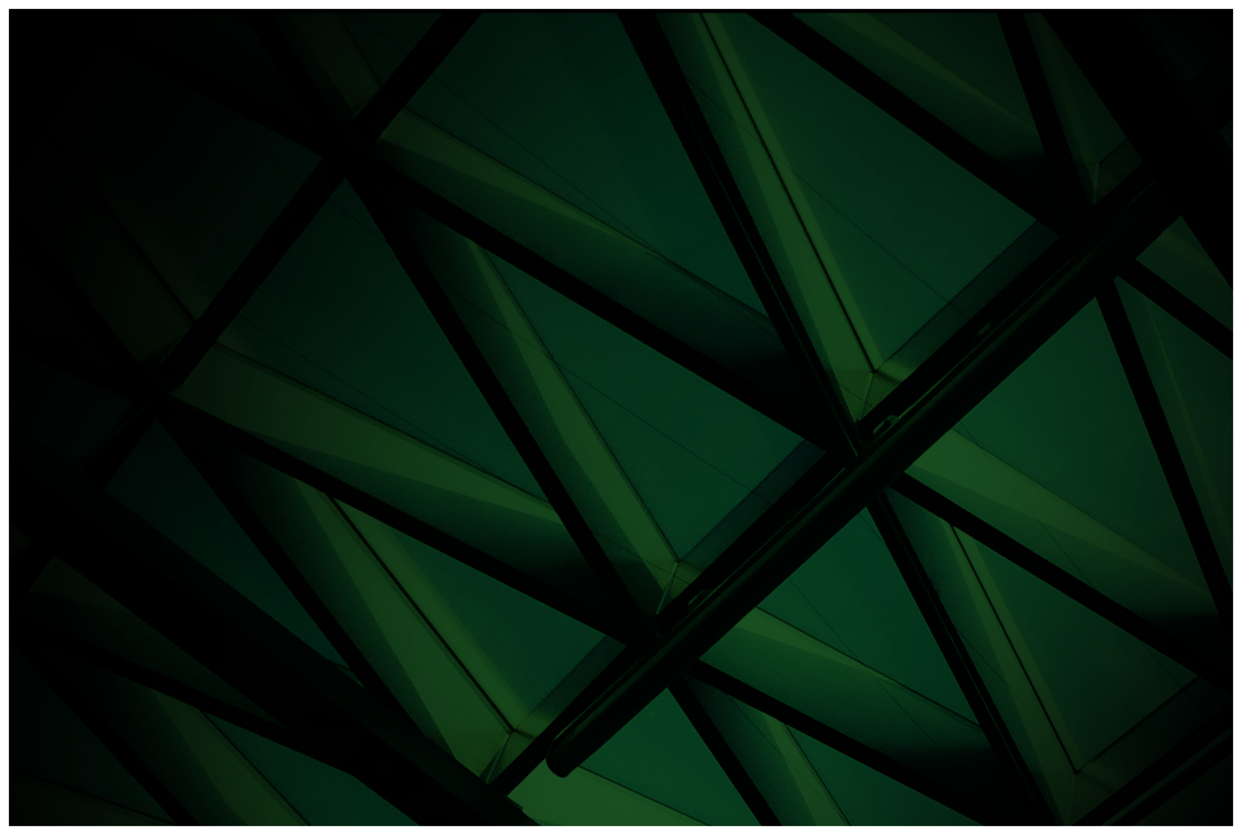


Erewhon Demonstration System Reference

VERSION 2022

Copyright©2024 Jade Software Corporation Limited. All rights reserved.

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2024 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the Jade **Readme.txt** file.

Contents

Introduction **7**

Part 1 Setting Up the Erewhon Demo System **8**

Batch Loading the Erewhon Schemas 9
Initializing the Erewhon Investments Database 9
Initializing the Database from the Command Line 9
Running the Administration Application (Standard Client) 10
Running the Shop Application (Standard Client) 10
Running the Tender Closure Application (Standard Client) 10
Running Jade in Thin Client Mode 10
Running the Web Shop Application using Apache HTTP Server 12
Running the Web Shop Using Internet Information Server 13
Configuring IIS 13
Step 1: Installing CGI and ISAPI Extensions 13
Step 2: Adding an Application Pool 14
Step 3: Adding an Application 16
Step 4: Configuring Handler Mappings for the Application 16
Step 5: Adding a Virtual Directory for Images 18
Running the Web Shop Application 19
Authorizing the WebShop Application for IIS 19

Part 2 User Guide **21**

Administration Application 21
Logon 21
Main Administration Window – File Menu 21
Main Administration Window – Edit Menu 21
Company Details 22
Agent Commission Rates (Company User Only) 23
Locations (Company User Only) 24
Sale Item Categories (Company User Only) 25
Main Administration Window – View Menu 26
Agents and Clients (Company User Only) 27
Commission Rates 27
Sale Items (by Category) 29
Sales 31
Jade Thin Client Shop Application 31
Logon 32
Product Search 32
Viewing the Details of a Product 33
Buying or Bidding for a Product 33
Shopping Cart 33
Product Details 34
Checkout 35
Web Shop Application 35
Logon 36
Product Search 37
Viewing the Details of a Product 37
Buying or Bidding for a Product 37
Product Details/Tender 38
Checkout 39
Tender Closure Application 39

Part 3 Model Implementation **41**

Locations 42
Agents and Commission Rates 43
Sales and Clients 44
Jade Reference Diagram 44
Agent 45

Client 45
Company 45
Commission Rate 46
Country 46
Region 46
Retail Sale 46
Tender Sale 46
Retail Sale Item 46
Tender Sale Item 46
Sale Item Category 47
Tender 47

Part 4 Design Considerations 48

Conventions 48
Models, Views, and Controllers 48
Model and View Separation 49
Schemas 49
- CommonSchema 50
- ErewhonInvestmentsModelSchema 50
- ErewhonInvestmentsViewSchema 50
- SelfDocumentorSchema 50
- WebServiceConsumer 51

Transaction Separation 51
Model Operations 51
Exception Handling 53
Cache Synchronization 54
listCollection 55
CollectionListBox Class 56
Object Notifications 56
Edition Checking 56
Synchronization of Shop Views 57
Locking 58
- Exclusive Locks 58
- Shared Locks 58
- Reserve Locks 59
- Unlocking Objects 59

Inverses and Referential Integrity 59
- One-to-One Relationships 60
- One-to-Many Relationships 60
- Many-to-Many Relationship 60
- Parent-Child Relationships 60
- Multiple Inverse Relationships 60
- Automatic Key Maintenance 60

Key Paths 61
Server Methods 61
Skins 62

Part 5 Transaction Agent Framework 63

Best Practice Guidelines 64
Transaction Agent Framework (TAF) Overview 64
- What is the Transaction Agent Framework (TAF)? 65
- Why is a TAF Needed? 66
- Where Should the TAF Reside? 66
- How Does the TAF Work? 67
  - Manually Persisting an Object 68
  - Creating Objects using the BaseForm Class 69
  - Updating Objects using the BaseForm Class 70
  - Deleting Objects using the BaseForm Class 71
  - Reading Data 71

Locking Objects 73
Locking a Collection 73
Locking Collection Objects Before a Create Action 73
Locking Collection Objects Before a Delete Action 73
Locking Collection Objects Before an Update Action 73
PersistentModel Class 73
PersistentModel Code Implementation Examples 75
PersistentModel Properties 75
PersistentModel Methods 76
getTAClass 76
isSoftLockedByMe 76
onCreate 77
onDelete 77
onModify 77
onUpdate 78
setCommonProperties 78
ModelTA Class (Transaction Agent) 79
ModelTA Class Diagrams 81
ModelTA Code Implementation Examples 83
ModelTA Properties 89
ModelTA Methods 90
addError 92
addWarning 93
checkEdition 94
clearErrors 94
clearErrorsOnSubordinateTAs 95
clearWarnings 95
clearWarningsOnSubordinateTAs 96
copyErrors 96
copyWarnings 97
createEntity 97
createEntityInTransState 98
createEntityWithTransactionImplementor 98
createSubordinateObjects 99
deleteEntity 100
deleteEntityInTransState 101
deleteEntityWithTransactionImplementor 101
deleteSubordinateObjects 102
doAbortTransactionCleanup 103
doAbortTransactionCleanupForSubordinateObjects 103
doCreate 104
doDelete 104
doModify 105
doPreValidate 105
doUpdate 106
doValidate 107
getFullErrorDetails 108
getModelObject 108
getModelObjectClass 109
hasErrors 109
hasNoErrors 110
hasOnlySubordinatePersistentObjects 110
initialize 111
lockForCreate 112
lockForDelete 113
lockForModify 114
lockForUpdate 114
modifyEntity 115
modifyEntityInTransState 115
modifyEntityWithTransactionImplementor 116
modifySubordinateObjects 117

persistEntity ... 118
persistEntityInTransState ... 119
populateFromObject ... 121
populateSubordinateObjects ... 121
tryLockingObject ... 122
updateEntity ... 123
updateEntityInTransState ... 123
updateEntityWithTransactionImplementor ... 124
updateSubordinateObjects ... 125
TransactionImplementor Class ... 126
TransactionImplementor Class Diagram ... 128
TransactionImplementor Abstract Class ... 128
doAbortTransaction (Abstract) ... 129
doBeginTransaction (Abstract) ... 129
doCommitTransaction (Abstract) ... 129
doIntermediateCommitIfDue ... 129
AbortTransactionTI Class ... 129
doAbortTransaction ... 130
doBeginTransaction ... 130
doCommitTransaction ... 131
CommitTransactionTI Class ... 131
doAbortTransaction ... 132
doBeginTransaction ... 132
doCommitTransaction ... 133
CommitTransactionOnIntervalTI Class ... 133
create ... 134
doIntermediateCommitIfDue ... 134
maxDurationMS ... 135
setMaxDurationMS ... 135
NoTransactionTI Class ... 135
doAbortTransaction ... 135
doBeginTransaction ... 136
doCommitTransaction ... 136
SubordinateTransactionTI Class ... 136
BaseForm Class ... 137
BaseForm Class Diagram ... 138
EditClientForm Code Implementation Examples ... 139
displayObject ... 139
getCurrentObject ... 140
getTA ... 140
getTAClass ... 140
populateTAFromForm ... 141
processAfterDelete ... 141
processAfterSave ... 142
BaseForm Properties ... 142
BaseForm Methods ... 142
displayErrors ... 143
displayObject ... 144
doDelete ... 145
doSave ... 146
formLoad ... 147
formUnload ... 148
getCurrentObject ... 149
getTA ... 149
getTAClass ... 149
populateTAFromForm ... 150
processAfterDelete ... 151
processAfterSave ... 151
setContextObject ... 152

Introduction

The Jade Platform Erewhon demonstration system is an Internet-enabled online purchasing and tendering application. It has been built to give you a good appreciation of Jade's features, including web deployment, Jade smart client technology, and web services. You can also look at the code to see how a Jade system is built.

This document provides you with all of the information you need to install and run the Erewhon demonstration system. It also looks at how some of Jade's key features are used.

This system is more than just a demonstration of Jade's capabilities. It is a resource that you can draw upon when building your own Jade applications. You can also use it for your own Jade presentations in the public arena.

The demonstration system has been created to enable an imaginary company called Erewhon Investments Inc. to trade internationally over the Internet.

Erewhon Investments Inc. is an online business specializing in the sale of high-value antiques, luxury homes, and luxury holidays. With some of the world's wealthiest individuals as clients, Erewhon Investments caters to a steadily growing niche market at the very top end. The company is based in New Zealand, yet operates in a global market place, with suppliers and customers all around the world.

Erewhon Investments needed a web-based merchandising system that could support their two real-time sale processes: retail purchases and a tendering process where clients can lodge date-constrained tenders to bid on an item.

The application had to have the reliability to accommodate the numerous Erewhon agents, each of whom has multiple items for sale, and provide facilities for these agents to access item and sale details from any location in the world, at any time of the day.

Added to this was the complication of different agents using varying commission rates – the system had to be able to define multiple commission rates per sale item category, with the ability for different rates to apply to different agents.

In terms of client user requirements, the system had to have a search capability, to enable clients to search the database and create a list of items filtered from some or all of the following criteria: region, category, tender or retail, and price range.

Having completed the search, the client then had to be able to step through the search results and view details about each of the items filtered. The details had to include the name, description, price details, and a photograph.

From there, the client had to be able to make a retail purchase or bid on a tender item. Both orders had to be added to the client's shopping cart for later confirmation. The shopping cart had to be able to maintain a current list of the items selected for client query, confirmation, or deletion. Confirmation of the shopping cart processes the transactions.

As both agents and clients had to be able to access the system from geographically widespread locations, the system had to provide both thin client and web deployment options.

Part 1 Setting Up the Erewhon Demo System

Before you install the Erewhon Investments demonstration system, make sure you have installed your Jade system.

The following describes how to load the Erewhon schemas from the development environment. This requires the **examples\erewhon** folder to be local or visible via a file share from the machine on which you are running the development environment. If this is not the case, you can get the latest version of the Erewhon files from https://github.com/jadesoftwarenz/JADE-Erewhon, by following the instructions in the **README.md** Markdown language document. If you do not want to use the Jade Platform development environment to load the **Erewhon** schemas, you can load them in batch mode. For details, see "Batch Loading the Erewhon Schemas", later in this document.

1. From the **Schema** menu, select **Load**.

2. In the Load Options dialog, check the **Load Multiple Schemas** check box.

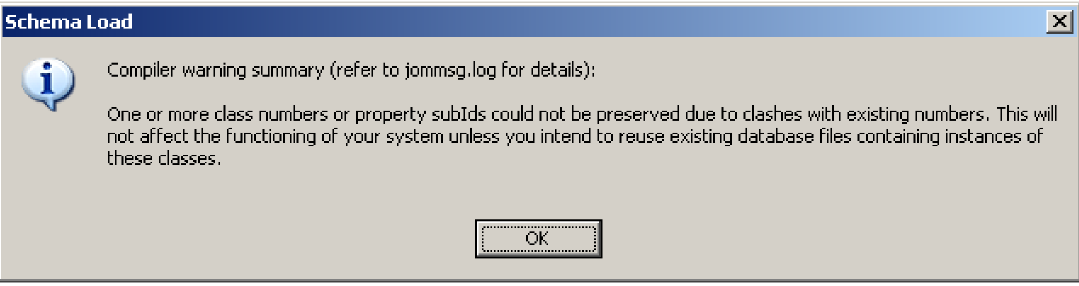
3. Click the **Browse** button and find folder containing the **Erewhon** schema files. Select the **ErewhonInvestments.mul** file in this folder and then click **Open**.

4. In the Load Options dialog, click **OK** to load the demonstration system, or click **Cancel** to return to the main window.

   The following schemas are loaded into your environment.

   - **CommonSchema**
   - **ErewhonInvestmentsModelSchema**
   - **ErewhonInvestmentsViewSchema**
   - **SelfDocumentorSchema**
   - **WebServiceConsumer**.

   If a message box about class numbers and property subIds (shown in the following image) is displayed, click **OK** to ignore it.



Batch Loading the Erewhon Schemas

The following describes how to batch-load the Erewhon schemas. Ensure that any Jade database and application servers for this system are shut down before proceeding.

1. Start a command line session on the host where your database is located.
2. Change to the ***<install-dir>*** folder, where *<install-dir>* is the folder in which your Jade system is installed.
3. Enter the following command, where the *Erewhon-dir* value is the folder containing the **Erewhon** schema files.

```
bin\jadloadb ini=<install-dir>\system\jade.ini path=<install-dir>\system
scmFile=Erewhon-dir\ErewhonInvestments.mul
```

Initializing the Erewhon Investments Database

The following describes how to initialize the Erewhon Investments database from the development environment. This requires the **\erewhon** folder to be local or visible via a file share from the machine on which you are running the development environment. If you want to initialize the database from the command line, see "Initializing the Database from the Command Line", later in this document.

1. In the Schema Browser window, select **ErewhonInvestmentsModelSchema**.
2. From the **Browse** menu, select **Classes**.
3. In the Class Browser, select the **JadeScript** class in the top-left panel of the window.
4. In the top-right pane of the window, scroll down the list of methods and then select the **initializeData** method.
5. From the **Jade** menu, select **Execute It** (or alternatively, press F9). This will run the method.
6. In the Browse for Folder form, find the **DataFiles** folder in the ***<install-dir>*\examples\erewhon** folder (where *<install-dir>* is the folder where you installed your Jade system). Select the **DataFiles** folder and then click **OK**. Click **Cancel** to return to the Class Browser window.
7. Progress messages are displayed in the Jade Interpreter Output Viewer Window during the database initialization. When the load completes, **Database initialized** is displayed in this window and **Execution complete** is displayed in the Jade status line. Select **Exit** from the **File** menu of the Jade Interpreter Output Viewer to close it.

The Erewhon demonstration system database is now initialized.

Initializing the Database from the Command Line

The following describes how to initialize the Erewhon Investments database from the command line. Ensure that any Jade database and application servers for this system are shut down before proceeding.

1. Start a command line session on the host where your database is located.
2. Change to the ***<install-dir>*** folder, where *<install-dir>* is the folder in which your Jade system is installed.
3. Enter the following command, where the *Erewhon-dir* value is the folder containing the **Erewhon** schema files.

```
bin\jadclient server=singleUser ini=<install-dir>\system\jade.ini
path=<install-dir>\system schema= ErewhonInvestmentsModelSchema
app=DataLoader startAppParameters Erewhon-dir\DataFiles
```

4. The database will now be initialized. You will see progress messages and when the load completes, a **Database initialized** message.

Running the Administration Application (Standard Client)

Select the Schema Browser window. If this window is not visible, select **Schema Browser** from the Window menu to set focus to it.

1. In the Schema Browser window, select **ErewhonInvestmentsViewSchema**.
2. Click the **Run Application** toolbar button (this is the one with the arrow icon).
3. Select **Administration** in the **Application Name** combo box. Click **OK** to start this application.
4. Select a name from the **User Name** combo box. The Company Administrator is **Erewhon Investments Inc.**
5. Click **OK** to run the application.

For information about using the application, see "Part 2 – User Guide", later in this document.

Running the Shop Application (Standard Client)

1. Select the Schema Browser window. If this window is not visible, select **Schema Browser** from the Window menu to set focus to it.
2. In the Schema Browser window, select the **ErewhonInvestmentsViewSchema**.
3. Click the **Run Application** toolbar button (this is the one with the arrow icon).
4. Select **ErewhonShop** in the **Application Name** combo box. Click **OK** to run this application.
5. Select a name from the **User Name** combo box.
6. Click **OK** to run the application.

For information about using the application, see "Part 2 – User Guide", later in this document.

Running the Tender Closure Application (Standard Client)

1. Select the Schema Browser window. If this window is not visible, select **Schema Browser** from the Window menu to set focus to it.
2. In the Schema Browser window, select **ErewhonInvestmentsViewSchema**.
3. Click the **Run Application** toolbar button (this is the one with the arrow icon).
4. Select **TenderClosureApp** in the **Application Name** combo box.
5. Click **OK** to run the application.

For information about using the application, see "Part 2 – User Guide", later in this document.

Running Jade in Thin Client Mode

Note The Jade thin client communicates with the application server via TCP/IP. You must have TCP/IP installed and configured to use the Jade thin client.

To run Jade in thin client mode, you must create two shortcuts: one for the application server, and one for the thin client.

1. For the application server, create a shortcut with the following properties.

   Target:

   ```
   <install-dir>\bin\jadapp.exe
   path=<install-dir>\system
   server=singleUser
   appserverport=60000
   ini=<install-dir>\system\jade.ini
   ```

   Start In:

   ```
   <install-dir>\bin
   ```

   The <***install-dir***> value is the folder in which you installed your Jade system.

2. For the Jade thin client, create a shortcut with the following properties.

   Target:

   ```
   <install-dir>\bin\jade.exe
   schema=JadeSchema
   app=Jade
   appserver=<computer-name>
   appserverport=60000
   ```

   Start In:

   ```
   <install-dir>\bin
   ```

   The <***install-dir***> value is the folder in which you installed your Jade system and <***computer-name***> is the name or IP address of your computer (or **localhost**, or the loop-back IP address 127.0.0.1). To find your computer name or IP address, open a the **Network and Sharing Center** in Control Panel and then click **Local Area Connection** to view the IP address.

3. You can now run Jade in thin client mode.

   Note Ensure that you have shut down any open Jade sessions before running the application server. This includes the Jade Platform development environment.

4. Start the application server from the application server shortcut. The application server will start and is now waiting for thin client connections.

5. Run Jade as a thin client of the application server from the thin client shortcut. The connection is displayed in the application server window. Log on to Jade as usual, and initiate applications using the standard client instructions in an earlier section. Notice that the interface presented under the Jade thin client is identical to the Jade standard client.

For information about using the application, see "Part 2 – User Guide", later in this document.

Running the Web Shop Application using Apache HTTP Server

See the *Installation and Configuration Guide* (which is also available from the Jade website at https://www.jadeworld.com/jade-platform/developer-centre/learn/documentation) for information about deploying Jade web applications.

1. Before running the web shop application, you must install the Jade HTTP driver for Apache, and define a virtual directory for Jade in your Apache configuration files. Copy the Windows version of the **mod_jadehttp.so** file to the Apache modules folder. Now edit the **conf/httpd.conf** file and at the end of the **Dynamic Shared Object (DSO) Support** section, add the following lines.

```
LoadModule jadehttp_module modules/mod_jadehttp.so
<IfModule mod_jadehttp.c>
    Include conf/jadehttp.conf
</IfModule>
```

   Now create a file called **conf/jadehttp.conf** with the following details.

```
<IfModule mod_jadehttp.c>
    <Location /jade-info>
        SetHandler jadehttp-info
    </Location>
    <Location /JadeEval>
        SetHandler jadehttp-handler
        Application WebShop
        TcpConnection 127.0.0.1 6107
    </Location>
</IfModule>

<Directory>
    require from all
</Directory>
Alias /images "C:\Temp"
```

   Create the **C:\Temp** folder on your machine if it does not already exist, and then restart your Apache HTTP Server.

   Note Ensure that you have shut down any open Jade sessions before proceeding further. This includes the Jade Platform development environment and the Jade application server.

2. Click the **Jade** icon from your Jade program folder in the Start menu. This will run the Jade Platform development environment in single-user mode.

3. Log on, and select **ErewhonInvestmentsViewSchema** in the Schema Browser window.

4. Click the **Run Application** toolbar button (this is the one with the arrow icon).

5. Select **WebShop** in the **Application Name** combo box.

6. Click **OK** to run the application.

7. The **WebShop** application will start and will appear as a single window. The application is now waiting for users to connect from a browser. If a warning message is displayed, advising you that you are running a web application with an invalid web working directory, you should click the **No** button on the message box, create the **C:\Temp** folder on your machine as stated in step 1 of this instruction, and then try again.

8. Bring up your Internet browser and then enter the following URL.

    `<computer-name>/JadeEval?WebShop`

    The <***computer-name***> value is the name or IP address of your machine (or **localhost** or the loop-back IP address 127.0.0.1), and **JadeEval** is the name of the virtual directory you created in your web server; for example, a URL might be one of the following.

    - erewhon/JadeEval?WebShop
    - 192.168.1.100/JadeEval?WebShop
    - localhost/JadeEval?WebShop
    - 127.0.0.1/JadeEval?WebShop

Running the Web Shop Using Internet Information Server

See the *Installation and Configuration Guide* (which is also available from the Jade website at https://www.jadeworld.com/jade-platform/developer-centre/learn/documentation) for information about deploying Jade web applications.

The following subsections contain instructions for running the **WebShop** application using Microsoft Internet Information Server (IIS).

- Configuring IIS
- Running the WebShop application
- Authorizing the WebShop application

Configuring IIS

The configuration instructions are grouped into the following subsections.

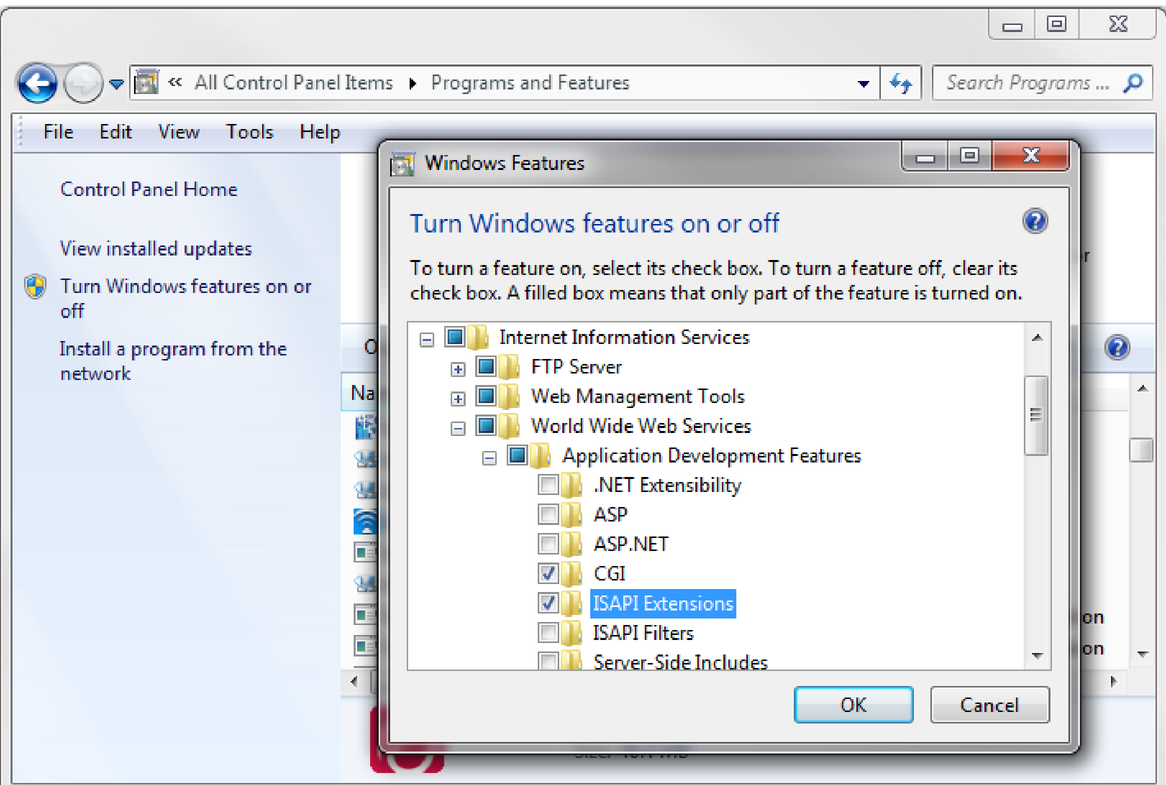
- Check that important IIS components are installed
- Add and configure an application (and application pool) for the **WebShop** application
- Add a virtual directory for **WebShop** image files

Step 1: Installing CGI and ISAPI Extensions

To install these optional components of IIS:

1. Select **Programs and Features** from the Control Panel.
2. Click the **Turn Windows features on or off** hyperlink on the left.
3. Expand **Internet Information Services**, then **World Wide Web Services**, and then **Application Development Features**.

