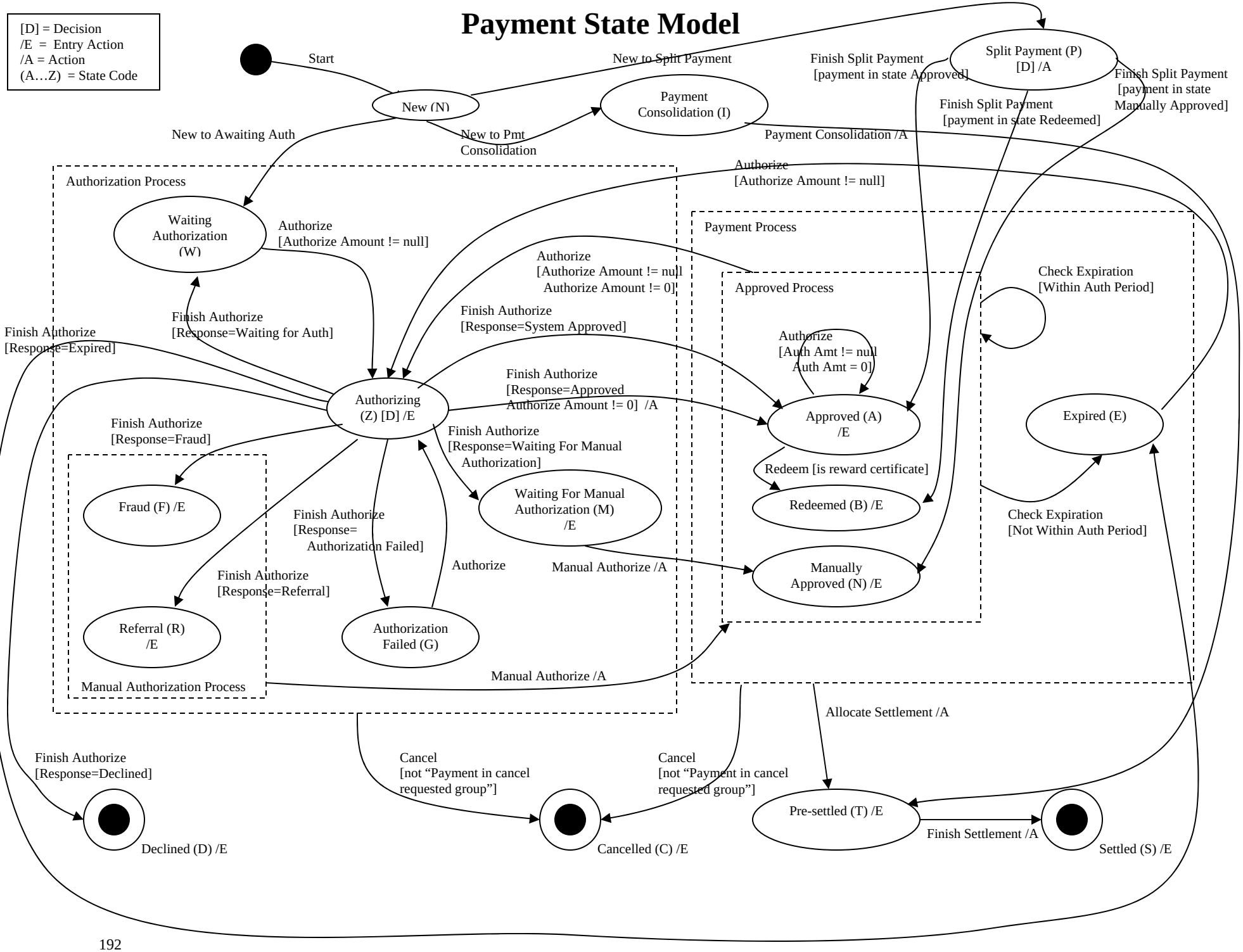
**Guard key**

| | |
|---|---|
| OL-1 | Not mail order pre-batch nor order line type zero sale and not pack component line |
| OL-2 | Order is mail order pre-batch |
| OL-3 | Not reserved quantity > 0 |
| OL-4 | Line is zero sale |
| OL-5 | Cancel quantity >= 0 and fulfilling quantity > 0 and not reserved Qty = 0 and shipped Qty = 0 and order line is Pack not reserved at component level |
| OL-6 | Cancel quantity >= 0 and not reserved quantity = 0 and fulfilling quantity = 0 and shipped quantity = 0 |
| OL-7 | Fulfilling quantity = 0 and backorder quantity = 0 and reserved quantity = 0 and shipped quantity > 0 |
| OL-8 | Back ordered quantity > 0 and cancelled quantity = 0 and not reserved quantity = 0 and order line is Pack not reserved at component level |
| OL-9 | Cancelled quantity > 0 and back ordered quantity = 0 and not reserved quantity = 0 virtual wh reserved quantity = 0 and order line is Pack not reserved at component level |
| OL-10 | Backordered quantity = 0 |
| OL-11 | Sufficient payments authorized |
| OL-12 | Fulfilling quantity > shipped quantity |
| OL-13 | Shipped quantity  = fulfilling quantity |
| OL-14 | Fulfilling quantity = 0 |
| OL-15 | Fulfilling quantity > 0 |
| OL-16 | Shipped quantity < fulfilling quantity |
| OL-17 | Fulfilling quantity > 0 and cancel reason not = direct ship vendor cancel |
| OL-18 | Fulfilling quantity = 0 and shipped quantity = 0 |
| OL-19 | Shipped quantity = fulfilling quantity |
| OL-20 | Fulfilling quantity = 0 and shipped quantity > 0 |

| | |
|---|---|
| OL-21 | Fulfilling quantity > 0 and cancel reason = direct ship vendor cancel |
| OL-22 | Order line is direct ship |
| OL-23 | Order line is no longer available |
| OL-24 | Order line is pack component and backorder quantity = 0 |
| OL-25 | Order line is pack component and backorder quantity > 0 |
| OL-26 | Mail order releasing state = order line releasing state |
| OL-27 | Order line is pack component |
| OL-28 | Order line is not pack component |
| OL-29 | Backordered = 0 and not (reserved = 0 and virtual warehouse reserved = 0 and canceled >= 0) and order line is not a pack line reserved only at the component level and the warehouse reserved > 0 |
| OL-30 | Not reserved quantity > 0 and requested quantity = 0 and is mail order |
| OL-31 | Order line is a pack line reserved only at the component level |
| OL-32 | Backordered quantity = 0 |
| OL-33 | Backordered quantity > 0 |
| OL-34 | Recommended pack line state is canceled |
| OL-35 | (order line is not pack component)  or (order line is pack component and backordered quantity = 0) |
| OL-36 | Order line is pack component and backordered quantity > 0 |
| OL-37 | Reserved quantity > 0 and warehouse reserved quantity > 0 |
| OL-38 | Backordered quantity = 0 |
| OL-39 | Order line is not pack component and order line has Ship to |
| OL-40 | Backordered quantity = 0 |
| OL-41 | Order line is not pack component |
| OL-42 | Pack line recommended state is cancelled |
| OL-43 | Pack line recommended state is reserved |

| | |
|---|---|
| OL-44 | Pack line recommended state is fufilling |
| OL-45 | Pack line recommended state is shipped |
| OL-46 | Pack line recommended state is backordered |
| OL-47 | Pack line recommended state is virtual warehouse reserved |
| OL-48 | Backorder Qty = 0 and Fulfilling Qty > 0 |

# Payment State Model



[T] = Transient
/E = Entry Action
/A = Action
(A…Z) = State Code

# Return Line State Model