4. Check the boxes **CGI** and **ISAPI Extensions** and then click the **OK** button.

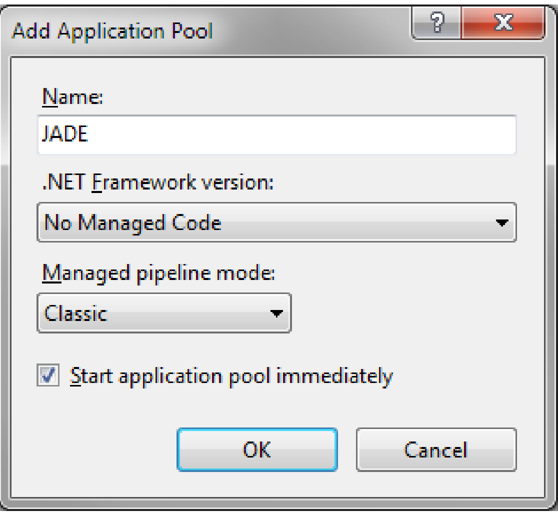


Step 2: Adding an Application Pool

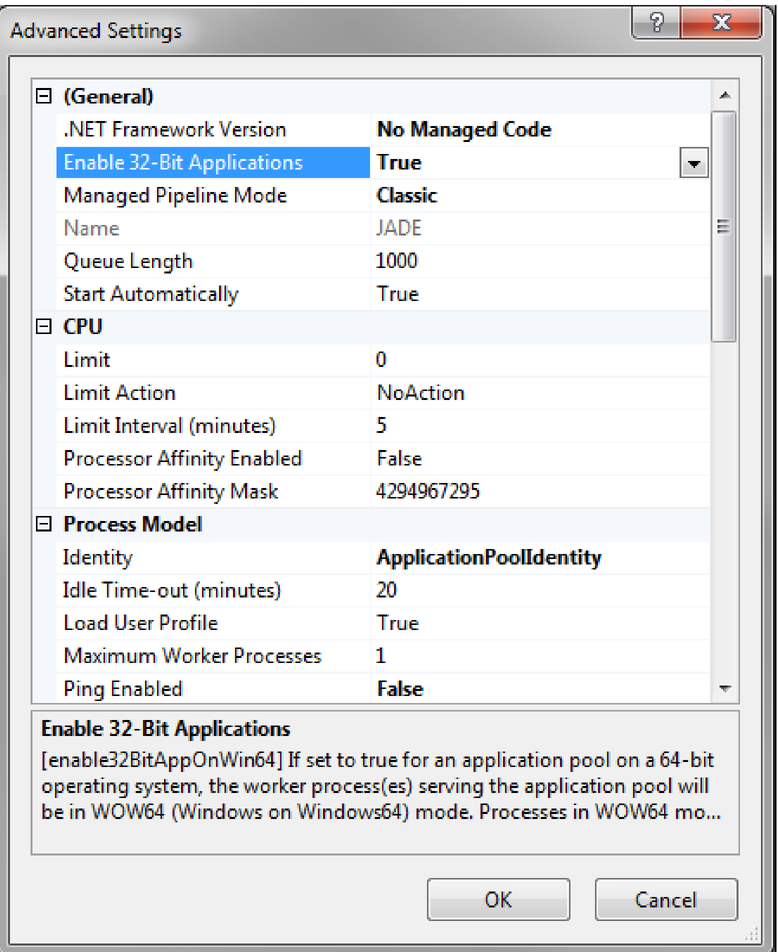
To add an application pool to be used by Jade web applications:

1. Select **Administrative Tools** from the Control Panel.
2. Open **Internet Information Services (IIS) Manager**.
3. In the **Connections** panel on the left, select **Application Pools**.
4. Right-click and then select **Add Application Pool**.

5. Configure the pool to use unmanaged (non-.NET) code and then set the **Managed pipeline mode** field to **Classic**, as shown in the following image.



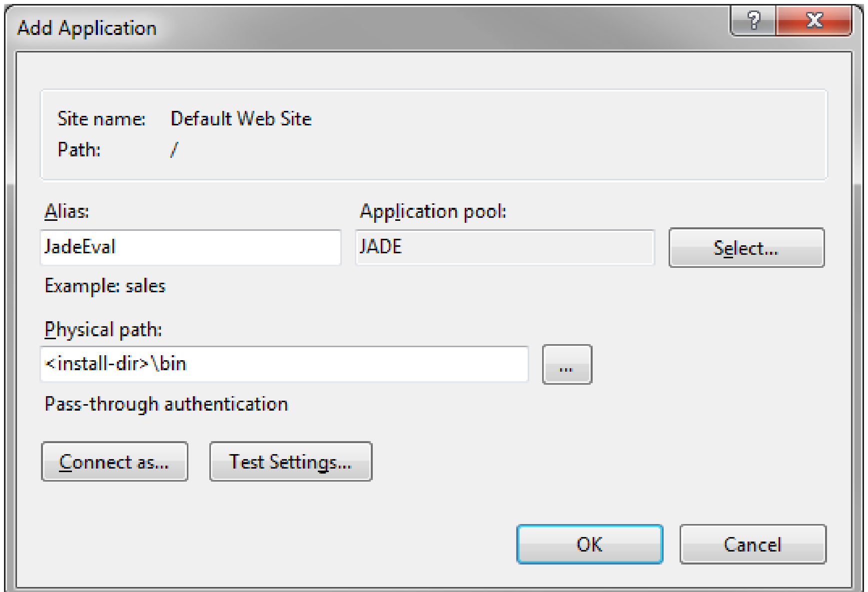
6. If you are using a 32-bit version of Jade, right-click on the application pool and then select **Advanced Settings**. In the Advanced Settings dialog, set **Enable 32-Bit Applications** to **True**.



Step 3: Adding an Application

To add an application:

1. Select the **Default Web Site** in the **Connections** panel.
2. Right-click and then select **Add Application**.
3. Enter the alias **JadeEval**. (This will be part of the URL for the **WebShop** application.)
4. Select the application pool that you created previously.
5. Enter the location of the **Physical path**, which is the **bin** folder for your Jade release; that is, replace **<install-dir>** in the following image with the correct location.



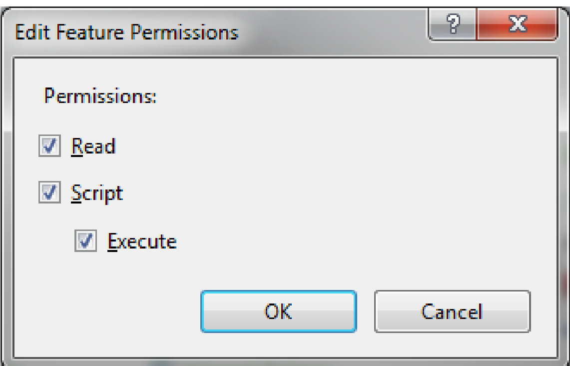
6. Click the **OK** button.

Step 4: Configuring Handler Mappings for the Application

To configure handler mappings for the application:

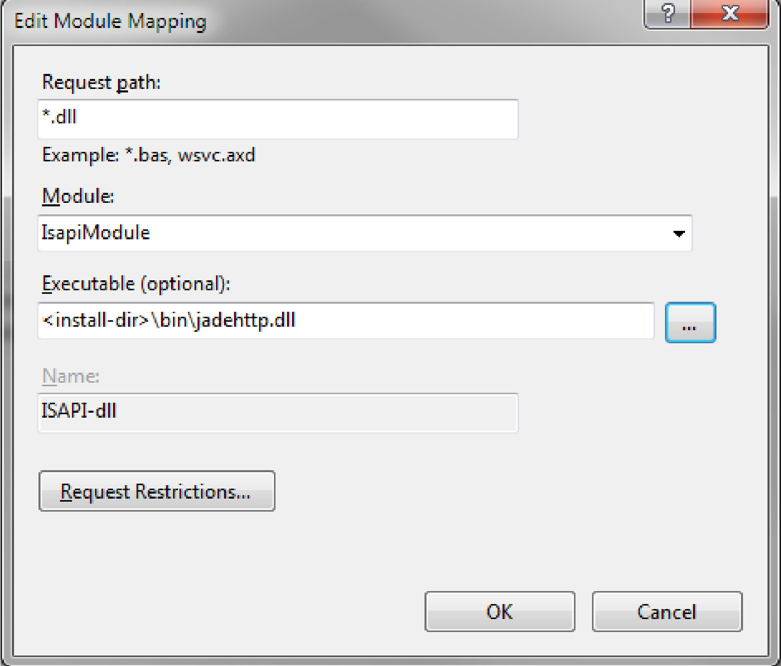
1. Select the application in the **Connections** panel.
2. Double-click the **Handler Mappings** icon in the central panel.
3. Right-click the **CGI-exe** handler mapping and select **Edit Feature Permissions**.

4. Enable all options, as shown in the following image.

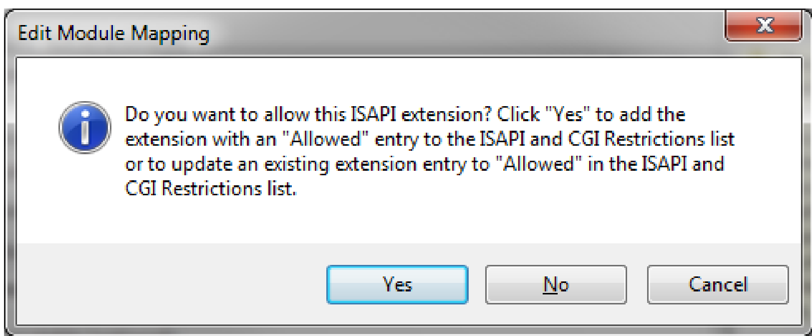


5. Right-click the **ISAPI-dll** handler mapping and then select **Edit**.

6. Set the **Executable** text box to the path and file name of the **jadehttp.dll** file in the **bin** folder of your Jade system.



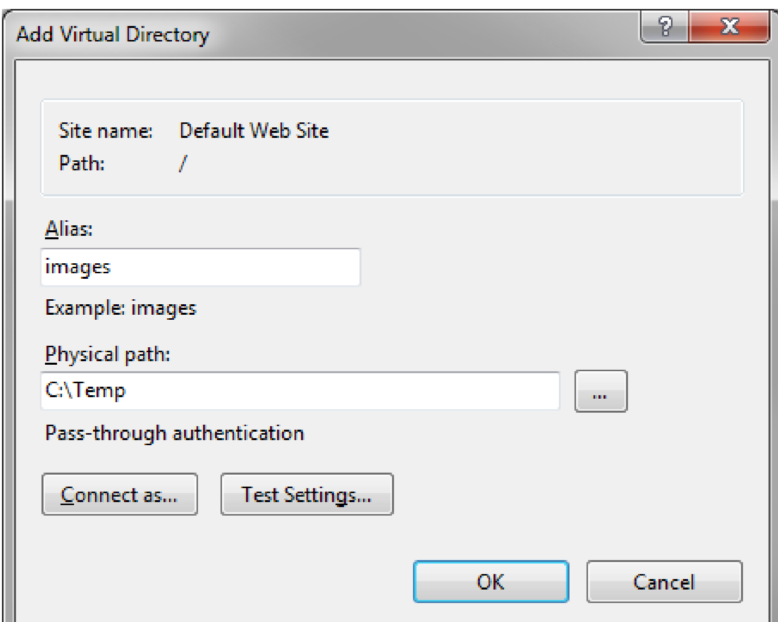
7. If the following dialog is displayed, click the **Yes** button.



Step 5: Adding a Virtual Directory for Images

To add a virtual directory for images:

1. Ensure that a **C:\Temp** folder exists on your machine.
2. Select **Default Web Site** in the **Connections** panel.
3. Right-click and then select **Add Virtual Directory**.
4. Complete the dialog as shown in the following image and then click the **OK** button.



5. Now start IIS for your website.

Note Ensure that you have shut down any open Jade sessions before proceeding further. This includes the Jade Platform development environment and the Jade application server.

Running the Web Shop Application

With the IIS configuration completed, you can now attempt to run the **WebShop** application.

1. Click the **Jade** icon from your Jade program folder in the Start menu. This will run the Jade Platform development environment in single-user mode.
2. Log on and in the Schema Browser window, select **ErewhonInvestmentsViewSchema**.
3. Click the **Run Application** toolbar button (this is the one with the arrow icon).
4. Select **WebShop** in the **Application Name** combo box.
5. Click **OK** to run the application, or **Cancel** to return to the main window.
6. The **WebShop** application will start and will appear as a single window. The application is now waiting for users to connect from a browser.
7. Open your Internet browser and then enter the following URL.

   ```
   <computer-name>/JadeEval/jadehttp.dll?WebShop
   ```

   The <***computer-name***> value is the name or IP address of your machine (or **localhost** or the loop-back IP address 127.0.0.1), and **JadeEval** is the name of the virtual directory you created in your web server; for example, a URL could be one of the following.

   - erewhon/JadeEval/jadehttp.dll?WebShop
   - 192.168.1.100/JadeEval/jadehttp.dll?WebShop
   - localhost/JadeEval/jadehttp.dll?WebShop
   - 127.0.0.1/JadeEval/jadehttp.dll?WebShop

8. At this point, Jade's security features that protect against unauthorized running of applications via a web browser will result in an error message (*Service unavailable* or similar) being displayed in your web browser.

We must now authorize the **WebShop** application for IIS. For details, see the following section.

Authorizing the WebShop Application for IIS

We now need to specifically allow the **WebShop** application to be accessed via IIS from a web browser.

Using Windows Explorer, you will find that Jade has created an additional four folders, as follows.

- <*install-dir*>\bin_jadehttp
- <*install-dir*>\bin_jadehttp\ini
- <*install-dir*>\bin_jadehttp\logs
- <*install-dir*>\bin_jadehttp\transfer

If these folders have not been created, ensure that IIS is running for your website. If it isn't, start it and then select the refresh option in your web browser.

1. Using Notepad, open the **<*install-dir*>\bin_jadehttp\ini\jadehttp.ini** file.

   The <***install-dir***> value is the folder in which you installed your Jade system.

2. Add the following lines at the end of the file.

```
[WebShop]
TcpConnection=127.0.0.1
TcpPort=6107
ConnectionGroup=WebShopForms
MinInUse=1
MaxInUse=1
CloseDelay=600
ApplicationType=WebEnabledForms
```

   Save the file and then close Notepad.

3. Now return to your web browser window and select the refresh option.

   You should now be presented with the **WebShop** logon form, at which point you can log on and run the application.

Part 2 User Guide

The Jade Platform Erewhon Demo System consists of an **Administration** application (for Erewhon company and agent users) and a **Shop** application (for clients).

The **Administration** application is delivered as either a Jade standard client or a Jade thin client. The **Shop** application comprises two applications: one delivered as a Jade standard client or a Jade thin client, and the other as a Jade HTML thin client. For more details, see the following subsections.

Administration Application

The **Administration** application allows company and agent users to maintain core system data. For details, see the following subsections.

- Logon
- Main Administration Window - File Menu
- Main Administration Window - Edit Menu
- Main Administration Window - View Menu

Logon

To log on, select a user name from the drop-down list box on the Logon form then and click **OK**.

Note The list of user names includes the name of the **Erewhon Investments Inc.** company user and the names of all agents.

To access the **Administration** application's full range of functionality and data, select the **Erewhon Investments Inc.** user. Selecting an agent user name will mean that a limited subset of the **Administration** application's functionality and data will be available.

Main Administration Window – File Menu

To exit from the application, select the File menu **Exit** command (Alt+F, X), or press Alt+F4.

To view copyright and version details, select the File menu **About** command (Alt+F, A).

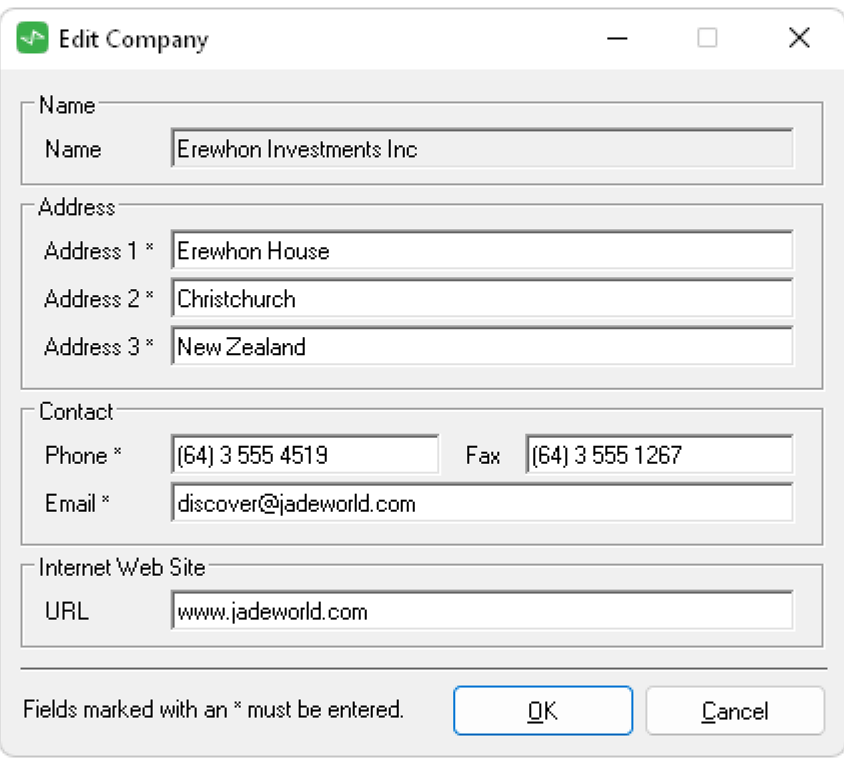
Main Administration Window – Edit Menu

This section contains the following topics.

- Company Details
- Agent Commission Rates (Company User Only)
- Locations (Company User Only)
- Sale Item Categories (Company User Only)

Company Details

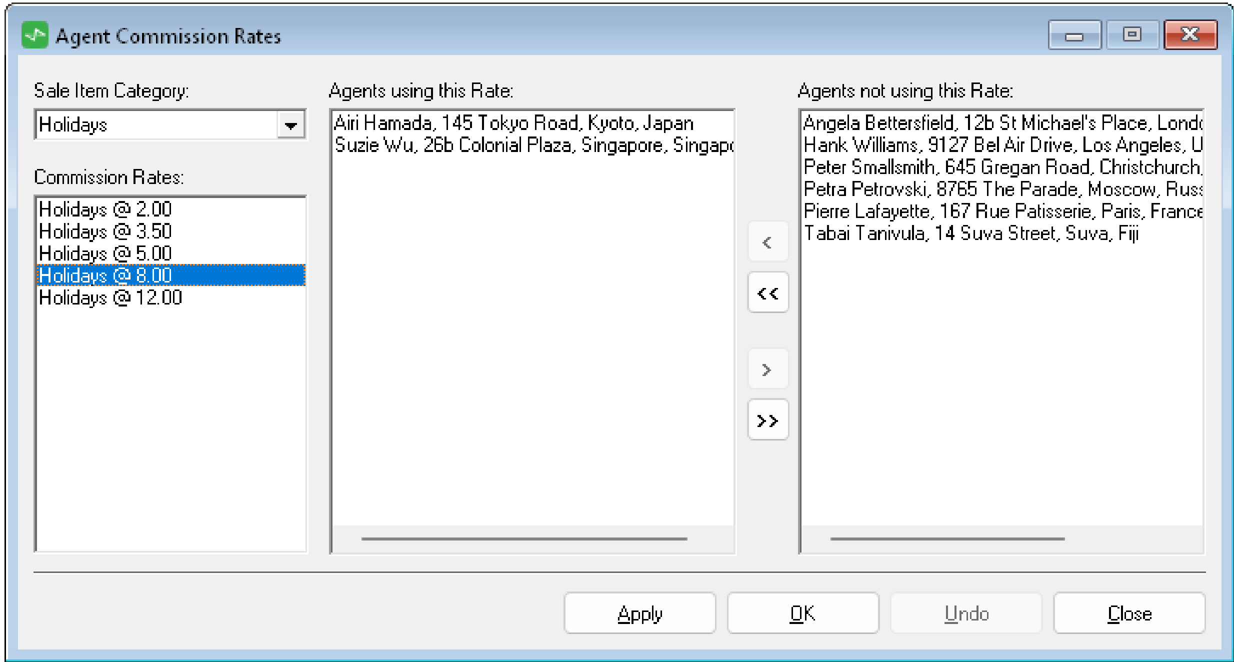
Use the Edit Company screen, shown in the following image, to maintain company user details.



To edit the company details, select **Edit Company Details** (Alt+E, D) from the Edit menu. Make changes as necessary and then click the **OK** button to update the database.

Agent Commission Rates (Company User Only)

Use the Agent Commission Rates screen, shown in the following image, to maintain a list of agents for each sale item's range of commission rates. (Note that an agent can use one rate only per sale item.)

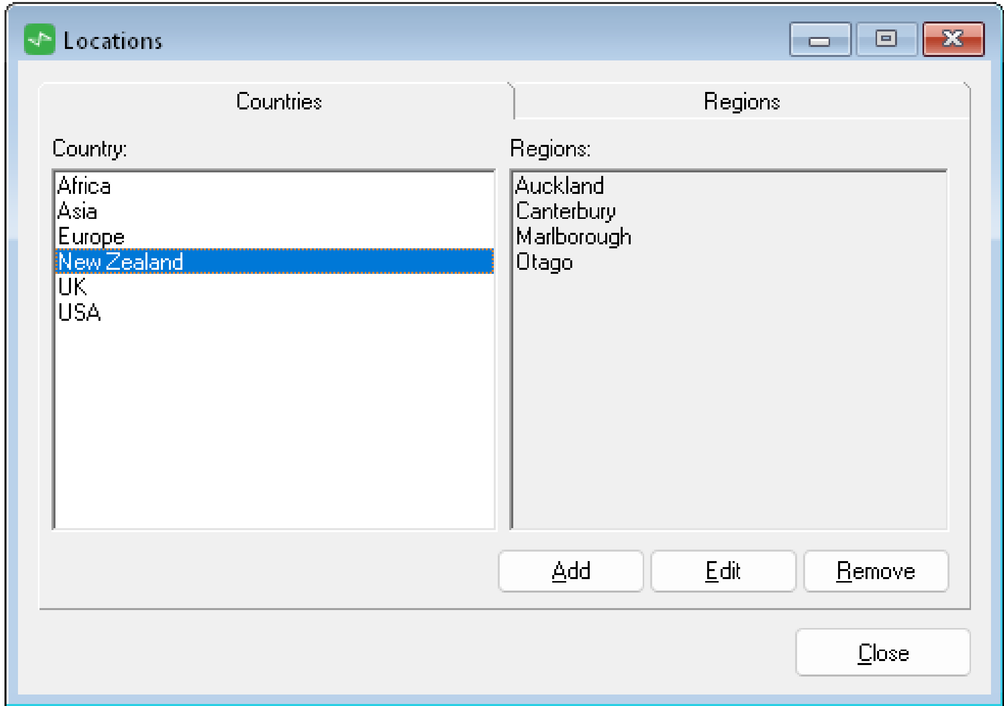


To modify the list of agents for a commission rate:

1. After opening the form, select a sale item category. The list of commission rates for that category is then displayed.
2. Select a commission rate from the list. This will populate two lists on the right: a list of all agents using the selected commission rate, and a list of all agents who do not use the selected commission rate.
3. To remove an agent from the list of those who use the selected commission rate, select the agent from the list on the left and then click the **>** button.
4. To remove all agents from the list of those who use the selected commission rate, click the **>>** button.
5. To add an agent to the list of those who use the selected commission rate, select the agent from the list on the right and then click the **<** button.
6. To add all agents to the list of those who use the selected commission rate, click the **<<** button.
7. Once you are satisfied with the list selections, click the **Apply** button to update the database or click the **OK** button to update the database *and* close the form. If you do not want to save the changes, click the **Undo** button.
8. To exit the form, without changing details, click **Close**.

Locations (Company User Only)

Use the Locations screen, shown in the following image, to maintain country and region details. Note that country is used to define a *macro* geographic area and region is used to define an associated *micro* geographic area. A form with a folder showing two tabs is then displayed. (Note that a region must always belong to a country.)

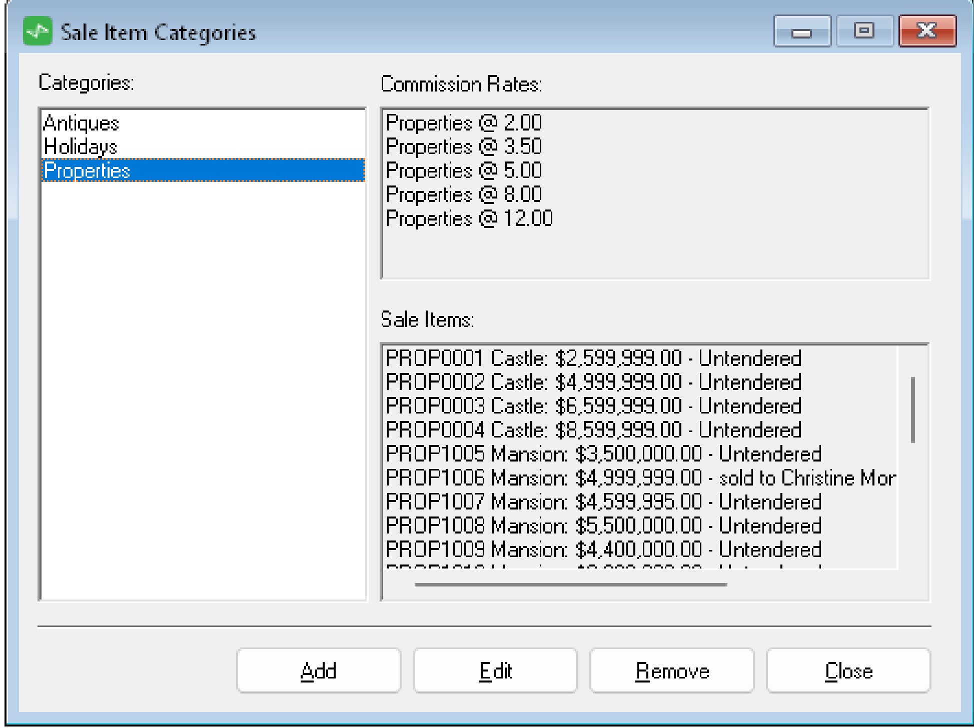


There are **Add**, **Edit**, and **Remove** buttons on both the **Countries** and **Regions** sheets.

- To add a new country or region, select the appropriate tab and then click the **Add** button. A form is then displayed, in which to enter the name of the location. Type the name and then click the **OK** button to add the location to the database.
- To change a country or region, select the appropriate tab and then click the **Edit** button. A form is then displayed, showing the current name. Change the name and then click the **OK** button to update the database.
- To remove a country or region, select the appropriate item and then click the **Remove** button. Note that when a country is removed, all of its regions are also removed.

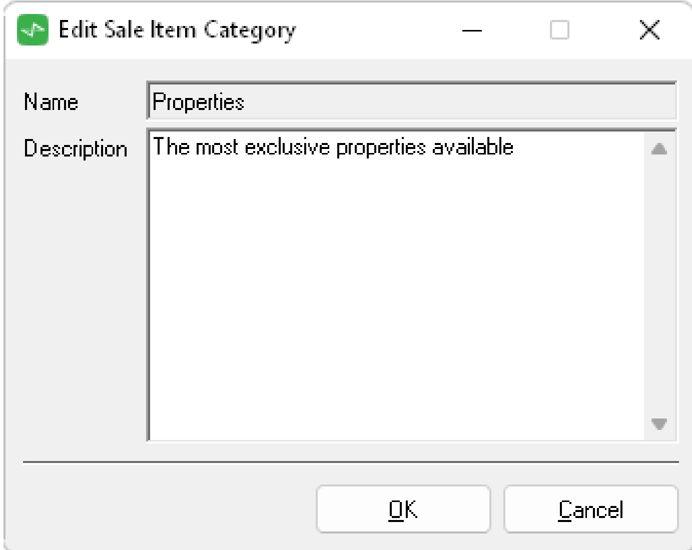
Sale Item Categories (Company User Only)

Selecting the Edit menu **Edit Sale Item** command displays the Sale Items Categories form with a list of sale item categories. The right-hand panes display the sale item categories associated commission rates and sale items.



- To add a new category, click the **Add** button. A form will then be displayed, in which to enter the name and description of the category. Type the name and description, then click the **OK** button to add the category to the database.

- To change a category, click the **Edit** button. A form will then be displayed, showing the current name and description. Change the description and then click the **OK** button to update the database (note that the name of the category cannot be changed).



- To remove a category, click the **Remove** button. Note that when a category is removed, all of its commission rates will also be removed.

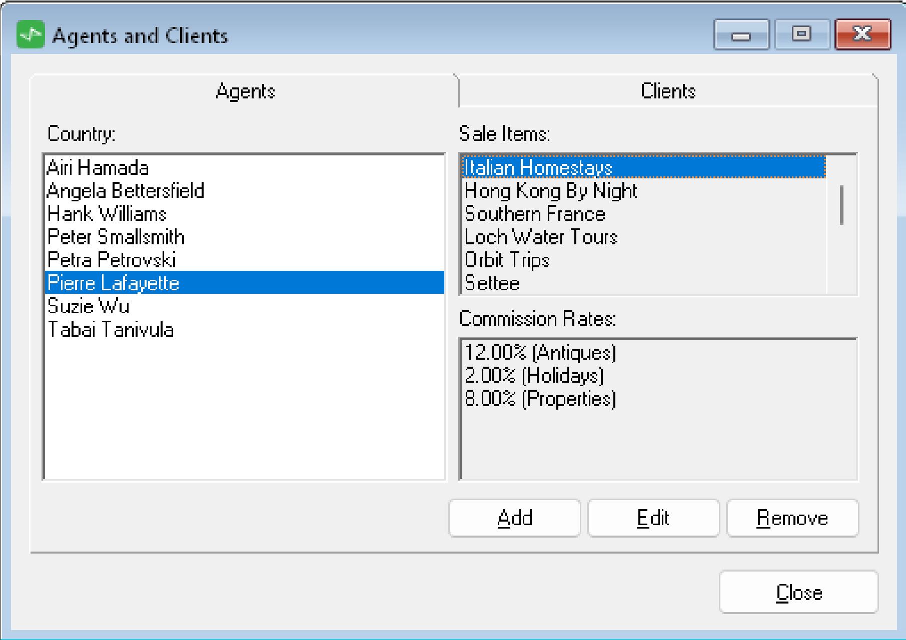
Main Administration Window – View Menu

This section contains the following topics.

- Agents and Clients (Company User Only)
- Commission Rates
- Sale Items (by Category)
- Sales

Agents and Clients (Company User Only)

Use the Agents and Clients screen to display a form with a folder containing two tabs: one for a list of agents and the other for a list of clients.



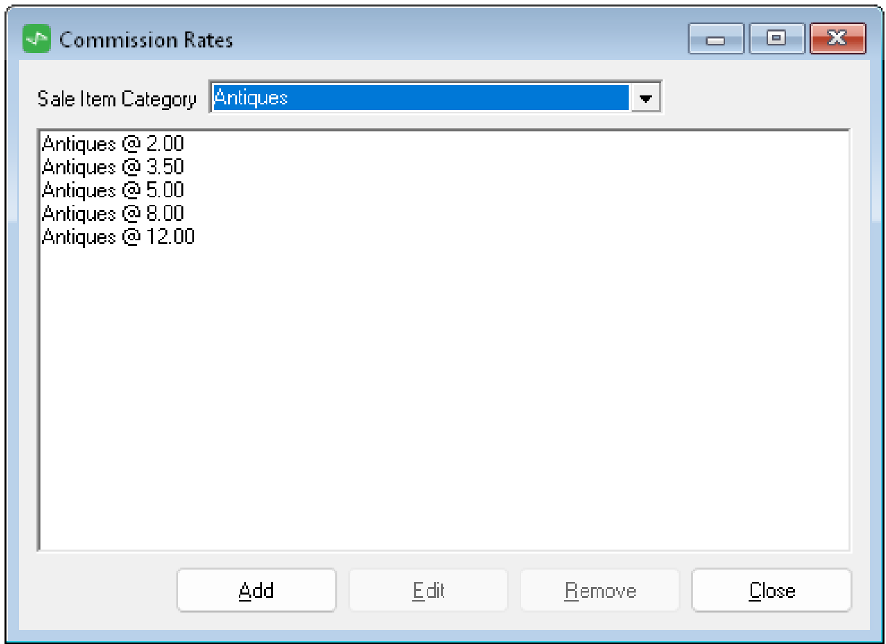
On each sheet there are **Add**, **Edit**, and **Remove** buttons.

- To add a new agent or client, select the required tab and then click the **Add** button. A form will be displayed, in which to enter the agent's or client's details. Enter the details and then click the **OK** button to add the agent or client to the database.
- To change an agent's or client's details, select the required tab and then click the **Edit** button. A form will be displayed, showing the details. Make any changes and then click the **OK** button to update the database.
- To remove an agent or client, select the required tab and then click the **Remove** button. Note that when a client is removed, all of its retail sales, tender sales, and outstanding tenders will also be removed.

Commission Rates

Use the Commission Rates screen to add or maintain variable commission rates for each sale item category.

A form displays with a drop-down list of sale item categories. Upon selecting a sale item category, a list is displayed of all commission rates for that category.



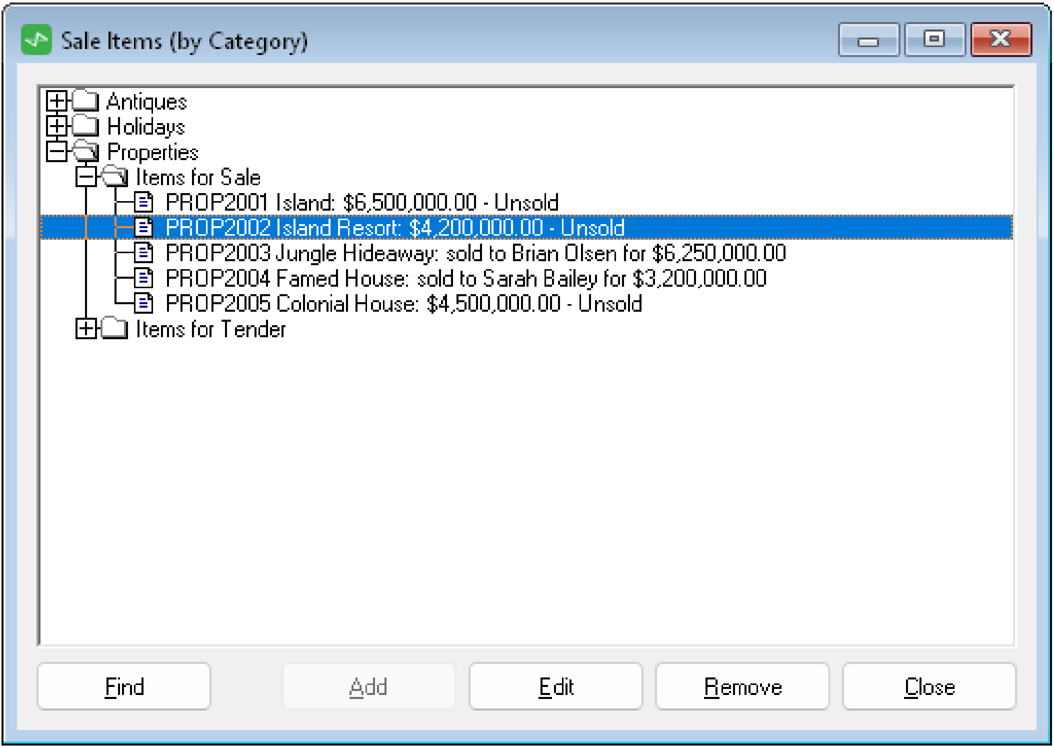
This form also has the following maintenance capabilities (company user only).

- To add a new commission rate, select your required sale item category from the drop-down list box and then click the **Add** button. A form will be displayed, in which to select the rate's category and enter the rate percentage. Enter the category and percentage, and then click the **OK** button to add the new commission rate to the database.
- To change a commission rate, select your required sale item category from the drop-down list box and the required commission rate and then click the **Edit** button. A form will be displayed, showing the current sale item category and rate percentage. Change the details to met your requirements and then click the **OK** button to update the database.
- To remove a commission rate, select it and then click the **Remove** button.

Sale Items (by Category)

Use the Sale Items (by Category) screen to add or maintain sale items. Note that only agents can add a new sale item. This screen employs a hierarchical tree of sale item categories, which can be expanded by clicking the **+** icon or collapsed by clicking the **-** icon.

As each sale item category's folder is opened, two subfolders are shown: one for the **Items for Sale**, and the other for the **Items for Tender**. Opening either of the subfolders will display a list of their respective sale items.



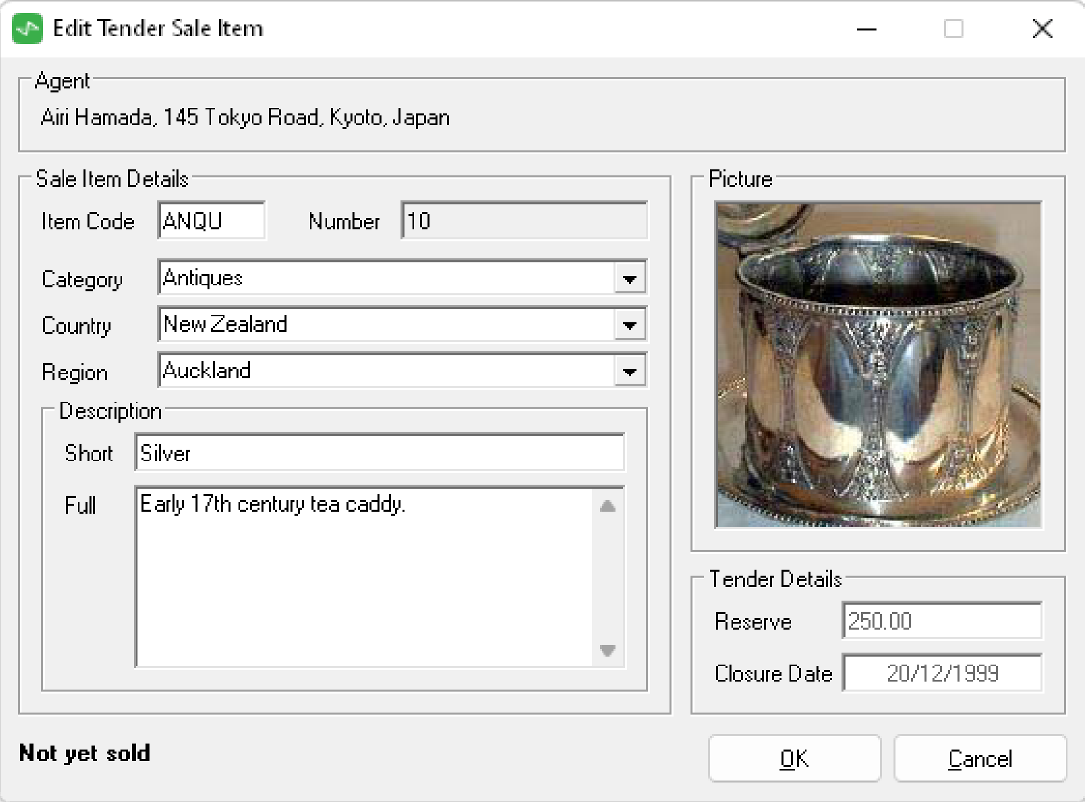
This form also has the following maintenance capabilities.

- To add a new sale item, open the folder of the sale item category to which the new sale item will belong, select the **Items for Sale** or **Items for Tender** subfolder, and then click the **Add** button (note that the **Add** button can also be clicked if a sale item leaf is selected).

  A form will be displayed, in which to enter the details of the sale item (the sale item category will have been preselected).

  To load a photo of the sale item from a disk file, double-click the empty picture frame. Enter the remainder of the details and then click the **OK** button to add the sale item to the database. Note that only agents can add sale items.

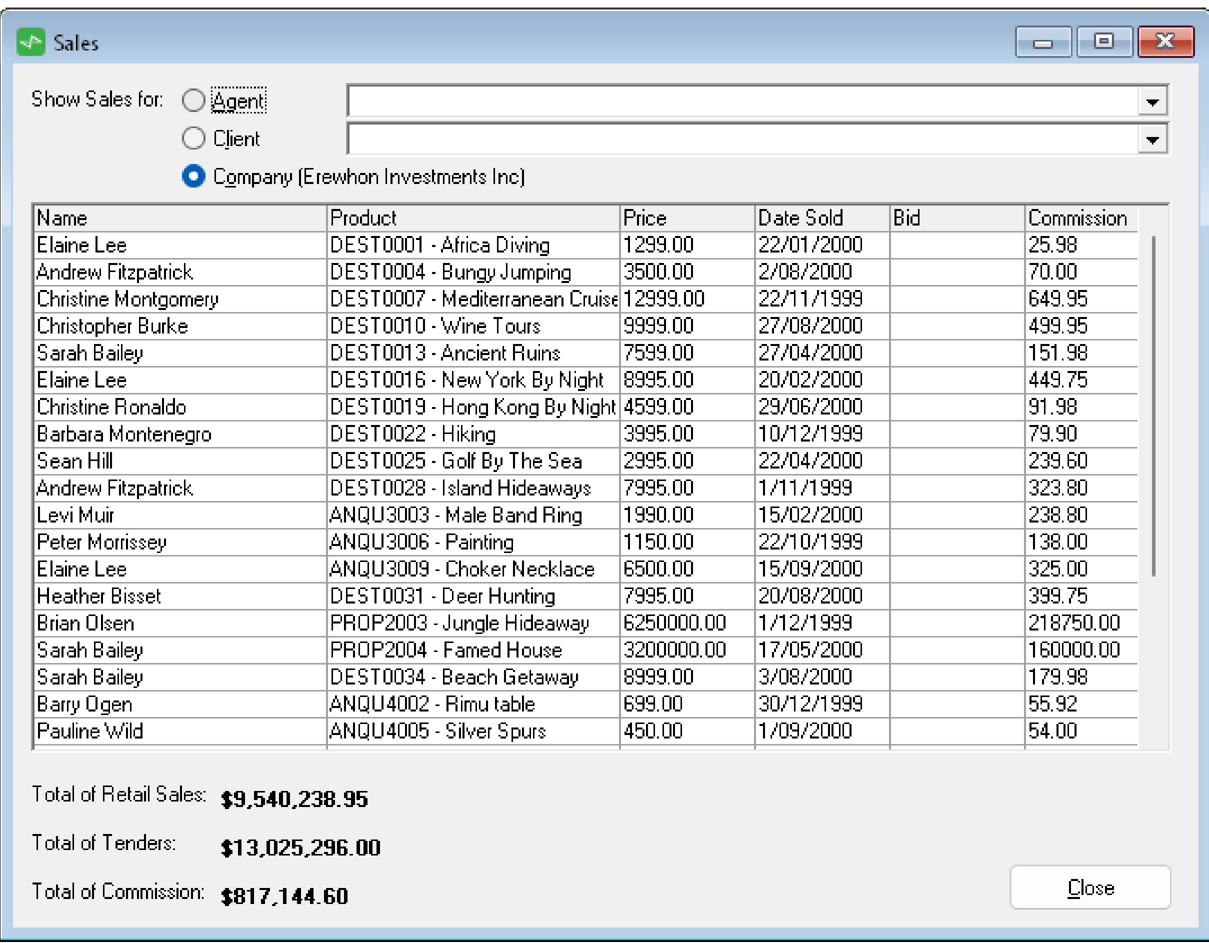
- To change a sale item, open the required sale item category folder and subfolder. Next, select the sale item that you want to change, and then click the **Edit** button. A form will be displayed, showing the current sale item details. Change the details to meet your requirements and then click the **OK** button to update the database.



- To remove a sale item, click the **Remove** button. Note that when a tender sale item is removed, all associated tenders are also removed.
- While a sale item can be added only by an agent, it can be edited or removed by that agent or by the company user.

Sales

Use the Sales screen to view sales summaries by agent, client, or for the whole company.



Name	Product	Price	Date Sold	Bid	Commission
Elaine Lee	DEST0001 - Africa Diving	1299.00	22/01/2000		25.98
Andrew Fitzpatrick	DEST0004 - Bungy Jumping	3500.00	2/08/2000		70.00
Christine Montgomery	DEST0007 - Mediterranean Cruise	12999.00	22/11/1999		649.95
Christopher Burke	DEST0010 - Wine Tours	9999.00	27/08/2000		499.95
Sarah Bailey	DEST0013 - Ancient Ruins	7599.00	27/04/2000		151.98
Elaine Lee	DEST0016 - New York By Night	8995.00	20/02/2000		449.75
Christine Ronaldo	DEST0019 - Hong Kong By Night	4599.00	29/06/2000		91.98
Barbara Montenegro	DEST0022 - Hiking	3995.00	10/12/1999		79.90
Sean Hill	DEST0025 - Golf By The Sea	2995.00	22/04/2000		239.60
Andrew Fitzpatrick	DEST0028 - Island Hideaways	7995.00	1/11/1999		323.80
Levi Muir	ANQU3003 - Male Band Ring	1990.00	15/02/2000		238.80
Peter Morrissey	ANQU3006 - Painting	1150.00	22/10/1999		138.00
Elaine Lee	ANQU3009 - Choker Necklace	6500.00	15/09/2000		325.00
Heather Bisset	DEST0031 - Deer Hunting	7995.00	20/08/2000		399.75
Brian Olsen	PROP2003 - Jungle Hideaway	6250000.00	1/12/1999		218750.00
Sarah Bailey	PROP2004 - Famed House	3200000.00	17/05/2000		160000.00
Sarah Bailey	DEST0034 - Beach Getaway	8999.00	3/08/2000		179.98
Barry Ogen	ANQU4002 - Rimu table	699.00	30/12/1999		55.92
Pauline Wild	ANQU4005 - Silver Spurs	450.00	1/09/2000		54.00

To list the sales, select the agent, client, or company option using the radio buttons. If the agent or client option is selected, one of those entities must be selected from the drop-down list box to display the sales summary specific to that agent or client. To sort the results, click on any of the table column headers to sort by that column.

Notes When this summary is viewed by an agent user, only his or her sales appear in the list.

Total amounts are displayed at the bottom of the screen.

Jade Thin Client Shop Application

The Jade thin client shop application is the shop-front interface for clients to run over local or wide-area networks, or the Internet. The **ErewhonShop** application can also be run as a standard Jade client application (that is, a two-tier or *fat* client application). For details, see the following subsections.

- Logon
- Product Search

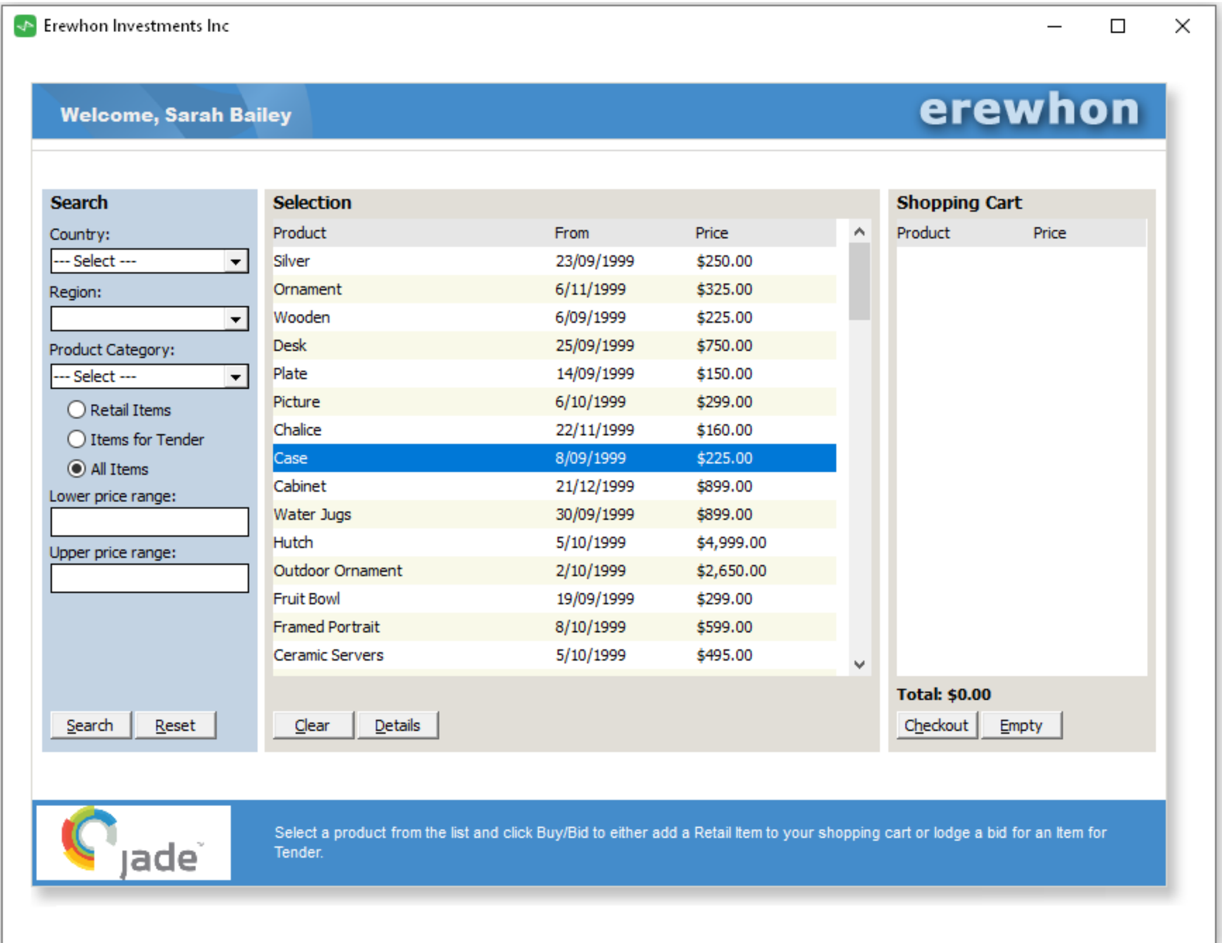
- Viewing the Details of a Product
- Buying or Bidding for a Product
- Shopping Cart
- Product Details
- Checkout

Logon

On the logon form, select a user name from the drop-down list box and then click the **OK** button. To cancel the logon, click the **Cancel** button.

Product Search

The Product Search form is the main form of the **ErewhonShop** application. To begin, search for a list of products by selecting the required search criteria in the **Search** panel, and then click the **Search** button.



A new search list can be generated at any time by changing the search criteria and then clicking the **Search** button again. Search criteria can also be reset by clicking the **Reset** button. The list of search results can be cleared at any time by clicking the **Clear** button below the list of results (this will not affect any of the items currently in the shopping cart).

Viewing the Details of a Product

To see more-detailed information about a product, select it in the search list and then click the **Details** button. This button will then be replaced by a button with a caption of **List** and the search results are replaced with details of the selected item. Clicking the **List** button while the item details are displayed returns to the list of search results.

Note If no product item is selected while viewing the list of search results, the **Details** button is disabled.

Buying or Bidding for a Product

Some products are retail sale items, while others are tender items for which you must enter a bid.

To buy a retail sale item, select it in the search results list and then click the **Buy/Bid** button, or click the **Buy/Bid** button while the item details for that item are displayed.

If the product is a tender item (that is, it requires you to make a bid), you must first click the **Details** button to see the item details for the item, and then enter a tender amount greater than or equal to the minimum price of the item.

After entering your offer, click the **Buy/Bid** button again and the tender will then be added to your shopping cart. Alternatively, if you do not want to bid on the item, click the **List** button to return to the search results list.

Shopping Cart

The shopping cart list will be updated whenever a product item is bought or tendered for. A running total is also displayed beneath the cart list. To empty the shopping cart, click the **Empty** button. To go to the checkout, click the **Checkout** button.

Product Details

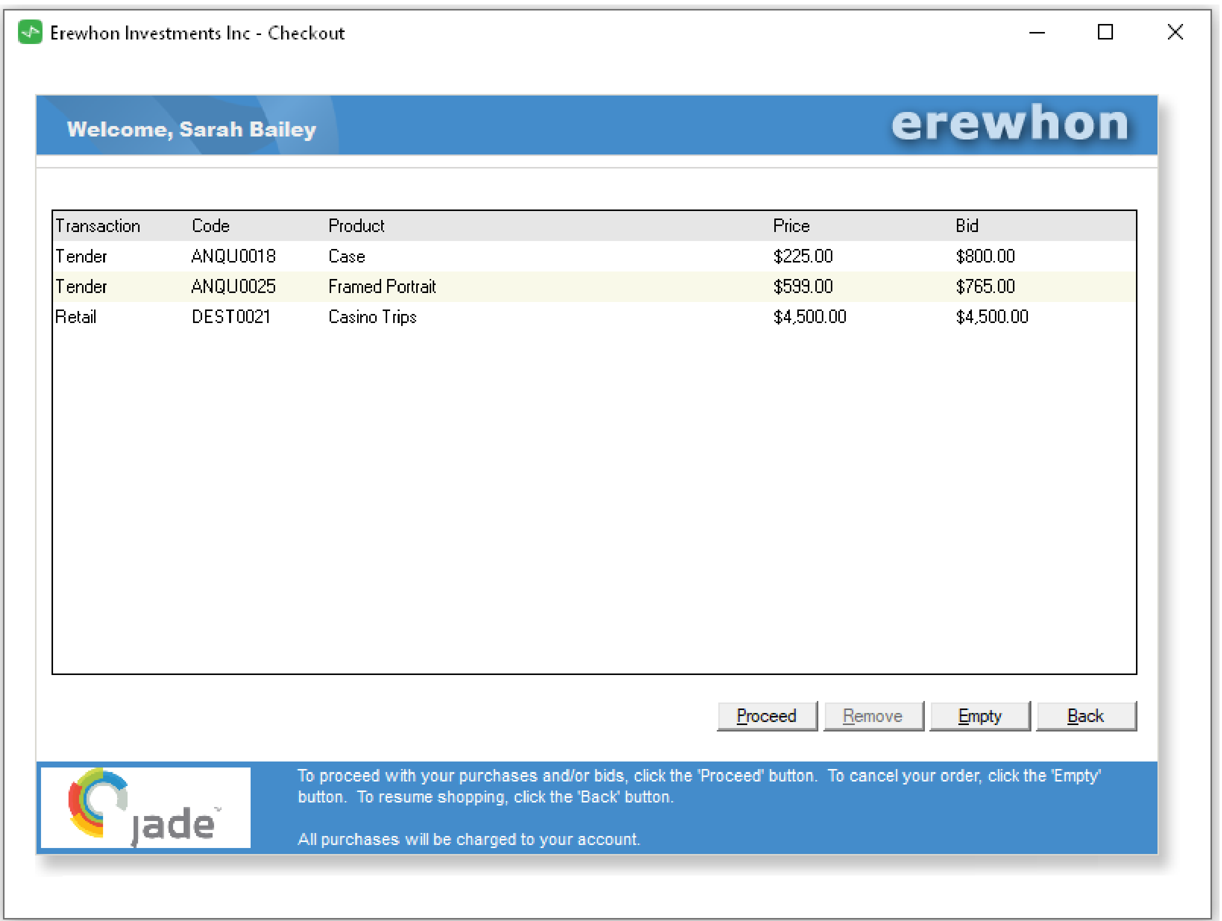
When a product is selected in the search results list, the **Details** button is enabled. Click this button to display more-detailed information about the product item.

If the product item is a tender item, the details of the product include a field in which to enter your offer (your tender amount). The same product searching and shopping cart functions are available as when the search results list is displayed (see above).

If the **Clear** button is clicked with the product details in view, the details will be replaced with an empty search results list. Clicking the **Reset** button (in the **Search** panel) with a product's details displayed will have the same effect as the **Clear** button, but the search criteria will also default to their original settings. If the **Search** button is clicked with a product's details displayed, the details will be replaced with the (new) search results list.

Checkout

The checkout is the final confirmation of your shopping cart before proceeding with the transaction of purchasing sale items or submitting any bids for tender items, or both purchasing sale items and submitting any bids for tender items.



To remove any unwanted items from the shopping cart, select the item in the list and then click the **Remove** button. To remove all of the items in the shopping cart, click the **Empty** button. If you want to return to the Product Search form, click the **Back** button. To initiate the final processing of the shopping cart, click the **Proceed** button. A list of the bought and tendered items will then be displayed so that you can confirm the transaction.

Web Shop Application

The **WebShop** application is the shop-front interface for clients to run over the World-Wide-Web. The application is built in Jade and is automatically deployed over the Internet using Jade's native functionality. For details, see the following subsections.

- Logon
- Product Search
- Viewing the Details of a Product
- Buying or Bidding for a Product

- Product Details - Tender
- Checkout

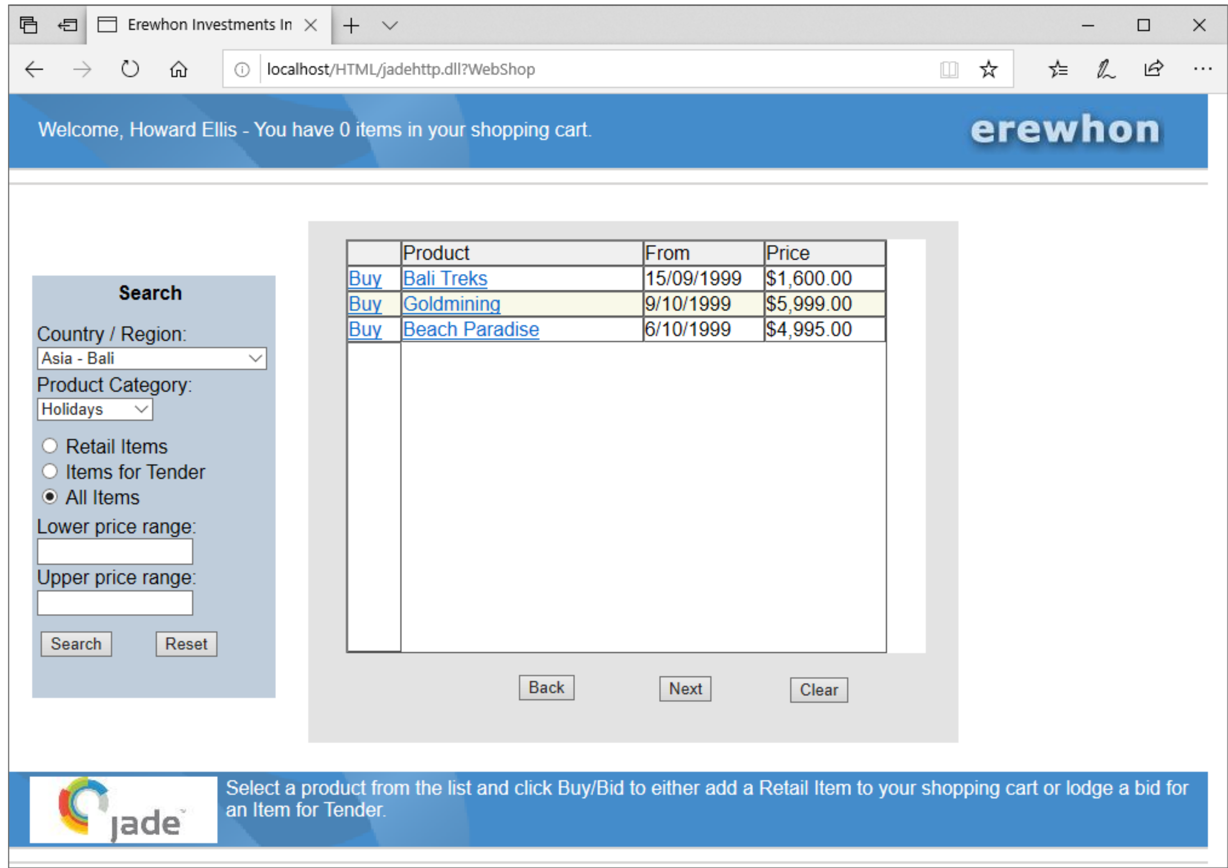
Logon

The following image is the **WebShop** application logon form.

Select a user name from the drop-down list box and then click **Enter**.

Product Search

The Product Search form is the main form of the **WebShop** application. To begin, search for a list of products by selecting the required search criteria in the **Search** panel and then click the **Search** button.



A new search list can be generated at any time by changing the search criteria and then clicking the **Search** button. The search criteria can also be reset by clicking the **Reset** button. The list of search results can be cleared at any time by clicking the **Clear** button below the list of results (this will not affect any of the items currently in the shopping cart). To scroll through the search results list, click the **Next** button or the **Back** button.

Some products are retail sale items, while others are tender items for which you must enter a bid.

Viewing the Details of a Product

The second column in the search results list is the name of the product item, which is a link to the details of the product. By clicking on the product item's name link, the details of the product will be displayed. If the selected product is a tender item, then as the details of the product are displayed, a field will also be shown in which to enter your offer (your tender amount).

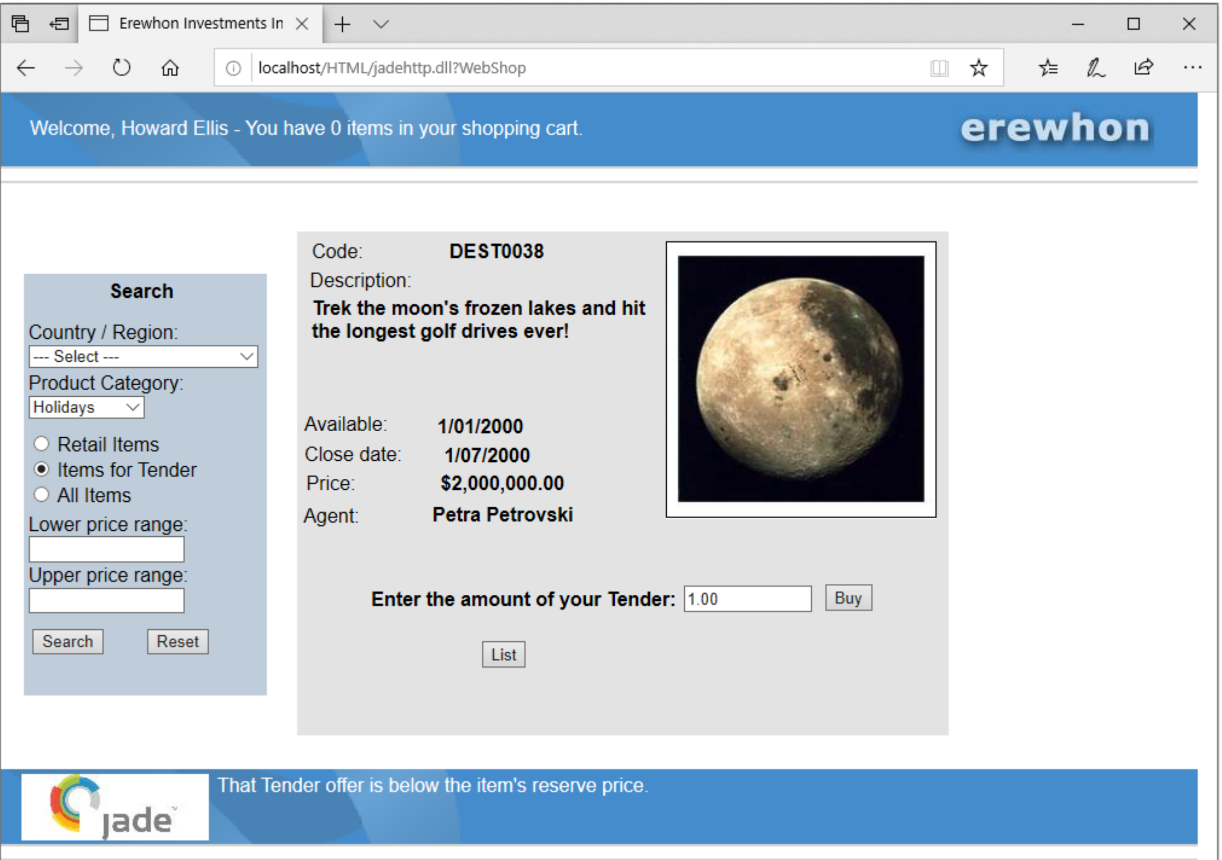
Buying or Bidding for a Product

The first column in the search results list will contain a link named **Bid** or **Buy**. Click this link to purchase a sale item (buy) or bid for a tender item. If you are buying a product item, it will be added to your shopping cart immediately. If the selected product is a tender item, the details of the product will be displayed and a field will also be shown, in which to enter your offer (your tender amount).

After entering your offer, click the **Buy/Bid** button and the tender will then be added to your shopping cart. To return to the search results list, click the **List** button. To go to the checkout, click the **Checkout** button.

Product Details/Tender

When a product is selected in the search results list by clicking the product name, more-detailed information about the product item will be shown. Click the **List** button to revert to the search results list.

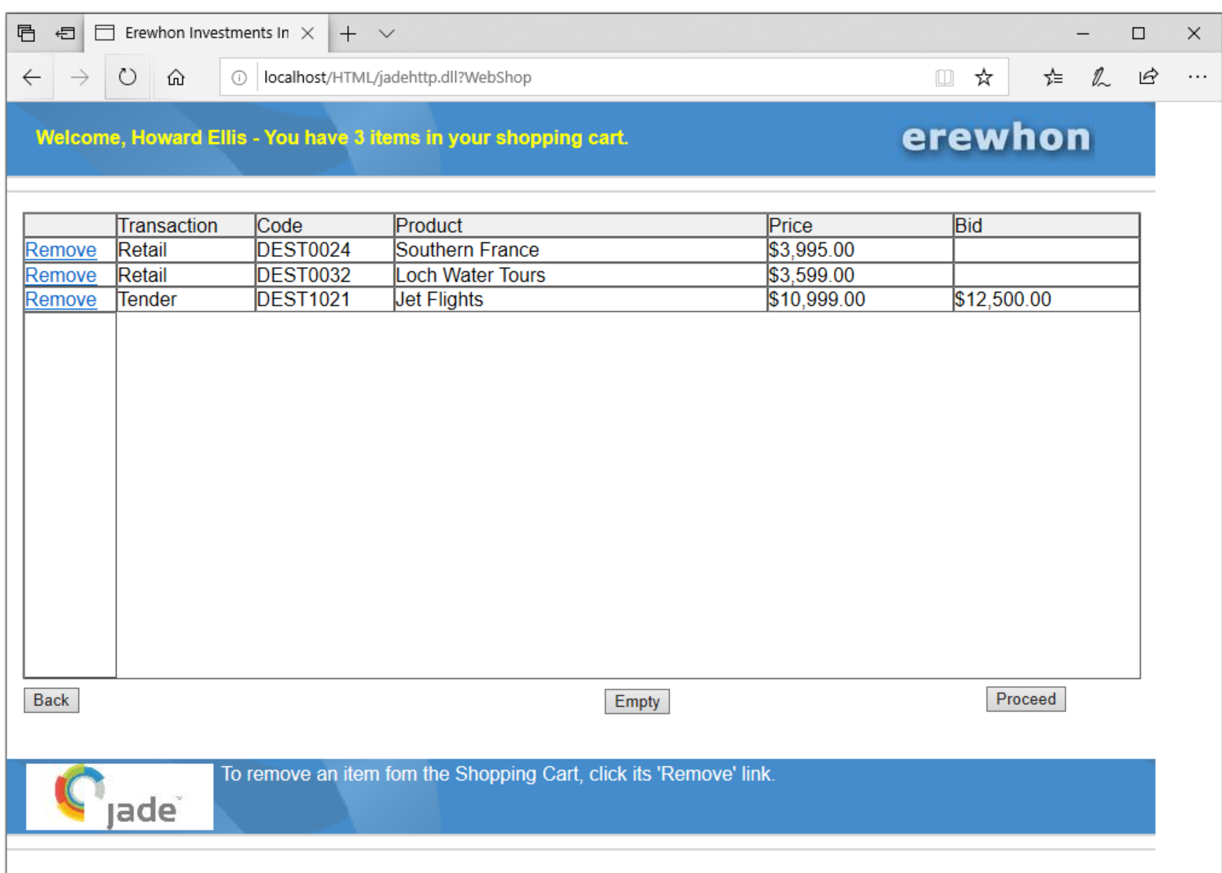


If the product item is a tender item, the details of the product will include a field in which to enter your offer (your tender amount).

Clicking the **Reset** button with the product details in view will cause the product details to be replaced with an empty search results list and the search criteria will also default to their original settings. If the **Search** button is clicked with a product's details displayed, the details will be replaced with the new search results list.

Checkout

The checkout is the final confirmation of your shopping cart before proceeding with the transaction of purchasing sale items or submitting any bids for tender items, or both purchasing sale items and submitting any bids for tender items.

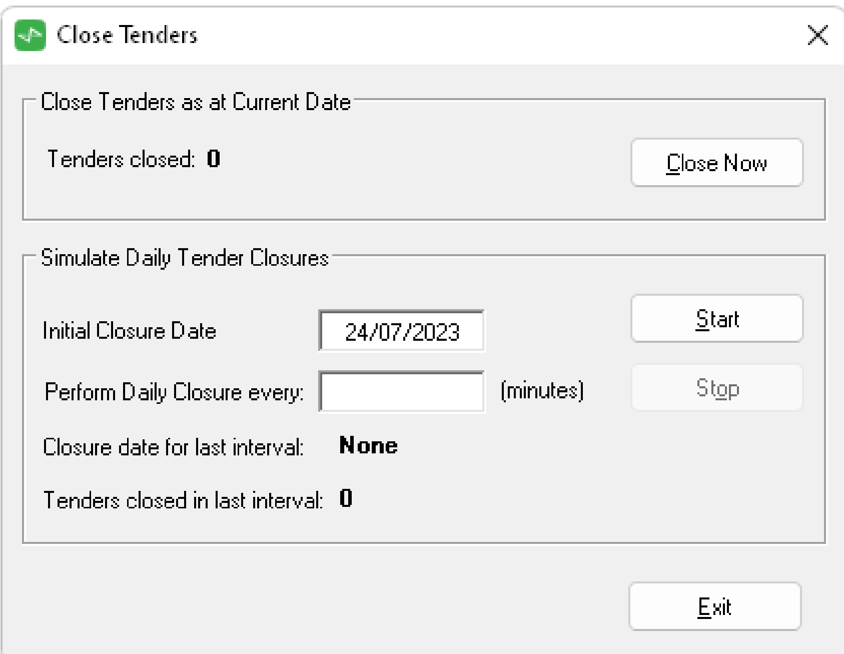


To remove any unwanted items from the shopping cart at this point, click the **Remove** link (underlined) in the first column of the item's row. To remove all of the items in the shopping cart, click the **Empty** button. If you want to return to the Product Search form, click the **Back** button. To initiate the final processing of the shopping cart, click the **Proceed** button. A list of the bought and tendered items will then be displayed, so that you can review the transaction.

Tender Closure Application

The **Tender Closure** application runs the processing that converts the highest tenders for sale items into actual sales. It does this by processing all tender items at a specific date, and if the tender item's closure date is on or prior to the specified date and the item is not sold, the highest tender offer is converted into a sale. Typically, such an operation would be implemented as a batch (separate) process that runs outside the main applications (for example, it can be implemented as a separate application that schedules the processing to be run once a day, late at night), which is why we have implemented this processing in a separate application.

The application has only one form, shown in the following image.



Enter the date in the **Close tenders as at date** text box at which tenders are to be closed. Any unsold tender items with a closure date on or prior to this date will be processed.

To process tenders immediately, click the **Close Now** button. The operation will start and when it completes, the number of closed tender items will be displayed. The application also gives an example of how to use Jade timers to schedule processing. Enter a number of minutes in the **Closure interval** text box and then click the **Start** button. This will start a timer that counts down from the specified number of minutes, with progress being displayed at the bottom of the form.

When the time period expires, any unsold tender items with a closure date on or prior to the date specified in the **Close tenders as at date** text box will be processed. The number of closed tender items will be displayed. The closure date will then advance one day and the timer will restart from the specified number of minutes. This allows you to simulate the scheduling of the operation to run once a day. To stop the timer, click the **Stop** button (which is enabled only when the timer is active). To shut down the application, click the **Exit** button.

Part 3 Model Implementation

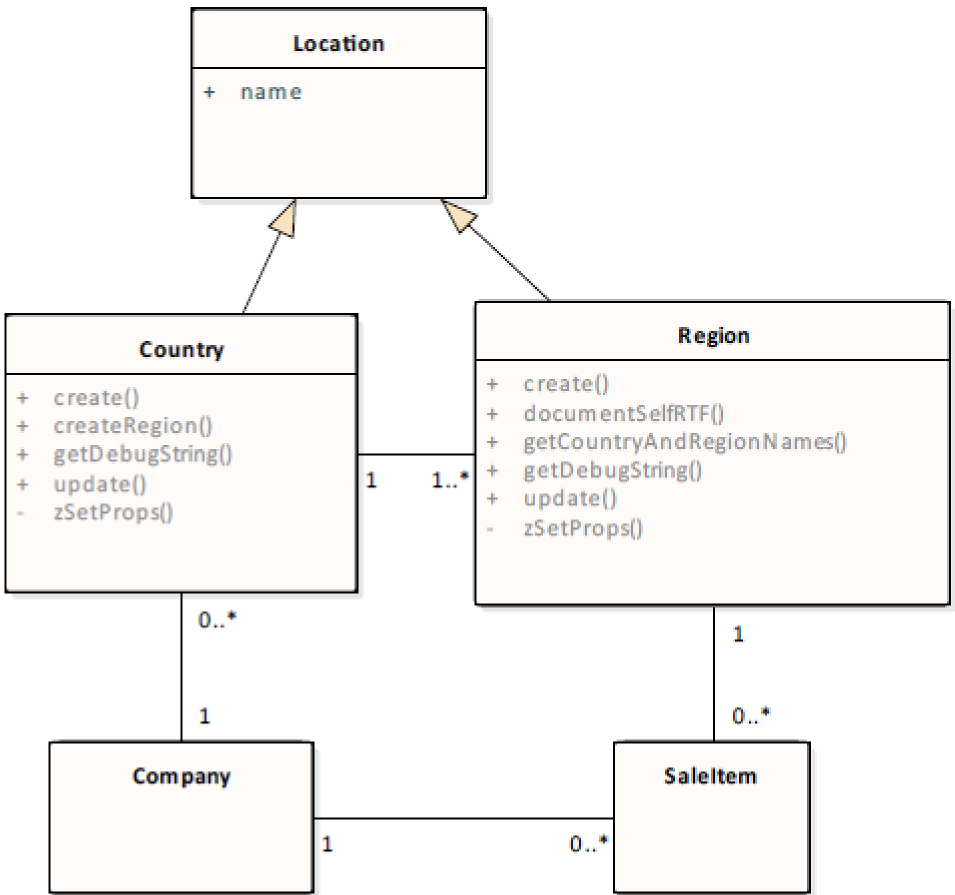
This section describes the model entity classes that implement the core object model of the Erewhon Investments system. All of these classes are defined in **ErewhonInvestmentsModelSchema** and they inherit from a common superclass, **ModelEntity**, as follows.

- ModelEntity (abstract)
  - Address
  - Agent
  - Client
  - Company
  - CommissionRate
  - Location (abstract)
    - Country
    - Region
  - Sale (abstract)
    - RetailSale
    - TenderSale
  - SaleItem (abstract)
    - RetailSaleItem
    - TenderSaleItem
  - SaleItemCategory
  - Tender

Diagrams describe the relationships between these classes, followed by an overview of each concrete class. For more details, see the following subsections.

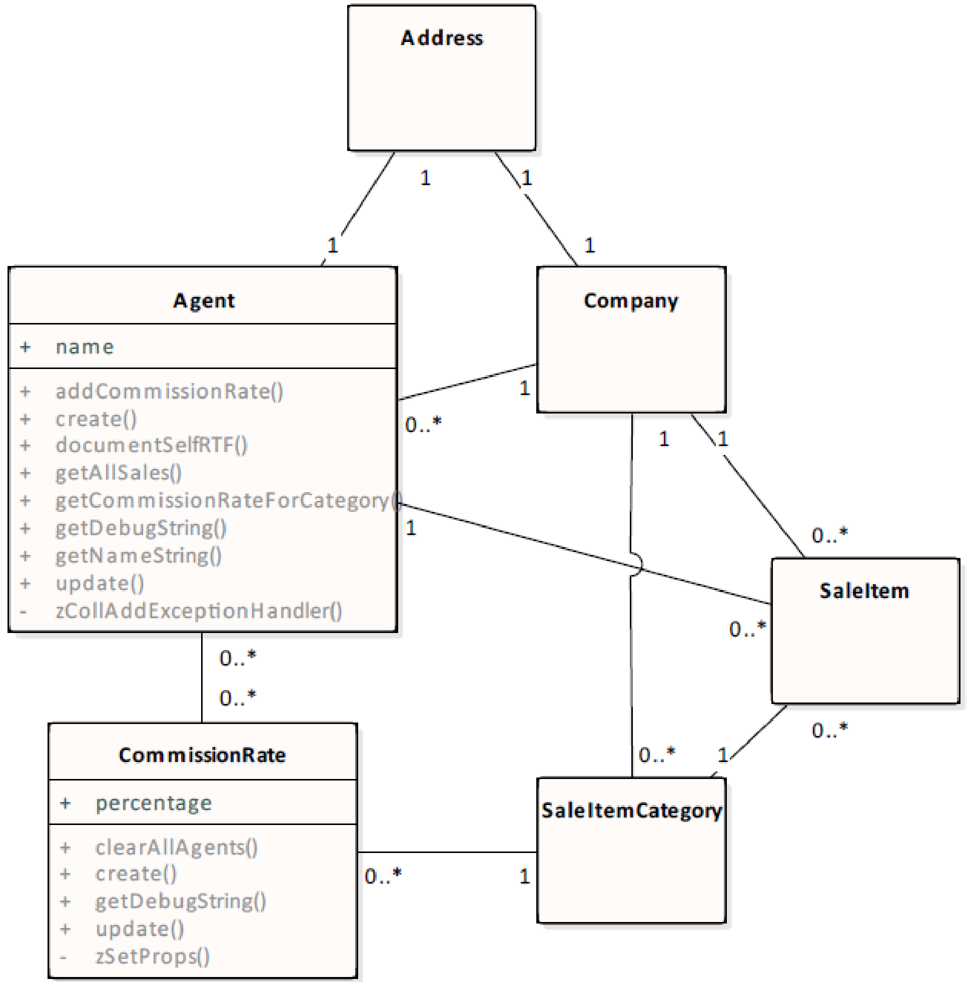
Locations

The following diagram describes the relationships between **Location**, **Country**, and **Region** and their related **ModelEntity** classes.



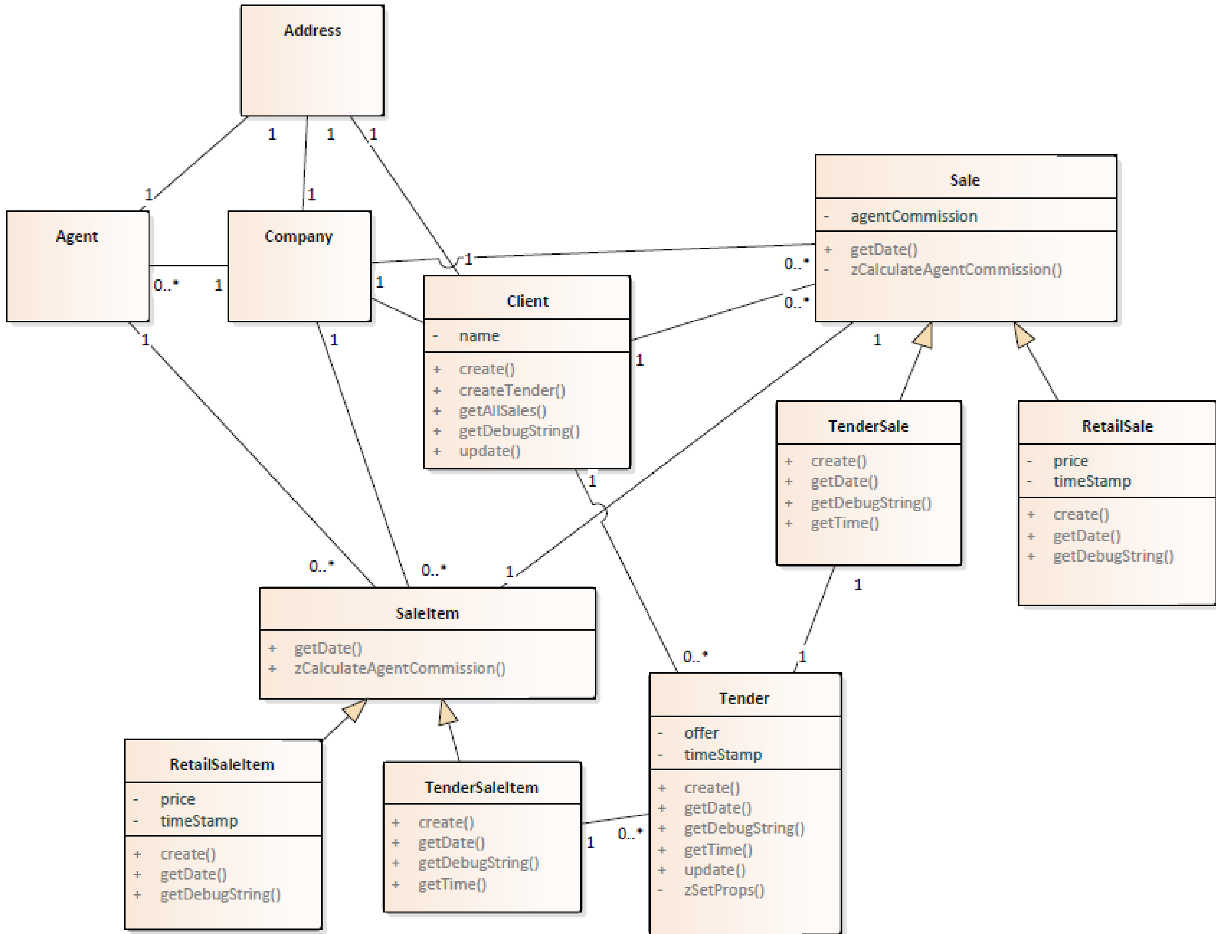
Agents and Commission Rates

The following diagram describes the relationships between **Agent** and **CommissionRate**, and their related **ModelEntity** classes.



Sales and Clients

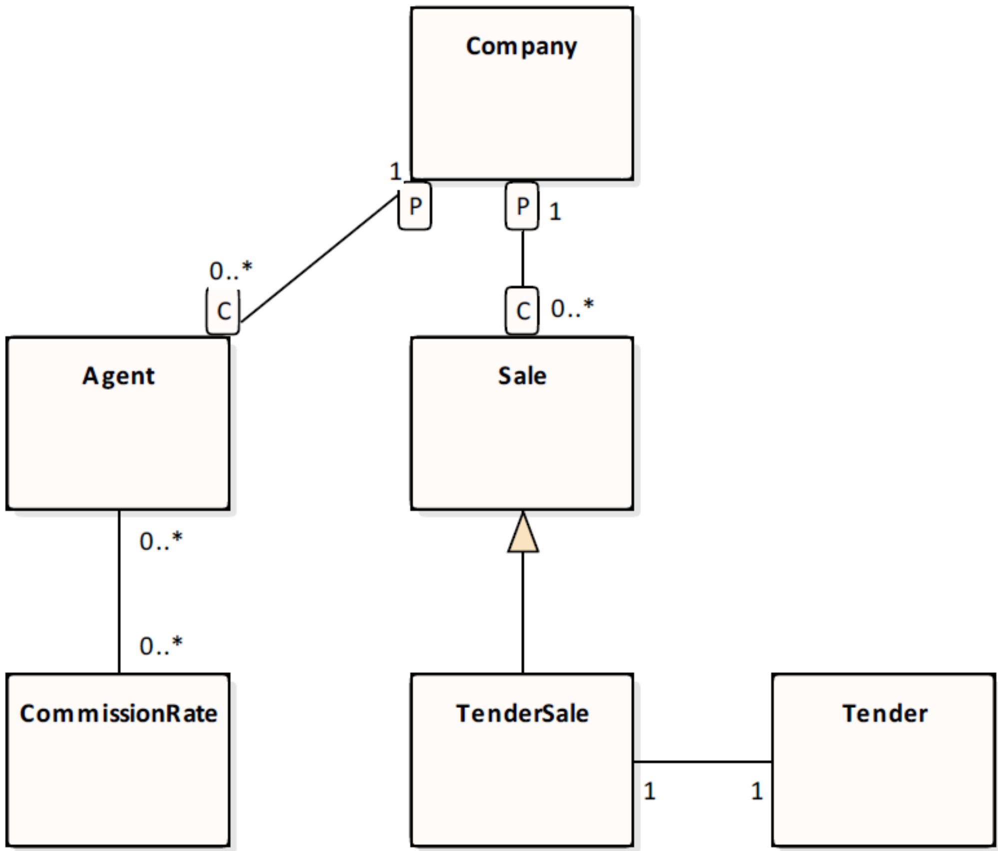
The following diagram describes the relationships between **Sale** (and subclasses), **SaleItem** (and subclasses), **Client** and **Tender**, and their related **ModelEntity** classes.



Jade Reference Diagram

The following diagram describes the Jade implementation of the relationships described above. In Jade, a two-way relationship is implemented by defining an *inverse* between two reference properties. Single-value references are used to implement the "one" side of a relationship; collection references are used to implement the "many" side. In this way, one-to-one, one-to-many, and many-to-many relationships can be implemented.

This diagram has an example of at least one relationship of each cardinality. Cardinalities are represented by the standard UML syntax, except that **P** and **C** represent a parent/child relationship, where **P** is the parent.



Agent

Agents represent users of the system who bring in items for sale. These items can be offered for retail sale or for sale by tender. An agent can have many items for sale at any one time. For each sale item category in the system, an agent operates at one (and only one) commission rate. The commission rate from which an agent is operating for a category determines the percentage of each sale from the category that an agent takes as commission.

Client

Clients represent users of the system who log on to search for and purchase items. Clients can purchase retail items immediately, or place bids on items offered for sale by tender. Once a tender sale item is closed, the highest tender is converted into a sale for the client. Clients know about all sales in which they have been involved, and all tender offers they have made.

Company

A single instance of **Company** provides the root object for the system. We assume only one persistent company instance at any one time and the **create** (constructor) method of **Company** enforces this. **Company** represents the top of the parent-child reference hierarchy, and provides collections through which we can navigate to all other objects in the system.

Commission Rate

One or more commission rates can exist for each sale item category. Each commission rate can have multiple agents operating from it. A commission rate determines the percentage commission its agents make on sales from the commission rate's category.

Country

Countries provide a means of grouping and organizing geographical regions. Each country can have multiple regions defined for it and as such, may or may not represent an actual country; for example, a continent that has relatively few regions may be counted as a country.

Region

Regions provide a means of grouping sale items into geographical areas. Each region is owned by a country and can have multiple sale items located in it.

Retail Sale

Retail sale objects model sales of retail items. Each retail sale has a price and a time stamp, and inherits its client, sale item, and company references from the **Sale** class. A retail sale knows the item that was sold and the client to whom it was sold. The agent's commission on the sale is calculated at the time the sale is created.

Tender Sale

Tender sales represent sales of items offered for sale by tender. A tender sale is created when the closure date on a tender sale item has passed, and the highest tender is accepted and converted into a sale. Each tender sale knows the item sold, the client to whom it was sold, and the winning tender object. The agent's commission on the sale is calculated at the time the sale is created.

Retail Sale Item

Instances of this class represent items offered for retail sale. All items are owned by the company and are organized into categories and geographical regions. Each item knows the agent who brought it in for sale. All sale items have a two-part code consisting of a string prefix followed by an integer number. The prefix is supplied by the application, while the number is allocated automatically when an item is created. Items can also hold a 200 by 200 pixel image of themselves.

Once an item has been sold, its **mySale** property refers to the sale in which it is involved. If **mySale** is not assigned (that is, it has a null value), the item is not yet sold.

Tender Sale Item

Instances of this class represent items offered for sale by tender. All items are owned by the company and are organized into categories and geographical regions. Each item knows the agent who brought it in for sale. All sale items have a two-part code consisting of a string prefix followed by an integer number. The prefix is supplied by the application, while the number is allocated automatically when an item is created. Items can also hold a 200 by 200 pixel image of themselves.

A tender sale item has a minimum (reserve) price, offers below which will not be accepted. The item's closure date indicates the date at which bidding will stop. At this date, the highest tender is accepted and a sale is created for the item. A tender sale item knows all bids that have been made for it. Once an item has been sold, its **mySale** property refers to the sale in which it is involved. If **mySale** is not assigned (that is, it has a null value), the item is not yet sold.

Sale Item Category

Sale item categories allow items to be grouped into logical categories. All categories are owned by the **Company** object.

Categories also hold all of the commission rates at which agents can operate for sale items belonging to the category.

Tender

Tender objects represent bids made by clients on items offered for tender sale. A tender holds the offer price and time stamp of the bid, as well as the client who made the bid and the item for which they have tendered. If a tender is accepted when bidding for an item is closed, a tender sale object is created and **myTenderSale** will be set to this object. If **myTenderSale** is non-null after bidding on the tender's item has closed, it means that the tenderer won the item.

Part 4 Design Considerations

This section discusses some of the design issues considered during implementation of the Erewhon Investments demonstration system. It focuses on those we feel are most important, and as such should not be seen as an exhaustive discussion of all design issues. What we propose in this document are guidelines.

We have tried to illustrate as many points as we can in the Erewhon system, without making it too complex. At the same time, it is intended to be a working multiuser system that deals with a number of issues encountered when building production applications. This is essential to illustrate the rationale of our design decisions.

What we're trying to stress are the design issues themselves, or *themes*. The demonstration system is just one possible implementation. There are undoubtedly many other ways of addressing the problems we shall discuss. We have tried to keep things as straightforward as possible to make it easier for those new to Jade, but without trivializing the points.

The main thing to take away from this document is an awareness of some of the issues that should be considered early on in a development project.

For details about design considerations, see the following subsections.

Conventions

Before we get underway, we should point out some of the conventions used in the Erewhon system. These should be seen as guidelines only, as several are subject to personal preference.

- All protected property and method names start with a lower case **z**. We use this to distinguish them from public features, and to force them to appear at the end of property and method lists.
- Single-value (that is, non-collection) references are prefixed with **my**.
- Multi-value (that is, collection) references are prefixed with **all**.
- Global constants are used extensively for such things as error numbers, application names, and version numbers. From the Browse menu in Jade, select **Global Constants** command to view the global constants for a schema.
- Except for development, testing, or peripheral methods, literal strings are not used in code. Instead, strings are defined as translatable strings. To view translatable strings for a schema in Jade, select the **Strings** command from the Schema menu.
- In general, we prefer to make properties read-only rather than implement *get* methods for them (see "Model Operations", later in this document).

Models, Views, and Controllers

The Model, View, Controller architecture (MVC) was popularized by Smalltalk. It divides a system into an underlying model, any number of different views of the model, and controllers that synchronize interaction between the model and the views. MVC makes it possible to concentrate on the essentials of a system (the model), and add the application and user interface layers independently. There can be many different view and controller pairs for each model; the intention is that views and controllers can be modified extensively with little or no change in the model.

For many systems, though, the role of the controller is small, with little or no distinction between views and controllers in terms of implementation. While they always represent distinct concepts, often they are implemented as one. Views can take responsibility for their own synchronization and sometimes the model provides synchronization services. The Erewhon system is an example of this. In such cases, we can simply refer to the Model and the Views (MV).

Model and View Separation

The model and views have been separated into their own schemas (which are discussed in the following section). This makes explicit the distinction between the model and its views.

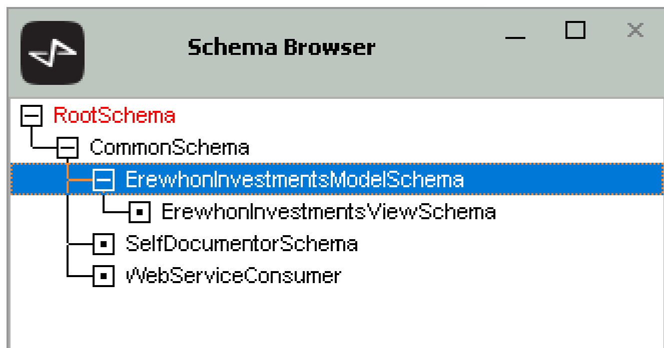
The model should focus on the problem (or business) domain. The question to start with is "How do we best model the business operations?". By separating the model from the views, the model can be made more independent of application-level and user interface requirements. Separating the model allows you to build a more-stable base, since the business domain (which the model represents) is generally less likely to change than the application layers or the user interface, or both the application layers and the user interface. A well-defined model can support several applications. For example, in the Erewhon system we have one model schema, with the view schema defining four applications that run over this model.

We have used subschemas in Jade to separate the model from the views. It allows for a cleaner, more well-defined design and implementation. It also means that separate development teams can more easily work on separate parts of the system, but still within the same single Jade environment. Separating the views from the model by packaging them in their own schemas prevents the model schema from becoming cluttered with user interface implementation, and means that the model schema can support many different views. It also makes it easier to identify the services provided by the model.

Schemas

A schema is the highest-level organizational structure in Jade and represents the object model for a specific domain. A schema is a logical grouping of classes, together with their associated methods and properties. These effectively define the object model upon which applications are based. The appearance and functionality of applications in a schema can differ, but they all share the underlying object model defined by the schema. Jade provides the **RootSchema**, which is always at the top of the schema hierarchy. The **RootSchema** provides essential system classes that are available to all subschemas.

The schemas that make up the Erewhon Investments system are shown in the following image of the Jade Schema Browser window.



The schema hierarchy is analogous to a class hierarchy and similar terminology is used. For example, **ErewhonInvestmentsModelSchema** is a subschema of **CommonSchema**, and **ErewhonInvestmentsViewSchema** is a subschema of **ErewhonInvestmentsModelSchema**. Subschemas inherit all the classes, methods, and properties that are defined in their superschemas. Therefore, all schemas in the Erewhon system inherit the entities defined in the **CommonSchema**.

The system is implemented in three main schemas and two supplementary schemas, as follows.

CommonSchema

This inherits from **RootSchema** and provides common services for all of its subschemas. Services include common exception-handling methods on the **GCommonSchema** class and a selection of subclassed controls, including a date text box and collection viewer list box.

ErewhonInvestmentsModelSchema

This schema implements the model for the system. All classes for which persistent objects are created are defined in this schema as subclasses of **ModelEntity**. The schema also implements a number of classes that provide services to the views including transaction agents and order proxies. A number of utility **JadeScript** methods are provided in this schema for development and testing use, such as methods to initialize the database.

ErewhonInvestmentsViewSchema

This schema implements the views or applications that run over the model. The entire user interface is implemented in this schema. The schema defines six applications.

- **Administration** is a back-office application that company staff and agents can use to administer the system. It is expected that this application would be deployed on a mixture of standard clients and Jade (smart) thin clients.
- **ErewhonShop** is a front-office application that clients will use to search the items for sale, and to buy or bid on items. It is expected that this application would be deployed on Jade thin clients, but the application can obviously be run on standard clients as well.
- **WebShop** is a web application server that allows clients to search, buy, and bid on items from within a web browser. It provides similar functionality to the **ErewhonShop** application but with a slightly different interface for web browsers.
- The **TenderClosureApp** closes all sale items open for tender if their closure date is at or prior to a specified date. The application closes each tender sale item and accepts the highest bid.

  You can specify a timer interval so that the operation is performed automatically on a regular basis. We expect that one copy of this application would run on a standard client (possibly on the same machine as the server) and would be set to run the operation at a time when activity is low.
- **WebServiceOverHttpApp** and **WebServiceOverTcpApp** are applications that demonstrate Jade's web service provider capabilities. For details, see the *SOAP Web Services* white paper (which is also available from the Jade website at https://www.jadeworld.com/jade-platform/developer-centre/documentation/white-papers).

The view schema also extends model classes by adding methods to them in the view schema. For example, **display** and **getSearchResultString** methods are added to **ModelEntity** subclasses in the view schema.

SelfDocumentorSchema

This schema demonstrates Jade interfaces and Jade packages by exporting a framework that allows objects to document themselves. Refer to the **FormDocumentorSetup** form in the **ErewhonInvestmentsViewSchema**, to see how this package and the interface that it exports is used.

The **btnShow_click** method on the form invokes the package. This form can be accessed by running the **Administration** application from the view schema and selecting the **Misc | Show Details via Interface** menu item.

WebServiceConsumer

This schema demonstrates Jade's web service consumer capabilities. For details, see the *SOAP Web Services* white paper (which is also available from the Jade website at https://www.jadeworld.com/jade-platform/developer-centre/documentation/white-papers).

Transaction Separation

The **TransactionAgent** class in the model schema provides activity methods that implement all transactions in the model. Except for development or peripheral tasks (for example, initializing the database), there are no other places in the model that begin or commit transactions. Our views also should rarely (if ever) begin and commit their own transactions. Rather, transaction methods should be added to the **TransactionAgent** in the model, or to the **TransactionAgent** subschema copy class in the view schemas.

Transactions are a concept in their own right. A transaction brackets one or more model operations into one activity (logical unit of work) bounded by begin transaction, and commit or abort transaction. Each operation in the model should generally not be responsible for going into and out of transaction state, as this is error-prone and reduces the flexibility by which operations can be combined into transactions or activities.

It is important to recognize that transactions are application-defined. The application requirements determine the boundaries of a transaction, as it is the application that determines what is a logical unit of work. In the Erewhon system, we have defined the **TransactionAgent** class in the model schema because we have only one set of transactions for the views. We thought it made the system an easier example to understand if the transaction methods were defined in the model schema along with the operations that they call. However, it is important to note that although they are located in the same schema, the transaction methods are distinct from the model. They are determined by the requirements of the applications in the view schema.

Indeed, for a larger system, it may be appropriate to implement transactions in their own subschema between the model and the views. Alternatively, each view can implement its own **TransactionAgent** class (or an equivalent mechanism). For systems with many transactions or complex transactions, creating several specialized **TransactionAgent** (or equivalent) subclasses is an approach worth considering. Some systems even go as far as defining classes to represent individual transactions. The technique that you adopt depends on the specific requirements of your application.

Separating out transactions in this way brings several benefits, as follows.

- Transaction code is centralized.
- Model operations do not have to worry about beginning, committing, or aborting transactions.
- Model operations can be more-easily combined into different transactions or activities.
- It provides a centralized layer for enforcing certain lock policies.
- It provides a layer to encapsulate exception handling.

Model Operations

Operations in the model should be based on business application requirements. In **ErewhonInvestmentsModelSchema**, operations are implemented as methods on the **ModelEntity** subclasses.

We have avoided *unconditionally* defining *get* and *set* methods for all **ModelEntity** subclass properties. It is not uncommon for projects (almost religiously) to insist on defining all properties as protected and implementing public *get* and *set* methods. However, this tends to overlook certain higher-level Jade concepts that provide an effective alternative without sacrificing encapsulation or flexibility, and with lower runtime overhead. In general, we have defined properties that are part of the public interface of a class as read-only, for the following reasons.

- A property, when accessed in the Jade language, is already equivalent both conceptually and in reality to a pair of related *get* and *set* operations implemented by the Jade Object Manager. When you refer to a property using the Jade language, you are doing so via the default *get* and *set* operations provided by the Object Manager; never directly.
- The access option for a property defines which of the implicit *get* and *set* operations are part of the public interface of the class; protected implies none, read-only implies get only, and public implies both.
- If you insist on defining and writing *get* method wrappers for *all* properties that simply return the property value, this does not really increase encapsulation. All it does is incur the unnecessary runtime overhead of dynamic binding and method dispatch.

It is accepted, though, that for implementation encapsulation (that is, information hiding) *get* methods are appropriate. However, these should be identified on a case-by-case basis. For example, the **Sale** class in **ErewhonInvestmentsModelSchema** implements a **getAgentCommission** method that simply returns the value of the **zAgentCommission** property. The implementation of sale commissions has in fact changed a couple of times from being a value derived when the *get* is requested, to being a value calculated and stored on the sale when it is created. These changes indicate that in this case, having a *get* wrapper for the agent commission is warranted but implementing a **getName** method on **Address**, for example, instead of simply defining **name** as read-only would seem heavy-handed.

- If you ever need to redefine the behavior of the implicit *get* or *set* operations (without changing the type of the property), Jade has the solution: mapping methods. These can be added at any time, so there is no loss of flexibility.
- The read-only option still imposes the discipline whereby only the methods defined in a class (the implementation) can change the state of its instances (the desirable level of encapsulation).

Unconditionally defining *set* operations for *all* properties defeats encapsulation, as it exposes every property to updates from any other class. This can also give rise to update order dependencies, as the order in which properties are set cannot be controlled if they can be set from anywhere. It is common to set references individually and to set groups of attributes (that is, properties other than references) in a single call. This is what we have implemented in the Erewhon system (see the update methods on the **ModelEntity** subclasses).

Bear in mind that our decision to implement a single update method for attributes on some classes was based on our business/application requirements. Those classes implementing a single update method for attributes represent low-volatility data that will be updated one object at a time from the views. However, imagine, for example, that we have to support a frequent (albeit fictitious) transaction that requires updating just the e-mail address on *all* **Address** objects. It would make sense in this case to have an individual **setEmail** method rather than passing all attribute values to update, knowing that only the e-mail address is going to change. The key point is to determine what *set* or *update* methods you need, and how many properties each of them update, based on the transactions and operations your model must support.

Be wary of defining *set* or update methods for each class that set *all* of its properties (attributes *and* references, as opposed to just attributes) in one call. An exception to this is when an object is first created. For object creation, it is often useful to have a method that sets all properties in one call. The model schema does this by implementing a **create** method on **ModelEntity** subclasses. This method is to be invoked only when an entity is first created. For updates to existing objects, setting all attributes (that is, properties other than references) in one method is common, but setting references should be considered more carefully. The Erewhon model schema implements an update method on several **ModelEntity** subclasses that sets all *attributes* only.

The decision as to what references can be updated (after an object has been created) should be based on business/application requirements. Methods should be defined that represent the operations to be performed, rather than just implementing methods that set all references at once. Having such generic methods makes model operations less clear, reduces encapsulation, and can introduce update order dependencies (increasing the chance of deadlocks). Typically, changing a single reference is a single operation. Of course, that does not exclude the possibility of operations that need to change multiple references. However, methods that do this should be the exception, not the rule.

ModelEntity classes implement specific operations to change those references to which updates are permitted; for example:

- **Agent::addCommissionRate**
- **CommissionRate::clearAllAgents**
- **SaleItem::updateCategory**
- **SaleItem::updateRegion**

Exception Handling

Exception handlers are an effective means of encapsulating code for handling unexpected or infrequent errors. Within a transaction, exception handlers are often responsible for restoring things to a consistent state if something goes wrong (usually by aborting the transaction). They are useful for efficiently guarding against invalid object references on an exception basis, rather than always checking the validity of an object in-line (which can require more round trips to the server and defeat caching).

For a discussion of exception handling in Jade, see the *Jade Exception Handling* white paper (which is also available from the Jade website at https://www.jadeworld.com/jade-platform/developer-centre/documentation/white-papers).

Exception handling is used extensively in the Erewhon system. Some examples are:

- **TransactionAgent** methods arm handlers to catch exceptions and translate them into error numbers that are returned to the views. This shields the views from having to implement their own exception handling around transaction requests. For examples of this, look at any **TransactionAgent** method, the **TransactionAgent zExceptionHandler**, **zLockExceptionHandler**, **zSilentLockExceptionHandler** methods, and the **ActivityAgent zRegisterObjectAndErrorCode** method.
- The model schema **Application** subclass **ErewhonInvestementsModelApp::initialize** method arms a generic global exception handler and a generic global lock exception handler that are used to catch any exceptions not caught locally. The **GCommonSchema** class in **CommonSchema** provides both of these exception handler methods. The **commonExceptionHandler** method gives an example of a simple generic exception handler and the **commonLockExceptionHandler** gives an example of a lock exception handler. The view schema **GErewhonInvestmentsViewSchema** class reimplements the **commonExceptionHandler** method to perform some exception handling specifically for the **WebShop** application.
- The **ModelEntity** class implements **zCollAddExceptionHandler** to safely add an object to a collection when it is already there and **zCollRemoveExceptionHandler** to safely remove an object from a collection when it is not there. To see uses of these methods, select them in the Class Browser window in Jade and then select **References** from the **Methods** menu.
- The **FormClientApp** class in the view schema implements a **zInvalidObjectExHandler** method that catches all invalid object or deleted object exceptions and redisplays the current form. This exception handler is armed at the start of **FormClientApp** subclass event methods.

Cache Synchronization

When an application references a persistent object, Jade first looks to see if the object is resident in local cache. If it is, the cached object is used for the current operation. If the object is not in cache, it is fetched from the server, brought into cache, and used for the current operation. Once an object has been brought into cache, it is available for use in subsequent operations. Objects do not exist in cache indefinitely. Jade can discard objects from cache when required, to make space for objects being brought into cache. Jade also provides facilities for you to manually discard objects. When an object is discarded, the next reference to it will cause it to be fetched again from the server. In a multiuser system, a locally cached object can be made obsolete when another user updates it. A caching strategy is necessary to keep locally cached objects synchronized with the database, when required.

The Erewhon Investments applications are multiuser and therefore require a caching strategy. A good caching strategy ensures that the objects stored in local cache are the latest editions where necessary, and that this is maintained with the minimum amount of network and processing activity.

An application that makes efficient use of cache will have significant performance advantages over one that does not, as Jade's use of cache is one of its key strengths. A caching strategy comprises all of the mechanisms you use to synchronize local cache with the Jade database.

In the Erewhon Investments system, we have the following considerations.

- While Jade has facilities for developers to manually request that objects be resynchronized in local cache, the automatic cache coherency provided by Jade makes life much easier for developers, and Erewhon takes advantage of this feature. Readers familiar with earlier versions of Erewhon will notice how much code the automatic cache coherency eliminates!

  Automatic cache coherency is enabled by adding the following lines to the Jade initialization file (an example **jade.ini** file for the Erewhon system is provided in **examples/erewhon/erewhonjade.ini**).

  ```
  [JadeServer]
  AutomaticCacheCoherencyDefault=true
  AutomaticCacheCoherency=ServerDefault

  [JadeClient]
  AutomaticCacheCoherency=ServerDefault
  ```

  With automatic cache coherency enabled, objects updated in other nodes (database server, application servers, background nodes, or standard clients) are automatically reloaded in local cache.

- Each operation in the model (that is, methods defined on **ModelEntity** classes) must handle its own integrity locking. By integrity locking, we mean that each operation is responsible for locking those objects of which it requires the latest editions in order to ensure data integrity. Each method assumes responsibility for its own integrity so that the operation is safe, regardless of the context in which it is invoked. Any synchronization locking (that is, locking specifically to serialize transactions) will be done in the respective transaction methods; for example:

  ```
  TransactionAgent::trxCloseTendersAtDate
  ```

- Any **TransactionAgent** method that allows an object to be updated must provide a mechanism for the caller to request that it performs an edition check. This allows the **TransactionAgent** method to verify, on behalf of the caller, that the expected edition of the object is being updated.

- Outside of **TransactionAgent** methods, we are concerned primarily with ensuring that application forms are kept synchronized when objects that they are viewing change and that we have the latest edition of an object before comparing it against search criteria.

We have made use of several mechanisms to implement our caching strategy, in order to illustrate some of the approaches available to you in Jade:

- Automatic cache coherency
- **listCollection**
- **CollectionListBox** subclassed control
- Object notifications
- Edition checking

These are discussed in the following sections.

listCollection

The **listCollection** method of the **ListBox** and **ComboBox** classes (provided by the **RootSchema**) enables list box or combo box controls to have a collection attached to them. Logic attaches the collection to the list box or combo box by using the **listCollection** method. If you use this method to attach a collection to a list box or combo box, little is required to load entries into the list.

If the list box is not sorted, an entry is retrieved from the collection only when it is to be displayed or accessed by logic. Only a few entries from the collection are therefore initially accessed, instead of the entire contents of the collection (though if the list box is sorted, every element in the collection must be accessed). However, as you scroll through the collection, list box entries are not discarded, which means that for large collections, the list box can contain an unacceptably large number of entries. For this reason, **listCollection** should be used only for small collections that will never contain too many items.

When you call **listCollection**, you specify **true** or **false** for an **update** parameter. If the **update** parameter in the **listCollection** method is **true**:

- Deleting the collection results in the list box or combo box being cleared and the collection is no longer associated with the list box or combo box.
- Any changes to the collection cause the contents of the list box or combo box to be discarded and the collection is rebuilt to the current display point (the current entry is reselected if it still exists).

If the **update** parameter is set to **false**, the list box or combo box is not updated and can contain out-of-date information.

The view schema makes use of **listCollection** in several of its forms (for example, the **zInitialize** method of **FormCommissionRate** and **FormLocationsList**). By setting the **update** parameter to **true**, the individual controls handle synchronization of the data they are displaying.

Note We assume that we will never have a large number of commission rates and locations. If this were not the case, use of the default **listCollection** would not be appropriate.

CollectionListBox Class

The **CollectionListBox** class in the **CommonSchema** presents an example of a subclassed control. It implements a **ListBox** subclass that can view a collection in subsets of its members. It implements the **listCollection** method (described earlier in this document) so that it presents the same interface as standard Jade list boxes and combo boxes. Once a collection has been registered with the **CollectionListBox** (using **listCollection**), it takes care of loading elements from the collection as required, depending on the scroll position (the entire collection is not loaded). As you scroll through a collection, members of the list box that are no longer visible are discarded. In this way, the **CollectionListBox** can view collections containing thousands of items without the actual list box contents ever growing to be too large. The **CollectionListBox** will register notifications on the collection and the members displayed in the list, so that it can automatically synchronize itself if they change.

While the **CollectionListBox** is capable of viewing very large collections, if the collection is very big, the time taken for the list box to position itself in the collection when scrolling can become quite noticeable. However, this occurs only with collections of thousands of elements and you would have to question the appropriateness of displaying that many entries in a list box in the first place.

Forms in the view schema make good use of the **CollectionListBox** to display information (for example, **FormAgentClientList** and **FormSaleItemCategoryList**). By using **CollectionListBox**, the forms do not need to worry about synchronizing this information. They can let the list box do it.

Object Notifications

If there are individual objects for which you want to implement specific behavior when they change (such as updating a view), you can use object notifications to manage that part of your caching strategy.

The **CollectionListBox** control class (described earlier in this document) in the **CommonSchema** uses object notifications to be informed of updates to objects that it is displaying, so that it can update itself if they change.

The **CollectionListBox** class begins object notifications in its **zLoadSubset**, **zLoadSubsetReversed**, and **zSetCollection** methods. An example is:

```
if showUpdates then
    // We want to be told about changes to this object
    beginNotification(obj, Object_Update_Event, Response_Continuous,
                        NotifyInstanceUpdate);
endif;
```

If the **obj** object is changed, the list box will receive a notification upon which it can update itself. Jade calls the **sysNotification** method when the notification is received.

Edition Checking

There are several forms in the **Administration** application that present an object to the user, enabling them to edit it. In a multiuser application, we must guarantee integrity by preventing two users from editing the same object at the same time, or by preventing the changes of one user being overwritten by the changes of another (who may have made his or her changes based on an obsolete object). There are several approaches, as follows.

- Share lock the object being edited as soon as the form is displayed. When the user goes to commit his or her changes, try to get an exclusive lock. If the exclusive lock cannot be obtained, display an error. This approach presents the problem that the object might be locked for a long period (for example, if the user takes a long time to make his or her changes, or goes out to lunch with the form open). It could also deadlock if two users, who both hold a share lock for an object, try to commit their changes at the same time (as neither user will be able to upgrade his or her share lock to an exclusive lock in order to update the object).

- Reserve lock the object during form initialization. This allows other users read access, but only we can upgrade the lock to an exclusive lock (which means that only we can update the object). Until we do so, other processes can still read the object. As with the share lock, this approach means that the object might be locked for a long period.
- Share lock and unlock, or resynchronize, the object during form initialization, and register a notification on it. If a notification is received while the user is editing, we display a message saying the object has been changed and discard the user's updates. The uses must start editing again. This approach introduces a timing hole in that a user may be able to commit his or her changes before the notification of an update arrives at the client from the server.
- Use edition checking (described in the following list). We have used this approach in the Erewhon system.

When presenting the user with a form to edit an object, the view does not keep a lock on the object in order to prevent it from being locked for a long period (potentially impacting concurrency). The form resynchronizes the object it is editing when it initializes using the **resynchObject** method (see **FormBase::zResynchObjectAndGetEdition** in **ErewhonInvestmentsViewSchema**).

We cannot allow an update to proceed if the object on which the user based his or her update is no longer current. We use edition checking to implement this, as follows.

1. The **zResynchObjectAndGetEdition** method synchronizes the object and saves its edition on the form.
2. When the form calls the required **TransactionAgent** method to perform the update, it passes in the saved edition (for example, see **FormAgent::zDoAction** and **FormClient::zDoAction**).
3. Each **TransactionAgent** method that receives a non-zero edition parameter first obtains an exclusive lock on the object to be updated. This brings the latest edition of the object into cache and locks it, thus preventing other users from updating it. We exclusively lock the object because we know we are about to update it. If the supplied edition is not equal to the latest edition of the object, we know that another user has changed it and we return **ObjectOutOfDate** to the caller. For examples of this, see the **TransactionAgent** methods **trxUpdateAgent** and **trxUpdateClient** in the model schema.

This approach gives us a good balance between ensuring that we do not process an out-of-date object, without requiring that the object be locked for the whole time the user is in the edit dialog.

Synchronization of Shop Views

The two shop views (all subclasses of **FormClientApp** in the view schema) implement searching and shopping cart facilities. Both of these features hold references to persistent **ModelEntity** objects during the session. As the shop view can be deployed on the web, we do not want to rely on notifications to synchronize the view. The shop views deal mainly with sale items, clients, categories, and locations. We expect such objects to be deleted only rarely, so have adopted a fairly straightforward approach of using exception handlers to trap object-not-found and object-deleted exceptions.

Each event method on **FormClientApp** and its subclasses arms a local exception handler at the start of the method. It then calls a non-event method to do the processing. If an invalid object is encountered, the exception simply resets the form, gives the user a message, and then resumes. For an example of this, see the **FormClientSaleItems::btnResultsDetails_click** and **FormClientApp::zInvalidObjectExHandler** methods.

Any exceptions not caught locally will be caught by the **commonExceptionHandler** method implemented in the **GCommonSchema** class. This exception handler method is armed globally when an application starts. The view schema reimplements this method in its **GErewhonInvestmentsViewSchema** class. For non-web applications, this reimplementation inherits the default behavior, which logs the exception and displays an error message box. For web applications, it simply aborts the current transaction and redisplays the last page.

Locking

Transactions protect against inconsistencies that can occur if something goes wrong within the transaction itself, but they can do so only within a single thread of execution. Whenever two or more database transactions are operating at the same time, there is the risk that they may interfere with each other by modifying the same objects.

Concurrency control is necessary, and for this we use locks.

To protect against inconsistencies, Jade provides mechanisms to lock objects. In Jade, a lock does two things. Firstly, it controls concurrent access to an object. Secondly, locking an object ensures that the latest edition of the object is brought into local cache in the node. In the Jade language, you can use the **exclusiveLock**, **sharedLock**, and **reserveLock** methods of the **Object** class to lock objects. The valid concurrent lock combinations are displayed in the following table.

	Exclusive	**Shared**	**Reserve**
Exclusive	No	No	No
Shared	No	Yes	Yes
Reserve	No	Yes	No

Exclusive locks are also known as *write* locks and shared locks are also known as *read* locks.

Locks can have two durations: session and transaction. Session locks are held until the end of the session (process/application) that acquired the lock or until the lock is explicitly released using the **unlock** method. Transaction duration locks are held until the end (either commit or abort) of the next transaction (at which point *all* transaction duration locks for the process are released) or until the lock is explicitly released using the **unlock** method when not in transaction state (manual unlocks of transaction duration locks within a transaction are ignored).

Ignoring explicit unlocks of transaction duration locks when in transaction state and releasing all transaction duration locks at the end of a transaction is known as two-phase locking. By doing so, Jade avoids the classic "assumed update" problem, by not allowing a second process to update objects modified by a first process until the first process has committed or aborted the entire transaction.

For examples of locking in the Erewhon Investments system, see the methods on the **TransactionAgent** class and **ModelEntity** subclasses in the model schema.

Exclusive Locks

Before an object can be updated, Jade insists that it be exclusively locked. This prevents two processes from updating the same object at the same time. An exclusive lock can be obtained only if there are no other locks in place for the object. When you lock an object using an exclusive lock, no other process can lock (and hence update) the same object. Jade automatically applies an exclusive lock when an object is updated. By default, updated objects are locked automatically for the duration of the transaction.

Shared Locks

A shared lock allows several processes to simultaneously read an object but not update it. Shared locks enable greater concurrency while ensuring that a process never works with obsolete data. If you lock an object using a shared lock, other processes attempting to update the object or explicitly acquire an exclusive lock wait until the lock is released, but can acquire a shared lock or a reserve lock.

Reserve Locks

A reserve lock is available for situations where you intend to update an object but you need to minimize the length of time the object is locked with an exclusive lock. When you place a reserve lock on an object, other processes attempting to acquire an exclusive lock or reserve lock on that same object wait until the reserve lock is relinquished, but those attempting to acquire a shared lock succeed.

Unlocking Objects

You can unlock objects manually. Use the **unlock** method to explicitly unlock an object. Requests to unlock transaction duration locks when in transaction state are ignored. All transaction duration locks are held until the next commit or abort transaction instruction, at which time they are all released, regardless of whether or not they were explicitly released with an **unlock**.

Inverses and Referential Integrity

A *reference* is a property that contains a reference to another object; that is, it is an end-point in a one- or two-directional relationship. The two types of reference in Jade are:

- An *implicit* reference, in which an object references another object and either of the following is true.
  - The referenced object does not contain a reference back to the first object.
  - The referenced object contains a reference to the first object, but the two properties have not been defined as end-points in a two-way relationship.
- An *inverse* (or *explicit*) reference, in which two objects reference each other and the two properties have been defined as end-points in a two-way relationship.

Inverse (or explicit) references are used in Jade to implement relationships between objects. They offer significant advantages in that Jade will automatically handle updating one side of a relationship (an inverse reference) whenever the other side changes. In addition, if one or both ends of a relationship is a dictionary, related elements in the dictionary are automatically updated whenever their keys change. This helps to ensure referential integrity in your model. In fact, in a persistent model, inverse (or explicit) references should be the rule. There should be few cases where they are not used, and in such cases, a good reason for not using them.

In Jade, a reference can refer to a single object, or to multiple objects (via a collection). This allows you to implement one-to-one, one-to-many, and many-to-many relationships. Relationships can be defined as peer-to-peer or parent-child. They differ only when objects are deleted. A parent-child relationship allows you to implement a cascading delete where all related children of a parent object are deleted when the object itself is deleted. In a peer-to-peer relationship, when one object is deleted, all references to it in its related objects are removed (set to null).

Jade allows a reference to have multiple inverses (that is, participate in multiple relationships). In such cases, Jade will automatically propagate updates on a single reference to multiple inverse references.

The **ErewhonInvestmentsModelSchema** employs inverse references extensively. Some examples are as follows.

- One-to-One Relationships
- One-to-Many Relationships
- Many-to-Many Relationship
- Parent-Child Relationships

- Multiple Inverse Relationships
- Automatic Key Maintenance

One-to-One Relationships

SaleItem::mySale to **Sale::mySaleItem**

TenderSale::myTender to **Tender::myTenderSale**

One-to-Many Relationships

Company::allAgents to **Agent::myCompany**

SaleItemCategory::allCommissionRates to **CommissionRate::mySaleItemCategory**

Many-to-Many Relationship

Agent::allCommissionRates to **CommissionRate::allAgents**

Parent-Child Relationships

Company::allClients to **Client::myCompany** (a one-to-many relationship)

Country::allRegions to **Region::myCountry** (a one-to-many relationship)

Parent-child relationships are what allow the **JadeScript** method **deleteAllData** to purge the database by simply deleting the **Company**. Jade cascades the delete through all of the parent-child relationships in the model.

Multiple Inverse Relationships

Tender::myTenderSaleItem to **TenderSaleItem::allTendersByOfferTime**

Tender::myTenderSaleItem to **TenderSaleItem::allTendersByTimeOffer**

Whenever **myTenderSaleItem** is set on a **Tender**, the **Tender** is added to both the **allTendersByOfferTime** and **allTendersByTimeOffer** dictionaries on the sale item.

Sale::myClient to **Client::allTenderSales**

Sale::myClient to **Client::allRetailSales**

These illustrate a conditional multiple inverse relationship. Whenever **myClient** is set on a **Sale**, the **Sale** is added to **allTenderSales** on the **Client** if it is a tender sale (because membership of the **allTenderSales** dictionary is **TenderSale**) and **allRetailSales**, if it is a retail sale (because membership of the **allRetailSales** dictionary is a **RetailSale**).

Automatic Key Maintenance

Client::myCompany to **Company::allClients**

In the above one-to-many relationship, if the name of a client is changed, the **allClients** dictionary of the **Company** to which the client is related will automatically be updated.

SaleItem::myCompany to **Company::allSaleItems**

In the above one-to-many relationship, if the code prefix or code number of the sale item changes, the **allSaleItems** dictionary of the sale item's company will automatically be updated.

Key Paths

A key path is a mechanism that enables you to define a dictionary key that is not an embedded property of the members of the dictionary, but is instead derived from the member objects. When you define a key path, you specify a chain of references starting from the member class and finishing at an end-point. At run time, the references are traversed to arrive at the end-point that yields the key value. Like all dictionary keys, if the dictionary participates in a relationship, changes to key path keys will automatically be propagated to the related dictionaries.

The **ErewhonInvestmentsModelSchema** has several dictionaries that make use of keys paths. Three of them are:

- **SaleByItemDict**
- **RetailSaleByTimeItemDict**
- **TenderSaleByTimeItemDict**

Server Methods

The **serverExecution** method option indicates that the method and all methods subsequently called by this method are to be executed at the database server node (unless they are **clientExecution** methods, in which case they are executed at the node of the client calling the method).

By simply adding **serverExecution** to a method signature, Jade will shift execution of the method to the database server node. This method option provides performance benefits (by reducing network traffic) when a method accesses a large number of persistent objects in multiuser mode. The methods are executed at the node in which the objects reside, rather than the required objects having to be passed across the network to the client node for processing.

See the **TransactionAgent::trxCloseTendersAtDate** and **Company::closeTendersAtDate** methods in the model schema for an example of **serverExecution**. These methods are used to close all open tender sale items at a specified date, and as such, we expect that they may reference a large number of objects. As this is a *batch*-type operation and there is no requirement for them to be processed at the client node, we implement this transaction as a server method to avoid all of the tender sale items and their associated tenders and related objects having to be brought across the network.

Server methods are great for distributing code to reduce network traffic. However, be aware of the following restrictions.

- Transactions must be total client transactions or total server transactions; that is, any begin and commit transaction pair of instructions must be done while executing on the client (without executing an updating server method), or while executing on the server (without the execution of updating client methods).
- Persistent transactions must be started, performed, and finalized at a single node. All of the update operations of the transaction must occur in the same node that started the transaction.
- Server methods cannot invoke GUI methods.

In the **ErewhonInvestmentsModelSchema**, it is for the first two reasons that the **InitialDataLoader loadData** method commits the first transaction before invoking the **zCloseTendersAtCurrentDate** method (which begins and commits a transaction on the server).

Tender closures are performed in a server method, so they require a separate transaction on the server. The first transaction, which we begin on the client, must be committed on the client before we start the server transaction.

Skins

A *skin* is a series of images that is applied to the caption line, menu line, and border areas of each form to provide an enhanced look and feel. The skin can also define images for most controls, to further enhance the look and feel of forms.

Jade provides a collection of skins for the Jade Platform development environment and a global collection that contains any user-defined skins for all schemas.

The Erewhon **Administration** application provides a Skins menu that is populated dynamically with the names of all skins that are present in the system. With this menu, users can select the skin they want applied to the application. To see how this is implemented, see the **zSetupSkinSelectMenu** and **mnuSkin_click** methods on the **FormAdminMdi** class in **ErewhonInvestmentsViewSchema**.

Part 5 Transaction Agent Framework

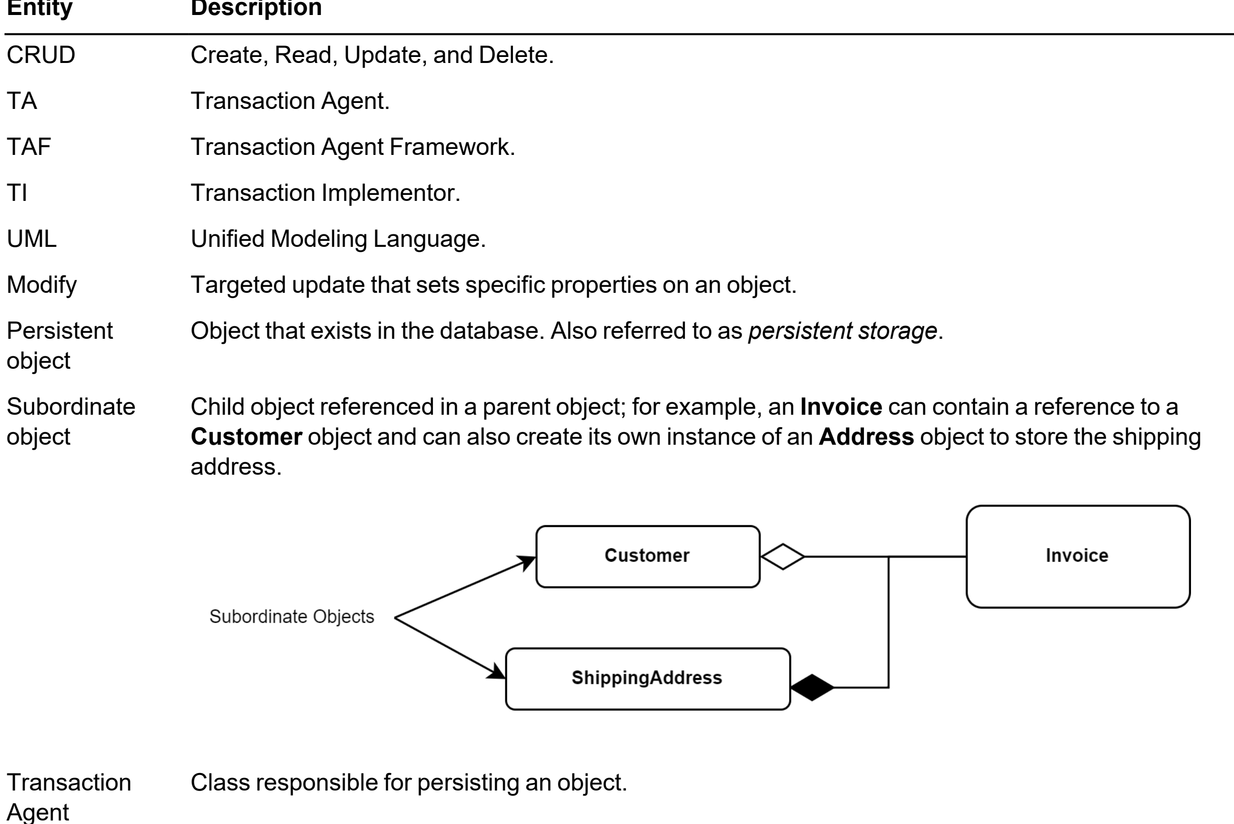
This section discusses the recommended approach to persisting objects using the Jade Platform and it provides examples of a Transaction Agent Framework (TAF) that you can use in your own applications.

By the end of this section, you should be able to implement all create, update, and delete functionality using a TAF in your own Jade applications.

The code examples shown in this section are written using the 22.0.01 release version of the Jade Platform. (Code example images enclosed in a solid (unbroken) border indicate a complete Jade method, whereas a code example enclosed in a jagged (broken) border indicates a code fragment.)

The following table lists the acronyms (pronounced as a word), initialisms (pronounced as a series of letters that are abbreviations derived from the letters of the words they represent), and terms used in this section.

Entity	Description
CRUD	Create, Read, Update, and Delete.
TA	Transaction Agent.
TAF	Transaction Agent Framework.
TI	Transaction Implementor.
UML	Unified Modeling Language.
Modify	Targeted update that sets specific properties on an object.
Persistent object	Object that exists in the database. Also referred to as *persistent storage*.
Subordinate object	Child object referenced in a parent object; for example, an **Invoice** can contain a reference to a **Customer** object and can also create its own instance of an **Address** object to store the shipping address.



Entity	Description
Transaction Agent	Class responsible for persisting an object.

For details about the Transaction Agent Framework (TAF), see the following subsections.

Applies to Version: 2022.0.01 and higher

Best Practice Guidelines

The following table lists some recommended best practices and coding styles recommended when developing Jade applications.

Entity	Guideline
Collection references	References to collections should be prefixed with **all**. For example, a **Client** class may have a reference to a collection of sales. The reference to the **Sales** collection in the **Client** class should be named **allSales**.
Global constants	Global constants should be used in place of arbitrary values. For example, rather than referring to exception 1048 (*Update outside transaction*), you can create an **Exceptions** category for your schema in the Global Constants Browser and then create a global constant called **UpdateOutsideTransactionException** with a 1048 definition. Global constants can be accessed using the Ctrl+G keyboard shortcut.
Inverse references	Inverse references automatically delete child objects when the parent object is deleted. In most cases, the manual update is on the **my** reference and the automatic update is on the **all** reference. (It can be useful to remember this as "**M**=Manual and **My A**=Automatic/All".) When the single reference is updated, the collection is automatically updated. It is recommended that you create inverse references whenever a parent object owns a child object or collection of objects.
Object references	References to objects should be prefixed with **my**. For example, a **Sale** class that contains a reference to a **Client** object should be named **myClient**.
Property / properties	Generic name to represent an attribute or reference property.
Self	Use the **self** system variable (keyword) when referencing a property, method, or control from within the same class instance.
Separation of concerns	Methods should be responsible only for performing a specific task. If a method is performing more than one task, it may need to be split into one or more methods. Separating logic improves code readability, unit testing activities, and debugging.
Transient objects	A *transient* object is an object that is local to the Jade process and cannot be created or accessed by another Jade process. Transient objects are stored in a transient cache. When the transient cache gets full, the least-recently used transient objects overflow to an unbound transient database. Because this database is unbound, transient leaks can cause significant amounts of disk space to be used. For this reason, transient objects must be manually deleted. Deleting transient objects is usually performed in the epilog of the method that created the transient object.

Transaction Agent Framework (TAF) Overview

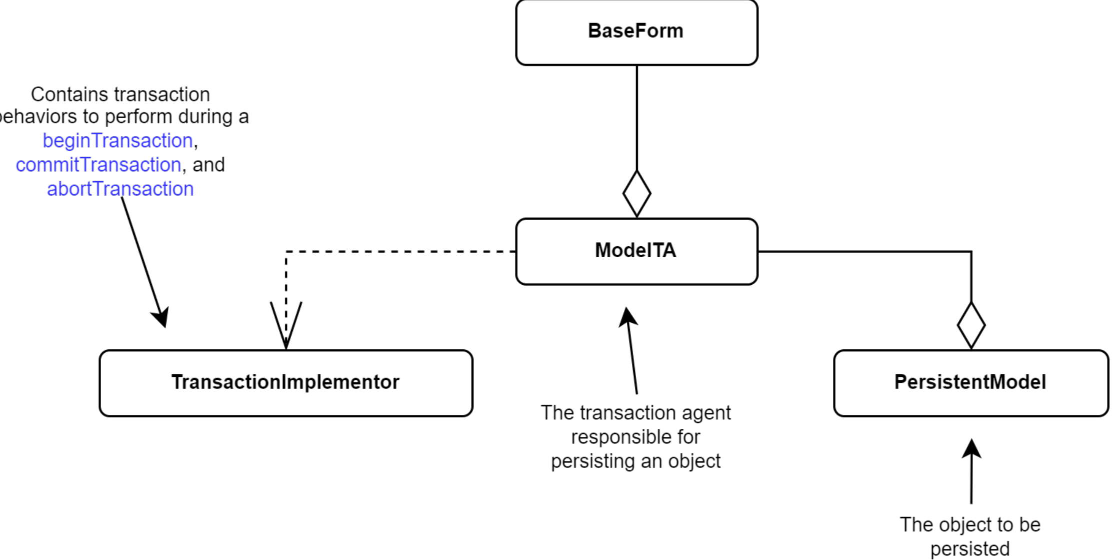
This section provides a basic high-level overview of the framework. Detailed examples and descriptions of the methods used in this section are provided in later sections of this document.

- What is the Transaction Agent Framework (TAF)?
- Why is a TAF Needed?
- Where Should the TAF Reside?
- How Does the TAF Work?

What is the Transaction Agent Framework (TAF)?

The Transaction Agent Framework (TAF) is a set of classes that work together to perform persistent storage of objects.

The four main classes of the TAF are **PersistentModel**, **ModelTA**, **TransactionImplementor**, and **BaseForm**. As the following diagram shows, these classes work together to commit transactions into a Jade database.



The following table summarizes the four main classes.

Class	Description
PersistentModel	Subclasses of this class represent the objects to persist. They could represent a physical entity such as a **Person** or a concept like a **Sale**, which are essentially objects being saved in the database.
ModelTA	Subclasses of this class are the transaction agents responsible for performing persistent operations. Each **PersistentModel** subclass requires its own transaction agent subclass to be created. For example, when creating a transaction agent for a **Client** object, the transaction agent should be named **ClientTA**. The **ClientTA** class will contain the same properties.
TransactionImplementor	These classes are injected into various transaction agent methods through dependency injection, to determine the type of transaction behavior to perform; for example, begin, commit, and abort operations.
BaseForm	This class contains properties and methods used to connect forms and user interfaces with the TAF. All child forms should inherit this class, and therefore it should be the top-level class of the **Form** object. The **BaseForm** class may not be needed in situations where data is being persisted programmatically; for example, when loading data from a text file or in a situation where a user interface may not be required. For details, see "Manually Persisting an Object" under "How Does the TAF Work?", later in this document.

Why is a TAF Needed?

Software requirements are constantly changing, so new methods and properties may need to be introduced at any stage of development. What the client initially requested is often different from their actual needs.

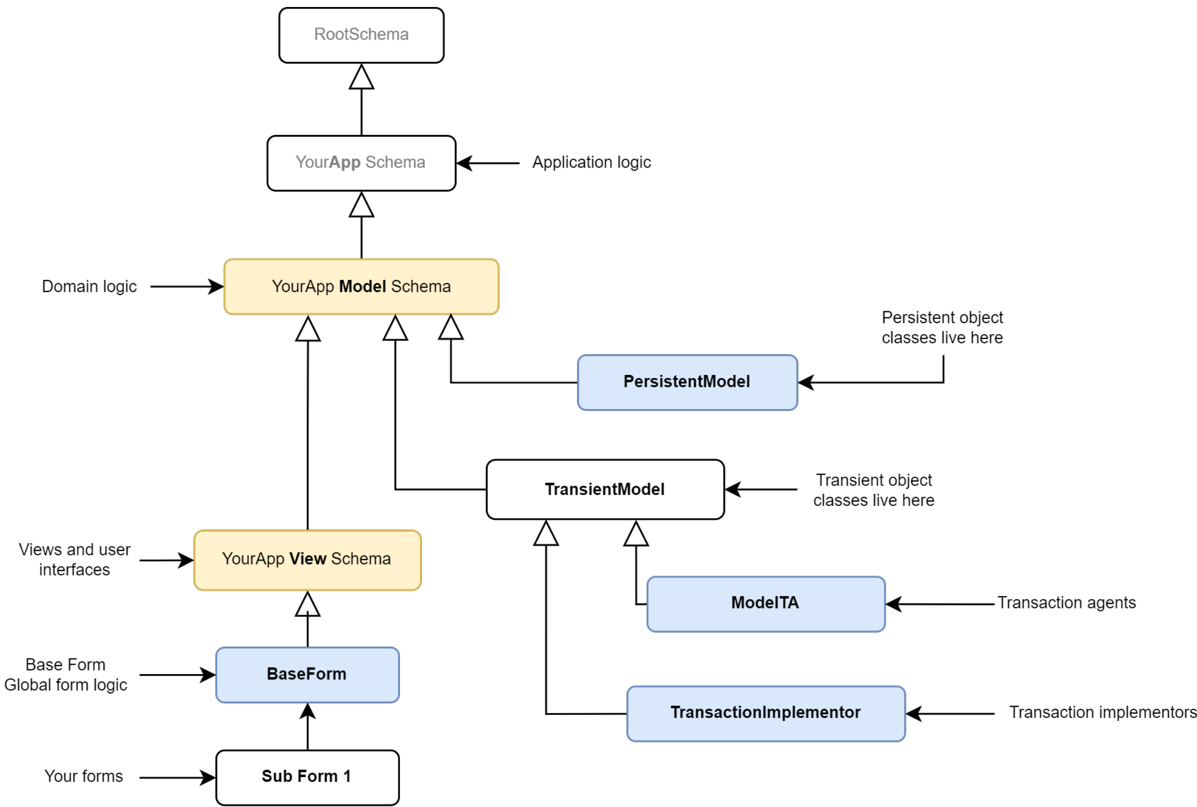
Due to the high possibility of change, software should be written in a way that is open for extension and closed for modification. Simply put, we don't want to modify existing working code in order to develop additional requirements; instead we want to extend the code base with the new logic. This is referred to as the *Open/Closed Principle*.

The Transaction Agent Framework does not use parameterized functions or constructors to create objects, because adding additional parameters when requirements change is likely to cause changes that break existing code. Instead, the TAF creates objects based on the properties stored in the transaction agent class. This method of creating objects not only eliminates parameterized functions and constructors but allows additional properties and logic to be added into the code base without affecting the original code.

Where Should the TAF Reside?

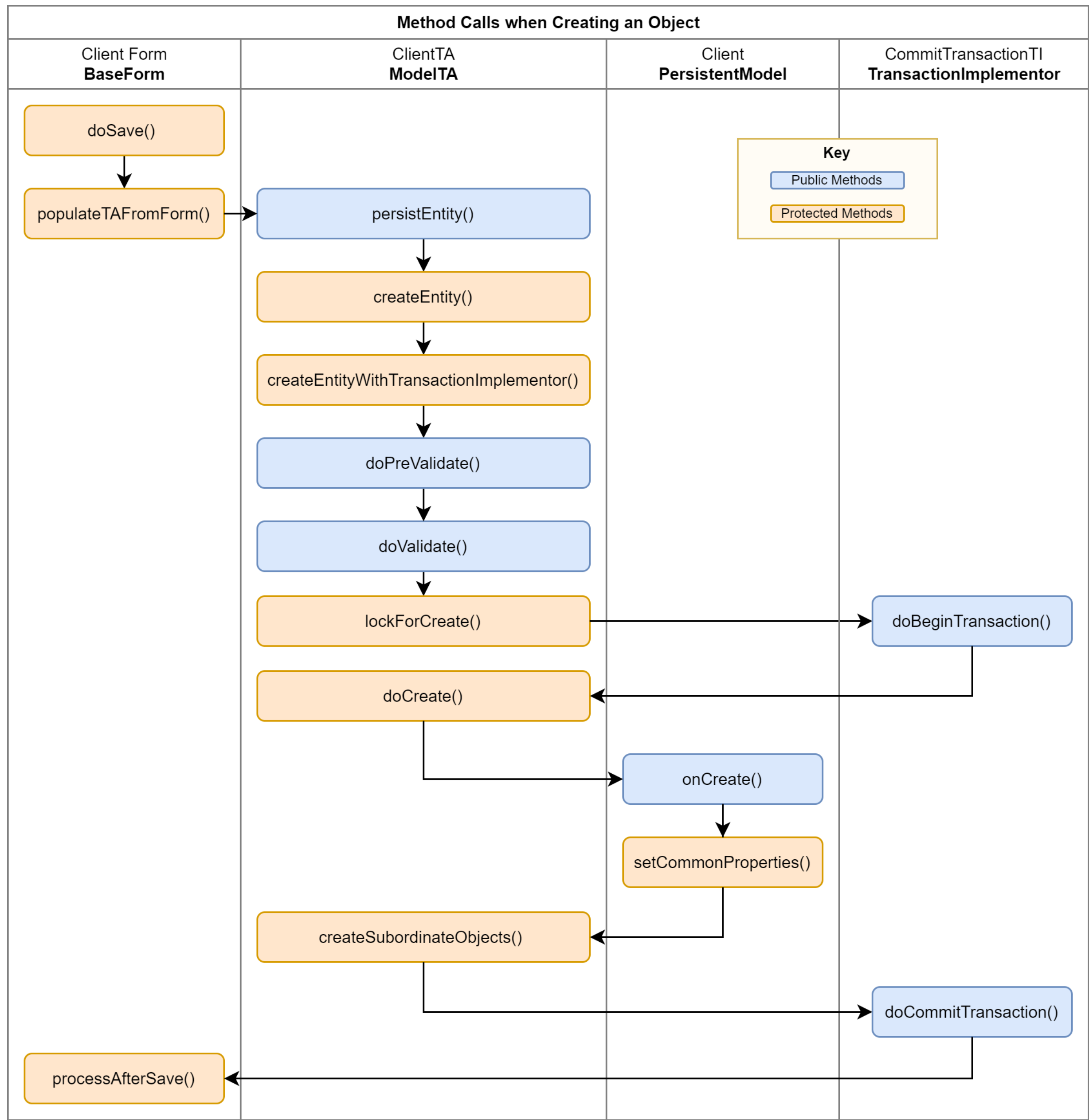
The TAF should be part of the **Model** schema (except for the **BaseForm** class residing in the **View** schema).

The following diagram shows a recommended setup of the Transaction Agent Framework in a standard Jade application.



How Does the TAF Work?

The following diagram shows the process of a **Client** object being created and persisted using the TAF.



Details about the methods in this diagram are described later in this document.

Manually Persisting an Object

The transaction agent framework does not have to use the **BaseForm** class to persist objects, as it can be done manually and may be required when creating unit tests or loading data from a text file.

To persist an object without using the **BaseForm** class, transaction agents need to be created as a transient object and populated manually by setting the properties directly or by passing in an object into the **ModelTA** class **populateFromObject** method.

The following method is an example of how to persist an entity by manually populating the transaction agent properties.

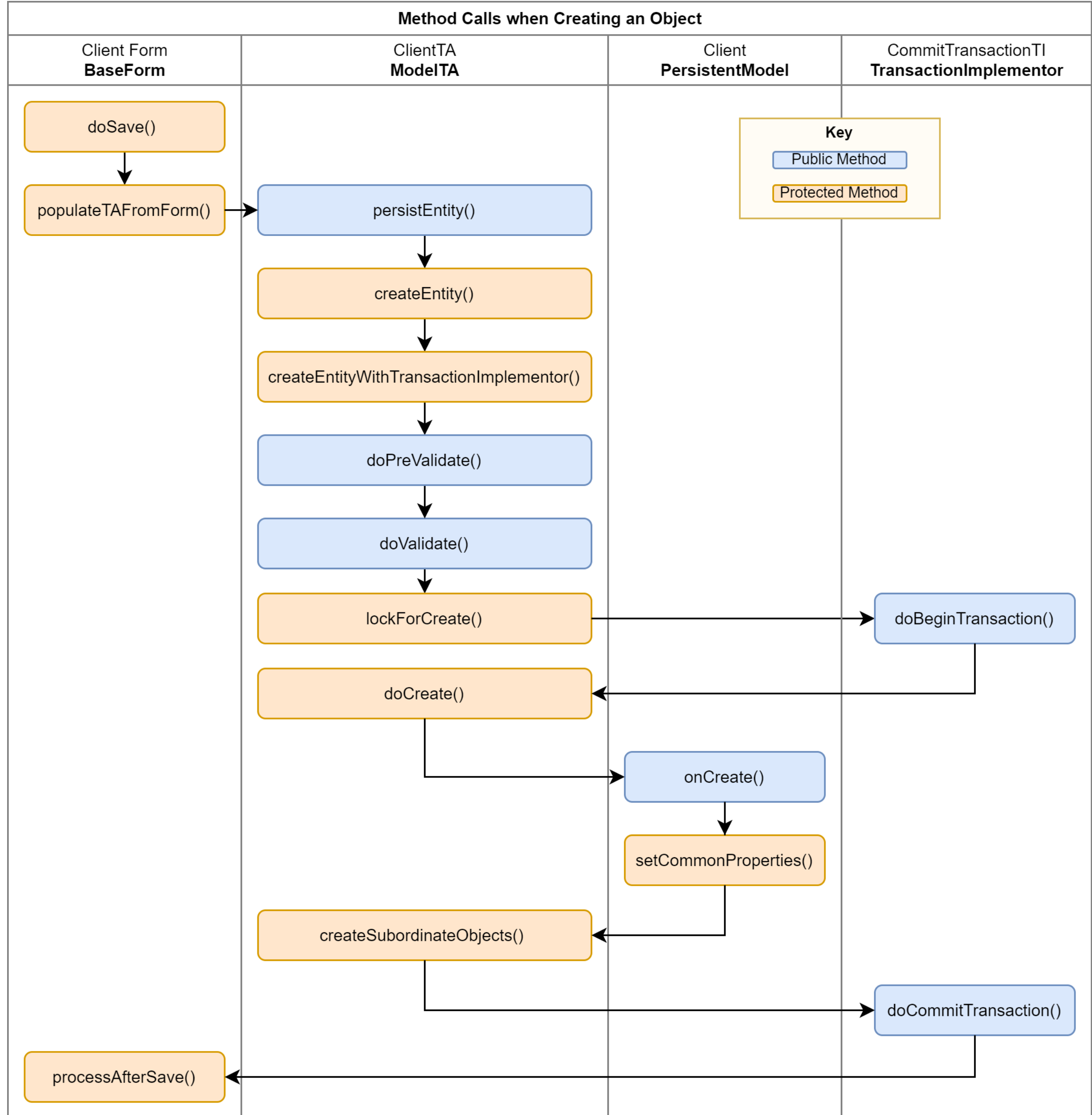
```
createClient();

vars
    clientTA : ClientTA;
    addressTA : AddressTA;
begin
    create clientTA transient;
    create addressTA transient;

    clientTA.name := "Clive Entworth";

    addressTA.street := "555 Fake St.";
    addressTA.city := "Dunedin";
    addressTA.country := "New Zealand";
    addressTA.phone := "555 5555555";
    addressTA.fax := "555 4444444";
    addressTA.email := "CliEnt@E.mail";
    addressTA.webSite := "www.website.com";
    addressTA.myModelTA := clientTA;

    clientTA.persistEntity( TransactionType_Persist );

epilog
    delete clientTA;
end;
```

The above example deletes the **ClientTA** transient in the epilog but does not delete the **AddressTA**. This is because the **AddressTA** transient object will be automatically deleted because of the parent/child relationship being set when the reference was created. One of the major benefits of the TAF is ensuring that references are always inversed correctly.

Caution Remember to delete transient objects! Transient objects are stored in a transient cache. When the transient cache gets full, the least-recently used transient objects overflow to an unbound transient database. Because this database is unbound, transient leaks can cause significant amounts of disk space to be used. For this reason, transient objects must be manually deleted, which is usually done in the epilog of the method that created the transient object.

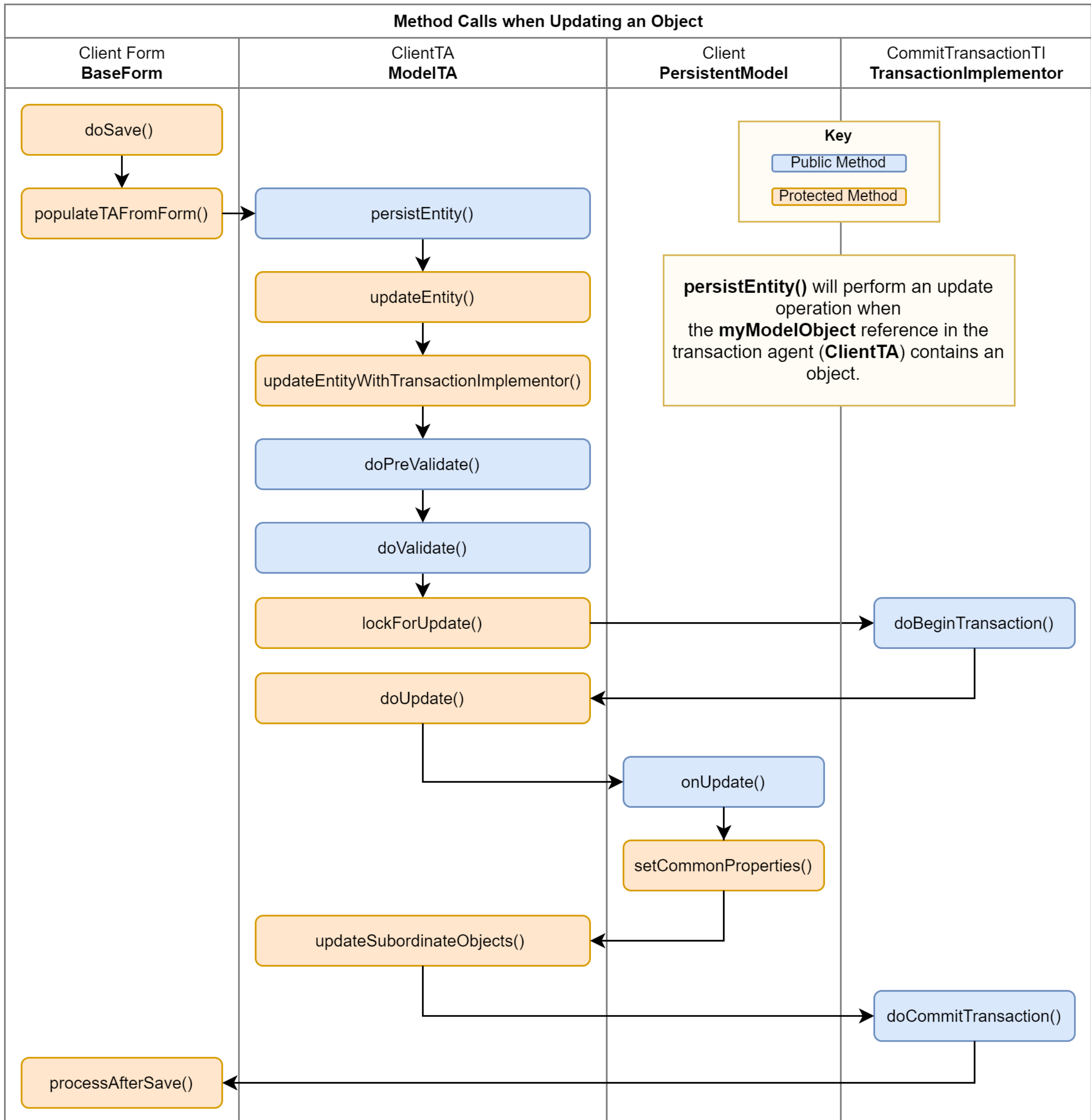
Creating Objects using the BaseForm Class

Persistent objects should be created using the TAF, by calling the **BaseForm** class **doSave** method.



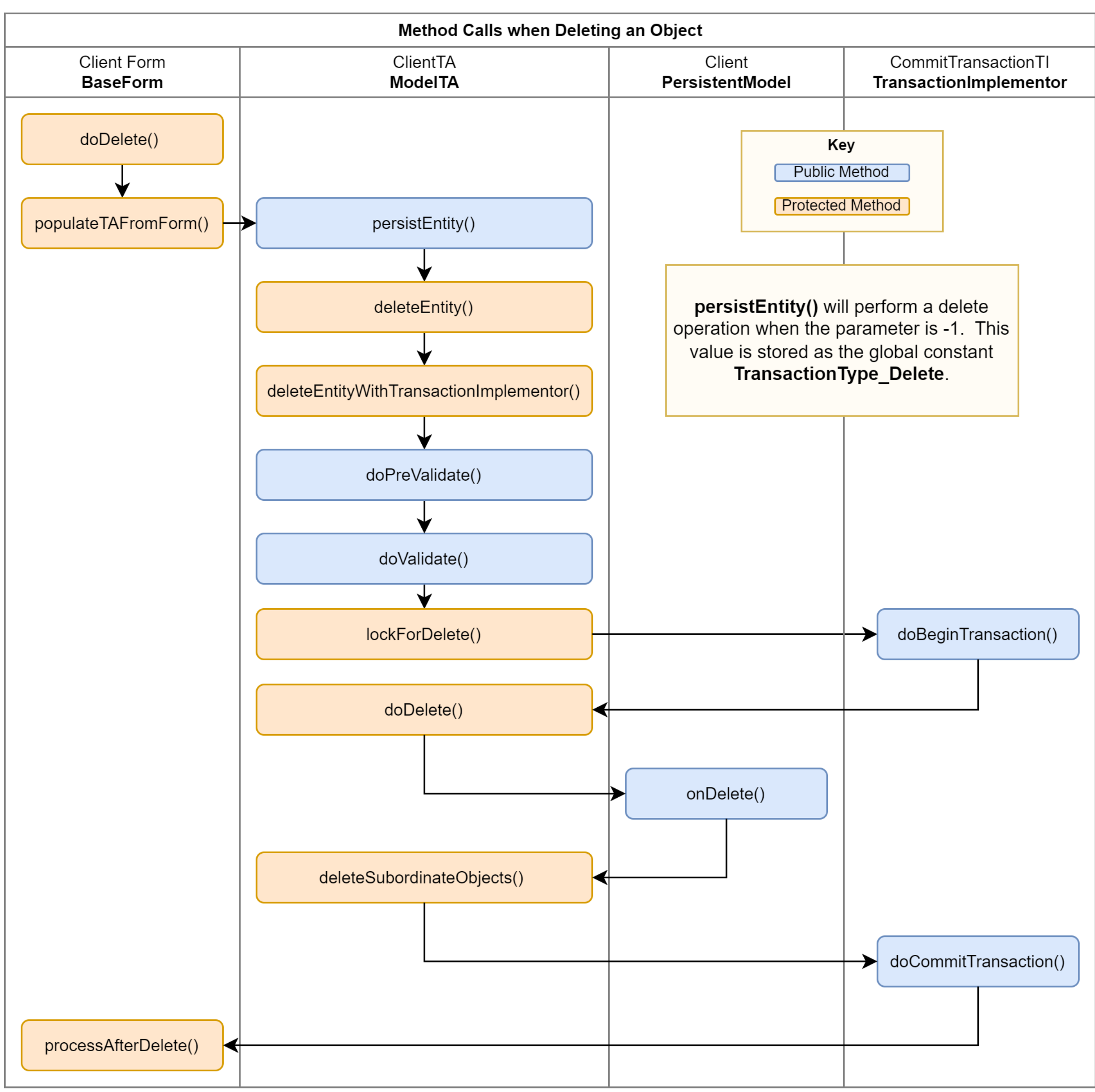
Updating Objects using the BaseForm Class

Updating objects is performed in exactly the same way as creating an object, by calling the **doSave** method on the **BaseForm** class. The transaction agent will check the value of the **myModelObject** parameter in the **persistEntity** method and perform an update if the model object value is not null.



Deleting Objects using the BaseForm Class

Deleting objects is performed by calling the **BaseForm** class **doDelete** method.



Reading Data

The transaction agent is not responsible for read operations when the object does not require persistence. Objects being used for display purposes should be retrieved using standard Jade practices.

Returning a Single Collection Item

To return a single client, you can use the **Dictionary** class **getAtKey** method and pass in the key value.

The following example shows the shorthand way of calling the **getAtKey** method.

```
app.myCompany.allClients["client-name-here"];
```

Populating a List Box

To populate a list box with a collection of clients, you can use the following code.

```
self.lstClients.listCollection(app.myCompany.allClients, true, 0);
```

The **listCollection** method in the **ListBox** or **ComboBox** classes enables list box or combo box controls to have a collection attached to them. Use the **displayCollection** method to automatically attach only as many entries as required to fill the list of the control.

The differences between the **listCollection** method and the **displayCollection** method are as follows.

- The **listCollection** method retains all entries added to the list or combo box when the user scrolls the view.
- For the **listCollection** method, the number of entries in the list (returned by the **listCount** method) is logically the size of the attached collection minus any discarded entries.
- For the **displayCollection** method, the **listCount** method returns only the number of entries that are displayed.
- The **displayCollection** method enables you to specify a starting object.

Populating a Table

To populate a table, use the **displayCollection** method, as follows.

```
self.tblTenders.displayCollection(self.myTendersSearch.allTenders,
                                  false, null, null);
```

The value to be displayed is set in the **displayRow** method of the table, as shown in the following example.

```
tblTenders_displayRow(table: Table input;
                      theSheet: Integer;
                      pTender: Tender;
                      theRow: Integer;
                      bcontinue: Boolean io):String updating;
begin
    // Color closed tenders red
    if pTender.getStatus() = Closed then
        table.accessRow( theRow ).foreColor := Red;
    endif;

    // Tender item was Successful
    if pTender.myTenderSale <> null then
        table.accessRow( theRow ).foreColor := Green;
    endif;

    return pTender.getStatus() & Tab &
            pTender.getDate().String & Tab &
            pTender.myTenderItem.closureDate.String & Tab &
            pTender.myClient.name & Tab &
            pTender.myTenderItem.getCode() & Space & Colon & Space & pTender.myTenderItem.name & Tab &
            pTender.myTenderItem.price.currencyFormat() & Tab &
            pTender.offer.currencyFormat() & Tab &
            pTender.myTenderItem.myAgent.name;
end;
```

Locking Objects

You should exclusive-lock persistent objects in a consistent order, including any collections which will be updated by automatic inverse maintenance, to avoid deadlocks when multiple processes attempt to update the same object or attempt to create, update, or delete instances of the same class. Obtaining the exclusive locks in a consistent order will ensure other processes queue behind the process that currently holds the exclusive locks, therefore removing the opportunity for a deadlock to occur.

An example of when locking may be required is an application requiring objects to remain unmodified while an operation is carried out; for example, a trial balance in which account objects are share locked before reading the balance, to guarantee that the latest edition of each account is used. The shared locks are held until the trial balance calculation is complete.

Note Share locking an object does not prevent other processes accessing it, but it *does* prevent them updating it.

Locking a Collection

Locking a collection is used to avoid objects being added or removed, or keys changing in a dictionary collection (member objects in a dictionary can always be updated). A collection should be exclusive locked when performing create, update, and delete operations if a property in the respective object is being used as a key or inverse.

To find the relevant collections for an object, select the *key* property (that is, it will have the key symbol displayed at the left of the property name) in the Properties List of the Class Browser and then click the **Property Details** tab above the editor pane, paying particular attention to the **Used as a key in** and **Inverses** details.

Locking Collection Objects Before a Create Action

All collections that will be updated by automatic inverse maintenance to add the object being created into that collection need to be exclusive locked in a consistent order, to remove the opportunity for a deadlock to occur. This is especially important if the object is generating unique identifiers from the collection.

There is no need to lock collections that use a key path (Key Path is a dictionary key that is not embedded on the object but derived from member objects) for the created object (for example, a **Category** object) because no other objects will reference the new object in their key path and no updates to these collections will happen.

Locking Collection Objects Before a Delete Action

Any collections that contain the object being deleted will need to be exclusive locked in a consistent order prior to deleting the object, to remove the opportunity for a deadlock to occur.

If the object to be deleted is being used as a key path, a check should be performed to ensure that no references to the object exist before the object is deleted. For this reason, there is no need to lock collections that use the object as a key path.

Locking Collection Objects Before an Update Action

If an updated property in the object is used as a key path, any collections that use that key will require exclusive locking; otherwise locking the collections is not required.

PersistentModel Class

The **PersistentModel** class contains subclasses that represent an entity or abstract concept, and it requires database persistence.

Classes should be singularly named, as each instance represents a single object persisted in a database. For example, a class representing an employee should be named **Employee**; not **Employees**.

The following class diagram shows a persistent **Client** class using the TAF in the Erewhon schema.

PersistentModel

+ lockedByProcessId : Decimal {readonly}

+ getTAClass( ) : Class abstract
+ isSoftLockedByMe( ) : Boolean
+ onCreate(pTA : ModelTA)
+ onDelete(pTA : ModelTA)
+ onModify(pTA : ModelTA)
+ onUpdate(pTA : ModelTA)
# setCommonProperties (pTA : ModelTA)
+ lockedTimeStamp : TimeStamp

Client

+ allRetailSales : RetailSaleByTimeItemDict {readonly}
+ allTenderSales : TenderSaleByTimeDict {readonly}
+ allTenders : TenderByItemTimeOfferDict {readonly}
+ myAddress : Address {readonly}
+ name : String {readonly key}

+ getAllSales (saleSet : SaleSet input)
+ getDebugString( ) : String
+ getNameAndAddress( ) : String
+ getTAClass ( ) : Class (r)
+ onCreate(pClientTA : ClientTA) (r)
# setCommonProperties (pTA : ClientTA) (r)

Three methods in the **Client** class are reimplemented from the **PersistentModel** class. They are **getTAClass**, **onCreate**, and **setCommonProperties**.

The **getTAClass** method must be reimplemented in each class to return the appropriate subclass of the **ModelTA** transaction agent class used for persistent transactions.

PersistentModel Code Implementation Examples

The following code examples show the use of the **PersistentModel** inherited methods in a **Client** class using the Erewhon Investments application.

getTAClass

The inherited **getTAClass** method returns the transaction agent responsible for performing the persistent database operations.

```
getTAClass() : Class;

begin
    return ClientTA;
end;
```

onCreate

The inherited **onCreate** method calls the inherited **onCreate** method to set the common properties from the transaction agent, then sets the **Client** class **myCompany** property.

```
onCreate( pClientTA : ClientTA ) updating;

begin
    inheritMethod( pClientTA );

    self.myCompany := app.myCompany;
end;
```

This method is called only when creating an object. Properties that need to be set once only or methods that are executed only at object creation should be coded in this method.

setCommonProperties

The inherited **setCommonProperties** method calls the inherited **setCommonProperties** method first and it then populates the model object properties from the transaction agent.

```
setCommonProperties( pClientTA : ClientTA ) updating, protected;

begin
    inheritMethod( pClientTA );

    self.name := pClientTA.name;
end;
```

This method is called when creating or updating objects.

PersistentModel Properties

The read-only properties defined in the **PersistentModel** class are summarized in the following table.

Property	Type	Holds the...
lockedByProcessId	Decimal	**Process** identifier that soft-locked the object. The value is automatically set by the **ModelTA** value specified in the **PersistentModel** class **onModify** method parameter.

Property	Type	Holds the...
lockedTimeStamp	TimeStamp	Timestamp of when the object was soft-locked. The value is automatically set by the **ModelTA** value in the **PersistentModel** class **onModify** method.

PersistentModel Methods

The methods defined in the **PersistentModel** class are summarized in the following table.

Method	Description
getTAClass	Called when the transaction agent class is required for an existing persistent object.
isSoftLockedByMe	Checks if the current process has soft-locked the object.
onCreate	Called when creating an object. It is used to set the model properties that do not change.
onDelete	Called when deleting an object, to delete the persistent object from the database.
onModify	Called when an object is modified. A modify operation is a targeted update that only sets some of the properties on an object.
onUpdate	Called when an object is updated. An update operation sets the properties on an object.
setCommonProperties	Called during a create or update operation to set the model property values by copying the properties from the transaction agent.

getTAClass

The public **getTAClass** method in the **PersistentModel** class returns a **Class**value. It is called when the transaction agent class is required for an existing persistent object.

This abstract method must be implemented on each subclass to return the transaction agent subclass (the **ModelTA** class type) that the framework should use for persisting that class.

Base Implementation

```
getTAClass(): Class abstract;
```

isSoftLockedByMe

The public **isSoftLockedByMe** method in the **PersistentModel** class returns a Boolean value. It checks if the current process has soft-locked the object.

This method returns **true** if the current process has soft-locked the object; otherwise it returns **false**.

Base Implementation

```
isSoftLockedByMe() : Boolean;

begin
    return self.lockedByProcessId = self.getInstanceIdForObject( process );
end;
```

onCreate

The public **onCreate** method in the **PersistentModel** class is called when creating an object. It is used to set the model properties that do not change.

A use case of this method is setting a **self.myCompany** reference to **app.myCompany**.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The **ModelTA** class that persists the object. Subclass reimplementation of this method changes the parameter to a more-specific subclass of the **ModelTA** class.

Base Implementation

```
onCreate( pTA : ModelTA ) updating;

begin
    self.setCommonProperties( pTA );
end;
```

onDelete

The public **onDelete** method in the **PersistentModel** class is called when deleting an object, to delete the persistent object from the database.

Additional work required when deleting an object can be implemented in this method.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The **ModelTA** class that initiated the delete operation. Subclass reimplementation of this method changes the parameter to a more-specific subclass of the **ModelTA** class.

Base Implementation

```
onDelete( pTA : ModelTA ) updating;

begin
    delete self;
end;
```

onModify

The public **onModify** method in the **PersistentModel** class is called when an object is modified. A modify operation is a targeted update that only sets some of the properties on an object.

You can implement additional work during object modification in this method.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The **ModelTA** class that initiated the modification. Subclass reimplementation of this method changes the parameter to a more-specific subclass of the **ModelTA** class.

Base Implementation

```
onModify( pTA : ModelTA ) updating;

begin
    if pTA.modificationCode = TransactionType_Modify_LockEntity then
        self.lockedByProcessId  := pTA.lockedByProcessId;
        self.lockedTimeStamp    := pTA.lockedTimeStamp;
    elseif pTA.modificationCode = TransactionType_Modify_UnlockEntity then
        self.lockedByProcessId  := null;
        self.lockedTimeStamp    := null;
    endif;
end;
```

onUpdate

The public **onUpdate** method in the **PersistentModel** class is called when an object is updated. An update operation sets the properties on an object.

Reimplement this method only if there is something other than the common properties that needs updating or there are specific notifications to be implemented.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The **ModelTA** class that initiated the update operation. Subclass reimplementation of this method changes the parameter to a more-specific subclass of the **ModelTA** class.

Base Implementation

```
onUpdate( pTA : ModelTA ) updating;

begin
    self.setCommonProperties( pTA );
end;
```

setCommonProperties

The protected **setCommonProperties** method in the **PersistentModel** class is called during a create or update operation to set the model property values by copying the properties from the transaction agent.

Reimplement this method to set the specific properties that are defined on that subclass and that should be set on a create or update. Be sure to include an **inheritMethod** instruction so any superclass implementations are called.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The **ModelTA** class from which to copy values. Subclass reimplementation of this method changes the parameter to a more-specific subclass of the **ModelTA** class.

Base Implementation

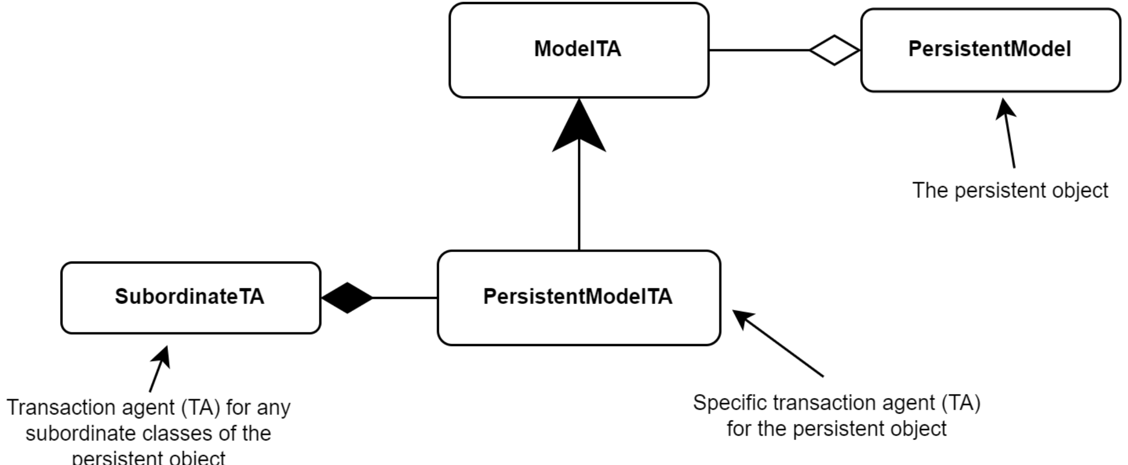
```
setCommonProperties( pTA : ModelTA ) updating, protected;

begin
    self.lockedByProcessId  := null;
    self.lockedTimeStamp    := null;
end;
```

ModelTA Class (Transaction Agent)

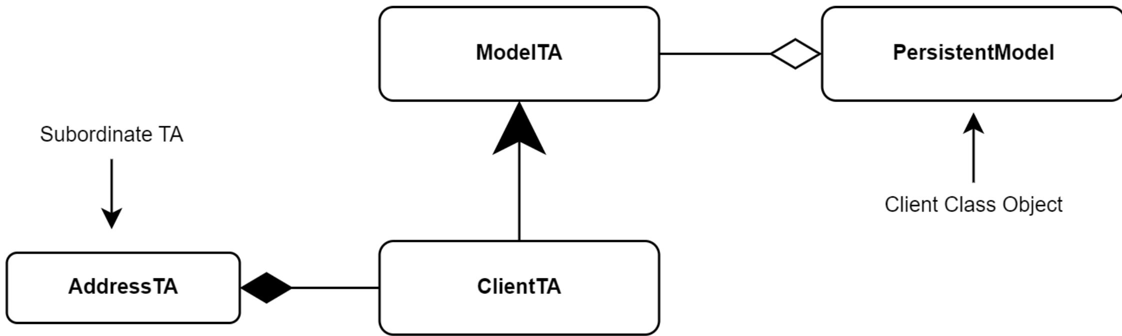
The transaction agent framework is responsible for communication between the views and the database. Persistent operations such as create, update, modify, and delete must be executed using a transaction agent.

Each class that is stored as persistent data should have a transaction agent class containing all of the properties of the **PersistentModel** child class other than automatically maintained inverse references.

The following diagram shows an abstract implementation of the **ModelTA** class and its relationships with other classes.



To better present this concept, an actual implementation of the **ModelTA** class for a **Client** class is shown in the following diagram.



SubordinateTA (AddressTA)

Subordinate transaction agents (TAs) are used to create, update, modify, and delete subordinate objects in the parent class. In the above diagram, the **ClientTA** class has a subordinate transaction agent named **AddressTA**, which is responsible for persisting an **Address** object.

PersistentModel

The **PersistentModel** class holds a **Client** object. **ClientTA** gets the **Client** object by calling the **getModelObject** method, which gets the value stored in the **myModelObject** property and is populated in the inherited **populateFromObject** method.

ModelTA Class Diagrams

The following diagram shows the properties and methods defined in the **ModelTA** class.

ModelTA
+ allErrors : StringArray + allWarnings : StringArray + expectedEdition : Integer + focusField : String + lockedByProcessId : Decimal + lockedTimeStamp : TimeStamp + modificationCode : Integer + myModelObject : PersistentModel
# addError(pError : String; pFocusField : String) # addWarning(pWarning : String; pFocusField : String) # checkEdition(pExpectedEdition : Integer) : Boolean # clearErrors( ) # clearErrorsOnSubordinateTAs( ) # clearWarnings( ) # clearWarningsOnSubordinateTAs( ) # copyErrors(pFromTA : ModelTA) # copyWarnings(pFromTA : ModelTA) # createEntity( ) : Boolean # createEntityInTransState( ) : Boolean # createEntityWithTransactionImplementor(pTransactionImplementor : TransactionImplementor) : Boolean # createSubordinateObjects( ) : Boolean # deleteEntity( ) : Boolean # deleteEntityInTransState( ) : Boolean # deleteEntityWithTransactionImplementor(pTransactionImplementor : TransactionImplementor) : Boolean # deleteSubordinateObjects( ) : Boolean # doAbortTransactionCleanup( ) # doAbortTransactionCleanupForSubordinateObjects( ) # doCreate( ) : Boolean # doDelete( ) : Boolean # doModify(pModification : Integer) : Boolean + doPreValidate( ) # doUpdate( ) : Boolean + doValidate(pValidationType : Integer) : Boolean + getFullErrorDetails( ) : String + getModelObject( ) : PersistentModel # getModelObjectClass( ) : Class # hasErrors( ) : Boolean # hasNoErrors( ) : Boolean # hasOnlySubordinatePersistentObjects( ) : Boolean + initialize( ) # lockForCreate( ) : Boolean # lockForDelete( ) : Boolean # lockForModify( ) : Boolean # lockForUpdate( ) : Boolean # modifyEntity(pModification : Integer) : Boolean # modifyEntityInTransState(pModification : Integer) : Boolean # modifyEntityWithTransactionImplementor(pTransactionImplementor : TransactionImplementor; pModification : Integer) : Boolean # modifySubordinateObjects(pModification : Integer) : Boolean + persistEntity(pTransactionType : Integer) : Boolean + persistEntityInTransState(pTransactionType : Integer) : Boolean + populateFromObject(pModelObject : PersistentModel) # populateSubordinateObjects(pModelObject : PersistentModel) # tryLockingObject(pObject : Object) : Boolean # updateEntity( ) : Boolean # updateEntityInTransState( ) : Boolean # updateEntityWithTransactionImplementor(pTransacitonImplementor : TransactionImplementor) : Boolean # updateSubordinateObjects( ) : Boolean

The **ClientTA** class is a subclass of the **ModelTA** class and therefore inherits the properties and methods of the **ModelTA** class, with **(r)** denoting a reimplemented class.

ClientTA
+ myAddressTA : AddressTA + name : String
# clearErrorsOnSubordinateTAs( ) # clearWarningsOnSubordinateTAs( ) # createSubordinateObjects( ): Boolean (r) # doAbortTransactionCleanupForSubordinateObjects( ) + doValidate(pValidationType : Integer) : Boolean (r) + getModelObject( ) : Client (r) # getModelObjectClass( ) : Class (r) + initialize( ) updating (r) # lockForCreate( ) : Boolean (r) # lockForDelete( ) : Boolean (r) # lockForModify( ) : Boolean (r) # lockForUpdate( ) : Boolean (r) # modifySubordinateObjects (pModification : Integer) : Boolean # populateFromObject(pClient : Client) : (r) # populateSubordinateObjects (pClient : Client) # updateSubordinateObjects ( ) : Boolean

Note The **ClientTA** class contains the same properties as the **PersistentModel** class.

The **ClientTA** class reimplements various methods of the **ModelTA** class as required. However, some methods are commonly implemented on all transaction agents and these are listed in the following table. In the first column of this table, ***** denotes a required implementation and ****** denotes a required implementation unless the **hasOnlySubordinatePersistentObjects** method returns **true**.

Method	Description
doValidate	Validates the **ClientTA** properties
getModelObject **	Retrieves the **Client** object stored in the **myModelObject** property
getModelObjectClass **	Retrieves the **Client** class type
initialize	Clears the **ClientTA** properties to an uninitialized state
lockForCreate *	Locks objects during a create operation

Method	Description
lockForDelete *	Locks objects during a delete operation
lockForModify *	Locks objects during a modify operation
lockForUpdate *	Locks objects during an update operation
populateFromObject **	Copies **Client** model properties to the **ClientTA** properties

ModelTA Code Implementation Examples

The following code examples show the implemented methods in the **ClientTA** class.

clearErrorsOnSubordinateTAs

This method clears the errors on the **myAddressTA** property and is reimplemented only if the transaction agent contains subordinate objects.

```
clearErrorsOnSubordinateTAs() updating, protected;

begin
    inheritMethod();

    if self.myAddressTA <> null then
        self.myAddressTA.clearErrors();
    endif;
end;
```

clearWarningsOnSubordinateTAs

This method clears the warnings on the **myAddressTA** property and is reimplemented only if the transaction agent contains subordinate objects.

```
clearWarningsOnSubordinateTAs() updating, protected;

begin
    inheritMethod();

    if self.myAddressTA <> null then
        self.myAddressTA.clearWarnings();
    endif;
end;
```

createSubordinateObjects

This method creates an **Address** object for a **Client** and is reimplemented only if the transaction agent contains subordinate objects.

The system should already be in a transaction state and therefore the subordinate object should be persisted in **TransState**.

```
createSubordinateObjects() : Boolean updating, protected;

begin
    inheritMethod();

    // Set the my model reference on the subordinate TA
    self.myAddressTA.myModel := self.getModelObject();

    // Create the Address object
    if not self.myAddressTA.persistEntityInTransState( TransactionType_Persist ) then
        // Error occured, copy AddressTA errors to self.allErrors attribute
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors();
end;
```

doAbortTransactionCleanupForSubordinateObjects

This method performs clean up logic on the **AddressTA** if a transaction is aborted by calling the **doAbortTransactionCleanup** method in the **ModelTA** class.

```
doAbortTransactionCleanupForSubordinateObjects() protected;

begin
    inheritMethod();

    if self.myAddressTA <> null then
        self.myAddressTA.doAbortTransactionCleanup();
    endif;
end;
```

Note An abort operation is usually the result of a failed transaction.

doValidate

This method validates the **ClientTA** and subordinate TAs and returns **true** if no errors exist.

A failed validation check should add an error to the **allErrors** collection to result in the method returning **false**.

```
doValidate( pValidationType : Integer ) : Boolean updating;

begin
    inheritMethod( pValidationType );

    // if delete operation, then no validation required
    if pValidationType = ValidationType_Delete then
        return true;
    endif;

    self.name :=self.name.trimWhiteSpace();

    // validate name
    if self.name = null then
        // name cannot be null
        self.addError( "Name is a required field", Focus_Name );
    endif;

    // validate client does not exist (validates on create only)
    if pValidationType = ValidationType_Create
    and app.myCompany.hasClientWithThisName( self.name ) then
        self.addError( "Client already exists", Focus_Name);
    endif;

    // validate another client does not have the same name (validates on update only)
    if pValidationType = ValidationType_Update
    and app.myCompany.hasAnotherClientWithThisName( self.name, self.getModelObject() ) then
        self.addError( "Another client with this name already exists", Focus_Name);
    endif;

    // validate address
    if self.myAddressTA <> null and not self.myAddressTA.doValidate( pValidationType ) then
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors();
end;
```

getModelObject

This method returns the object stored in the **myModelObject** property and casts it to a **Client** type.

```
getModelObject() : Client;

begin
    return inheritMethod().Client;
end;
```

getModelObjectClass

This method returns the **Class** type used by the transaction agent. In this case, it is a **Client** class type.

```
getModelObjectClass() : Class protected;

begin
    return Client;
end;
```

This is method is used in the **doCreate** method to create a persistent object, as shown in the following code fragment.

```
create self.myModelObject as self.getModelObjectClass() persistent;
```

initialize

This method is called when the transaction agent is initialized and it sets the transaction agent properties to an uninitialized state (which is generally null).

```
initialize() updating;

begin
    inheritMethod();

    self.name := null;
end;
```

lockForCreate

This method is used to manually place an exclusive lock on objects during a create operation.

The **app.myCompany.allClients** collection is being locked because the **Client** object is an inverse of the **allClients** collection.

```
lockForCreate() : Boolean protected;

begin
    if not self.tryLockingObject( app.myCompany.allClients ) then
        return false;
    endif;

    return true;
end;
```

lockForDelete

This method is used to manually place an exclusive lock on objects during a delete operation.

The **app.myCompany.allClients** collection is being locked because the **Client** object is an inverse of the **allClients** collection.

```
lockForDelete() : Boolean protected;

begin
    if not self.tryLockingObject( app.myCompany.allClients ) then
        return false;
    endif;

    return true;
end;
```

lockForModify

This method is used to manually place an exclusive lock on objects during a modify operation. A modify is a targeted update that sets only some of the properties on an object.

This method returns **true** because there is no modification code being called in the **ClientTA** class.

```
lockForModify() : Boolean protected;

begin
    return true;
end;
```

lockForUpdate

This method is used to manually place an exclusive lock on objects during an update operation.

The **app.myCompany.allClients** collection is being locked because the **Client** object is an inverse of the collection. The **name** attribute in the **Client** model is used as a dictionary key, therefore we need only to lock the **allClients** collection if the value of the **name** property has changed.

```
lockForUpdate() : Boolean protected;

begin
    // Is the key value on the object changing?
    // If it is, then we need to lock the collection
    if self.name <> self.getModelObject().name then
        if not self.tryLockingObject( app.myCompany.allClients ) then
            return false;
        endif;
    endif;

    return true;
end;
```

We attempt to lock during an update if the property being updated is used as a key.

modifySubordinateObjects

This method is used when the **ClientTA** modifies an object to modify the **Address** object.

A modify operation is a targeted update that sets only some of the properties on an object.

The system should already be in a transaction state, therefore we will persist the subordinate object in **TransState**.

```
modifySubordinateObjects( pModification : Integer ): Boolean updating, protected;

begin
    inheritMethod( pModification );

    // Modify the Address object
    if self.myAddressTA <> null
    and not self.myAddressTA.persistEntityInTransState( pModification ) then
        // Error occured, Copy AddressTA errors to self.allErrors attribute
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors();
end;
```

populateFromObject

This method copies the properties of a **Client** model into the relevant **ClientTA** properties.

```
populateFromObject( pClient : Client ) updating;

begin
    inheritMethod( pClient );

    self.name := pClient.name;
end;
```

populateSubordinateObjects

This method copies the properties of the subordinate objects into the respective subordinate transaction agents and sets the subordinate reference to **self**.

```
populateSubordinateObjects( pClient : Client ) updating, protected;

vars
    addressTA : AddressTA;

begin
    inheritMethod( pClient );

    create addressTA transient;

    addressTA.populateFromObject( pClient.myAddress );
    addressTA.myModelTA := self;
end;
```

updateSubordinateObjects

This method updates the properties of the subordinate objects (**Address**) in the **Client** model.

The system should already be in a transaction state and therefore the subordinate objects should be persisted in **TransState**.

```
updateSubordinateObjects() : Boolean updating, protected;

begin
    inheritMethod();

    if self.myAddressTA <> null
    and not self.myAddressTA.persistEntityInTransState( TransactionType_Persist ) then
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors();
end;
```

ModelTA Properties

The read-only properties defined in the **ModelTA** class are summarized in the following table.

Property	Value	Stores...
allErrors	HugeStringArray	A collection of errors (including validation errors) encountered during a persist operation. Errors will prevent the transaction agent from persisting a transaction.
allWarnings	HugeStringArray	A collection of all the warnings during a persist operation. Warnings will not prevent the transaction agent from persisting a transaction.
expectedEdition	Integer	The value of the expected object edition.
focusField	String	A global constant value for the field that receives focus if an error occurs.
lockedByProcessId	Decimal	The process identifier that has acquired a lock on the transaction agent.
lockedByTimeStamp	TimeStamp	The sever time when the transaction agent was locked.
modificationCode	Integer	A global constant value of the specific properrties to update during a modify operation. A modify operation is a targeted update that sets only some of the properties on an object.
myModelObject	PersistentModel	The persistent object of type **PersistentModel** for which the transaction agent is responsible.

ModelTA Methods

The methods defined in the **ModelTA** class are summarized in the following table.

Method	Description
addError	Called in the **doValidate** method to add an error to the **allErrors** collection and assigns the field to receive focus
addWarning	Adds a warning to the **allWarnings** collection and assigns the field to receive focus
checkEdition	Checks if the current edition of the class is the expected version
clearErrors	Called at the start of every persist operation to clear the **allErrors** collection
clearErrorsOnSubordinateTAs	Empty method to be reimplemented on subclasses that use subordinate objects to clear all errors on the subordinate transaction agents
clearWarnings	Called at the start of every persist operation to clear the **allWarnings** collection
clearWarningsOnSubordinateTAs	Clears the **allWarnings** collection on subordinate transaction agents
copyErrors	Copies the errors from the value specified in the **ModelTA** parameter into the calling transaction agents **allErrors** collection
copyWarnings	Copies the warnings from the value specified in the **ModelTA** parameter into the calling transaction agents **allWarnings** collection
createEntity	Programmatically called to persist an object when the system is not in a transaction state
createEntityInTransState	Programmatically called to persist an object when the system is already in a transaction state
createEntityWithTransactionImplementor	Programmatically called when creating an object
createSubordinateObjects	Called to persist subordinate objects when the system is in a transaction state
deleteEntity	Programmatically called to delete an object when the system is not in a transaction state
deleteEntityInTransState	Programmatically called to delete an object when the system is already in a transaction state
deleteEntityWithTransactionImplementor	Programmatically called when deleting an object
deleteSubordinateObjects	Programmatically called to delete subordinate objects when the system is in a transaction state
doAbortTransactionCleanup	Programmatically called to perform cleanup behavior if a transaction fails
doAbortTransactionCleanupForSubordinateObjects	Performs cleanup behavior for subordinate objects if a transaction fails

Method	Description
doCreate	Programmatically called to persist an object to the database
doDelete	Programmatically called to delete an object from the database
doModify	Programmatically called to modify a targeted property or subset of properties
doPreValidate	Empty method called before the **doValidate** method to perform pre-validation logic
doUpdate	Programmatically called to update a persisted object and its subordinates
doValidate	Validates the transaction agent properties prior to persisting an object
getFullErrorDetails	Returns a formatted string of all the errors in the **allErrors** collection separated by a carriage return/ line feed characters
getModelObject	Returns the object stored in the **myModelObject** property
getModelObjectClass	Returns the class type used by the transaction agent
hasErrors	Checks the **allErrors** collection to see if it contains errors
hasNoErrors	Checks the **allErrors** collection to see if it contains no errors
hasOnlySubordinatePersistentObjects	Declares the transaction agent to work with various subordinate objects and not a single persistent object or type
initialize	Initializes the transaction agent properties to an uninitialized state
lockForCreate	Manually places an exclusive lock on objects during a create operation
lockForDelete	Manually places an exclusive lock on objects during a delete operation
lockForModify	Manually places an exclusive lock on objects during a modify operation
lockForUpdate	Manually places an exclusive lock on objects during an update operation
modifyEntity	Programmatically called to modify a targeted property when the system is not in transaction state
modifyEntityInTransState	Programmatically called to modify a targeted property when the system is in transaction state
modifyEntityWithTransactionImplementor	Programmatically called to modify targeted properties
modifySubordinateObjects	Programmatically called to modify targeted properties on subordinate objects

Method	Description
persistEntity	Main method called to persist (create, update, modify, or delete) an object to the database when the system is not in a transaction state
persistEntityInTransState	Main method called to persist (create, update, modify, or delete) an object to the database when the system is in a transaction state
populateFromObject	Copies the **PersistentModel** object properties into the properties of the transaction agent
populateSubordinateObjects	Copies the **PersistentModel** object properties for the subordinate objects to the respective subordinate transaction agents
tryLockingObject	Attempts to place an exclusive lock on the object passed in as the value of the **pObject** parameter
updateEntity	Programmatically called to update an object when the system is not in transaction state
updateEntityInTransState	Programmatically called to update an object when the system is in a transaction state
updateEntityWithTransactionImplementor	Programmatically called to update an object by passing a **TransactionImplementor** as an argument to determine if the transaction should be committed to the database
updateSubordinateObjects	Programmatically called to update the subordinate objects

addError

The protected **addError** method in the **ModelTA** class is generally called in the **doValidate** method to add an error to the **allErrors** collection and assigns the field to receive focus.

Parameters

The parameters for this method are listed in the following table.

Name	Type	Description
pError	String	Description of the error.
pFocusField	String	Field to receive focus. This value should be stored as a global constant.

Base Implementation

```
addError( pError        : String;
          pFocusField   : String
          ) updating, protected, final;

begin
    self.allErrors.add( pError );

    // Focus should be set to the first field in error
    if self.focusField = null then
        self.focusField := pFocusField;
    endif;
end;
```

addWarning

The protected **addWarning** method in the **ModelTA** class adds a warning to the **allWarnings** collection and assigns the field to receive focus.

Parameters

The parameters for this method are listed in the following table.

Name	Type	Description
pWarning	String	Description of the warning.
pFocusField	String	The field to receive focus. This value should be stored as a global constant.

Base Implementation

```
addWarning( pWarning : String;
            pFocusField : String
            ) updating, protected, final;

begin
    self.allWarnings.add( pWarning );

    if self.focusField <> null then
        self.focusField := pFocusField;
    endif;
end;
```

checkEdition

The protected **checkEdition** method in the **ModelTA** class returns a Boolean value. This method checks if the current edition of the class is the expected version. This method should be reimplemented on subclasses where edition checking is required.

Reimplementations of this method should return **true** if the edition is the expected edition; otherwise it returns **false**.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pExpectedEdition	Integer	Number of the expected edition

Base Implementation

```
checkEdition( pExpectedEdition : Integer ) : Boolean protected;

begin
    return true;
end;
```

clearErrors

The protected **clearErrors** method in the **ModelTA** class is called at the start of every persist operation to clear the **allErrors** collection.

Base Implementation

```
clearErrors() protected, updating, final;

begin
    self.allErrors.clear();
    self.focusField := null;
    self.clearErrorsOnSubordinateTAs();
end;
```

clearErrorsOnSubordinateTAs

The protected **clearErrorsOnSubordinateTAs** method in the **ModelTA** class is an empty method to be reimplemented on subclasses that use subordinate objects, to clear all errors on the subordinate transaction agents (TAs).

The reimplemented method should call the **clearErrors** method on the subordinate transaction agent (TA), as shown in the following example.

```
clearErrorsOnSubordinateTAs() updating, protected;

begin
    inheritMethod();

    if self.myAddressTA <> null then
        self.myAddressTA.clearErrors();
    endif;
end;
```

Base Implementation

```
clearErrorsOnSubordinateTAs() updating, protected;

begin

end;
```

clearWarnings

The protected **clearWarnings** method in the **ModelTA** class is called at the start of every persist operation to clear the **allWarnings** collection.

Base Implementation

```
clearWarnings() updating, protected, final;

begin
    self.allWarnings.clear();
    self.focusField := null;
    self.clearWarningsOnSubordinateTAs();
end;
```

clearWarningsOnSubordinateTAs

The protected **clearWarningsOnSubordinateTAs** method in the **ModelTA** class clears all warnings on the subordinate transaction agents (TAs).

Reimplement this method if the transaction agent contains subordinates and should call the **clearWarnings** method in the subordinate transaction agent (TA), as shown in the following example.

```
clearWarningsOnSubordinateTAs() updating, protected;

begin
    inheritMethod();

    if self.myAddressTA <> null then
        self.myAddressTA.clearWarnings();
    endif;
end;
```

Base Implementation

```
clearWarningsOnSubordinateTAs() updating, protected;

begin

end;
```

copyErrors

The protected **copyErrors** method in the **ModelTA** class copies the errors from the **pFromTA** parameter into the calling transaction agents **allErrors** collection. This method is commonly used in transaction agents with subordinate objects to pass the subordinate transaction agent errors to the primary transaction agent.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pFromTA	ModelTA	Transaction agent from which to copy the errors

Base Implementation

```
copyErrors( pFromTA : ModelTA ) updating, protected, final;

vars
    errorMessage : String;

begin
    foreach errorMessage in pFromTA.allErrors do
        self.addError( errorMessage, null );
    endforeach;

    if self.focusField = null then
        self.focusField := pFromTA.focusField;
    endif;
end;
```

copyWarnings

The protected **copyWarnings** method in the **ModelTA** class copies the warnings from the **pFromTA** parameter into the calling transaction agents **allWarnings** collection. This method is commonly used in transaction agents with subordinate objects to pass the subordinate transaction agent warning to the primary transaction agent.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pFromTA	ModelTA	Transaction agent from which to copy the warnings

Base Implementation

```
copyWarnings( pFromTA : ModelTA ) updating, protected, final;

vars
    warningMessage : String;

begin
    foreach warningMessage in pFromTA.allWarnings do
        self.addWarning( warningMessage, null );
    endforeach;

    if self.focusField = null then
        self.focusField := pFromTA.focusField;
    endif;
end;
```

createEntity

The protected **createEntity** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to persist an object when the system is not in a transaction state.

This method returns **true** if the entity was created successfully; otherwise it returns **false** if the entity failed to be created.

Base Implementation

```
createEntity() : Boolean updating, final, protected;

vars
    transactionImplementor  : CommitTransactionTI ;

begin
    create transactionImplementor transient;

    return self.createEntityWithTransactionImplementor( transactionImplementor );

epilog
    delete transactionImplementor;
end;
```

createEntityInTransState

The protected **createEntityInTransState** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to persist an object when the system is already in a transaction state.

This method is generally used for persisting a collection of objects or subordinate objects.

This method returns **true** if the entity was created successfully; otherwise it returns **false** if the entity failed to be created.

Base Implementation

```
createEntityInTransState() : Boolean updating, final, protected;

vars
    transactionImplementor  : NoTransactionTI ;

begin
    create transactionImplementor transient;

    return self.createEntityWithTransactionImplementor( transactionImplementor );

epilog
    delete transactionImplementor;
end;
```

createEntityWithTransactionImplementor

The protected **createEntityWithTransactionImplementor** method in the **ModelTA** class returns a Boolean value. This method is programmatically called when creating an object.

A **TransactionImplementor** is passed as a parameter to determine if the transaction should be committed to the database. Generally, the transaction implementor is a **CommitTransactionTI** for general persists or a **NoTransactionTI** for persisting subordinate objects or collections.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTransactionImplementor	TransactionImplementor	The transaction implementor passed in determines if transaction state changes when the **TransactionImplementor** class **doBeginTransaction**, **doCommitTranasction**, or **doAbortTransaction** method is called

This method returns **true** if the entity was created successfully; otherwise it returns **false** if the entity failed to be created.

Base Implementation

```
createEntityWithTransactionImplementor(
        pTransactionImplementor : TransactionImplementor
        ) : Boolean updating, protected, final;

vars
    validateRtn             : Boolean;
    transactionCompleted    : Boolean;

begin
    self.clearErrors();
    self.clearWarnings();

    validateRtn := self.doValidate( ValidationType_Create );

    if validateRtn then
        if self.lockForCreate() then
            pTransactionImplementor.doBeginTransaction();
            if self.doCreate() then
                pTransactionImplementor.doCommitTransaction();
            else
                pTransactionImplementor.doAbortTransaction();
                return false;
            endif;
        else
            pTransactionImplementor.doAbortTransaction();
            self.addError( "Create conflicted with another user's activity. Try again shortly.", null );
            return false;
        endif;
    else
        return false;
    endif;

    transactionCompleted    := true;

    return true;

epilog
    if not transactionCompleted then
        self.doAbortTransactionCleanup();
    endif;
end;
```

createSubordinateObjects

The protected **createSubordinateObjects** method in the **ModelTA** class returns a Boolean value. This method is called to persist subordinate objects when the system is in a transaction state.

This method, shown in the following example, should be reimplemented by transaction agents that contain subordinate objects and should persist objects using the **persistEntityInTransState** method, as the system should already be in a transaction state.

```
createSubordinateObjects() : Boolean updating, protected;

begin
    inheritMethod();

    // Set the my model reference on the subordinate TA
    self.myAddressTA.myModel := self.getModelObject();

    // Create the Address object
    if not self.myAddressTA.persistentEntityInTransState( TransactionType_Persist ) then
        // Error occurred, copy AddressTA errors to self.allErrors collection
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors;
end;
```

This method returns **true** if the subordinate objects were created successfully; otherwise it returns **false** if the subordinate objects failed to be created.

Base Implementation

```
createSubordinateObjects() : Boolean updating, protected;

begin
    return true;
end;
```

deleteEntity

The protected **deleteEntity** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to delete an object when the system is not in a transaction state.

This method returns **true** if the entity was deleted successfully; otherwise it returns **false** if the entity failed to be deleted.

Base Implementation

```
deleteEntity() : Boolean updating, final, protected;

vars
    transactionImplementor   : CommitTransactionTI;

begin
    create transactionImplementor transient;

    return self.deleteEntityWithTransactionImplementor( transactionImplementor );

epilog
    delete transactionImplementor;
end;
```

deleteEntityInTransState

The protected **deleteEntityInTransState** method in the **ModelTA** class returns a Boolean value. This method is generally used for deleting a collection of objects or subordinate objects.

This method returns **true** if the entity was deleted successfully; otherwise it returns **false** if the entity failed to be deleted.

Base Implementation

```
deleteEntityInTransState() : Boolean updating, final, protected;

vars
    transactionImplementor   : NoTransactionTI;

begin
    create transactionImplementor transient;

    return self.deleteEntityWithTransactionImplementor( transactionImplementor );

epilog
    delete transactionImplementor;
end;
```

deleteEntityWithTransactionImplementor

The protected **deleteEntityWithTransactionImplementor** method in the **ModelTA** class returns a Boolean value. This method is programmatically called when deleting an object.

A **TransactionImplementor** is passed as a parameter to determine if the transaction should be committed to the database. Generally, the transaction implementor is a **CommitTransactionTI** for general persists or a **NoTransactionTI** for persisting subordinate objects or collections.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTransactionImplementor	TransactionImplementor	The transaction implementor passed in determines if transaction state changes when the **TransactionImplementor** class **doBeginTransaction**, **doCommitTranasction**, or **doAbortTransaction** method is called

This method returns **true** if the entity was deleted successfully; otherwise it returns **false** if the entity failed to be deleted.

Base Implementation

```
deleteEntityWithTransactionImplementor(
        pTransactionImplementor : TransactionImplementor
        ) : Boolean updating, final, protected;

vars
    transactionCompleted    : Boolean;

begin
    self.clearErrors();
    self.clearWarnings();

    if self.doValidate( ValidationType_Delete ) then
        if self.lockForDelete() then
            pTransactionImplementor.doBeginTransaction();
            if self.doDelete() then
                pTransactionImplementor.doCommitTransaction();
            else
                pTransactionImplementor.doAbortTransaction();
                return false;
            endif;
        else
            pTransactionImplementor.doAbortTransaction();
            self.addError(
                "Delete conflicted with another user's activity. Try again shortly.",
                null
            );
            return false;
        endif;
    else
        return false;
    endif;

    transactionCompleted    := true;

    return true;

epilog
    if not transactionCompleted then
        self.doAbortTransactionCleanup();
    endif;
end;
```

deleteSubordinateObjects

The protected **deleteSubordinateObjects** method in the **ModelTA** class returns a Boolean value. This method is called to delete subordinate objects when the system is in a transaction state.

By default, this method returns **true** but it should be reimplemented by transaction agents that contain subordinate objects.

Persistence should be performed using the **persistEntityInTransState** method, as the system should already be in a transaction state.

This method returns **true** if the subordinate objects were deleted successfully; otherwise it returns **false** if the subordinate objects failed to be deleted.

Base Implementation

```
deleteSubordinateObjects() : Boolean updating, protected;

begin
    return true;
end;
```

doAbortTransactionCleanup

The protected **doAbortTransactionCleanup** method in the **ModelTA** class is programmatically called to perform cleanup behavior if a transaction fails.

This method should also be called from the **doAbortTransactionCleanupForSubordinateObjects** method for each transaction agent that has subordinate transaction agents, as shown in the following example.

```
doAbortTransactionCleanupForSubordinateObjects() protected;

begin
    inheritMethod();

    if self.myAddressTA <> null then
        self.myAddressTA.doAbortTransactionCleanup();
    endif;
end;
```

Base Implementation

```
doAbortTransactionCleanup() updating, protected;

begin
    if not app.isValidObject( self.myModelObject ) then
        self.myModelObject  := null;
    endif;

end;
```

doAbortTransactionCleanupForSubordinateObjects

The protected **doAbortTransactionCleanupForSubordinateObjects** method in the **ModelTA** class performs cleanup behavior for subordinate objects if a transaction fails.

Note All classes that contain subordinate objects should reimplement this method.

Base Implementation

There is no implementation at this level, because not all classes contain subordinate objects.

```
doAbortTransactionCleanupForSubordinateObjects() protected;

begin

end;
```

doCreate

The protected **doCreate** method in the **ModelTA** class returns a Boolean value. It is programmatically called to persist an object to the database.

This method also calls the **createSubordinateObjects** method responsible for persisting subordinate objects.

This method returns **true** if the object and subordinates were persisted successfully; otherwise it returns **false** if the object or subordinates failed to be persisted.

Base Implementation

```
doCreate() : Boolean updating, protected, final;

begin
    if not self.hasOnlySubordinatePersistentObjects() then
        create self.myModelObject as self.getModelObjectClass() persistent;

        self.myModelObject.onCreate( self );
    endif;

    if not self.createSubordinateObjects() then
        return false;
    endif;

    return true;
end;
```

doDelete

The protected **doDelete** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to delete an object from the database.

This method also calls the **deleteSubordinateObjects** method responsible for deleting subordinate objects.

This method returns **true** if the object and subordinates were deleted successfully; otherwise it returns **false** if the object or subordinates failed to be deleted.

Base Implementation

```
doDelete() : Boolean updating, protected, final;

begin
    if not self.hasOnlySubordinatePersistentObjects() then
        self.myModelObject.onDelete( self );
    endif;

    if not self.deleteSubordinateObjects() then
        return false;
    endif;

    return true;
end;
```

doModify

The protected **doModify** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to modify a targeted property or subset of properties.

Note Modify is a targeted update that sets only some of the properties on an object. The targeted property or subset of properties depends on the modification code passed into the method.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pModification	Integer	The specific property or subset of properties to be modified. This value should be stored as a global constant.

This method returns **true** if the entity was updated successfully; otherwise it returns **false** if the entity failed to be updated.

Base Implementation

```
doModify( pModification : Integer ) : Boolean updating, protected, final;

begin
    if not self.hasOnlySubordinatePersistentObjects() then
        self.myModelObject.onModify( self );
    endif;

    if not self.modifySubordinateObjects( pModification ) then
        return false;
    endif;

    return true;
end;
```

doPreValidate

The public **doPreValidate** method in the **ModelTA** class is an empty method called before the **doValidate** method to perform pre-validation logic. Examples of pre-validation logic include:

- Sanitizing properties by removing whitespace
- Applying formatting to properties such as uppercase, phone formats, and so on
- Generating unique identifiers such as Globally Unique Identifiers (GUIDs) or incrementing identifiers (IDs)

Note Reimplement this method on all classes that require pre-validation logic.

The following is an example of the reimplementation of this method.

```
doPreValidate() updating;

begin
    inheritMethod();

    // Generate Item Code
    if self.codeNumber = null then
        self.codeNumber := self.getNextCodeNumber();
    endif;

    // Initialize Data
    if self.listedDate = null
    or not self.listedDate.isValid() then
        self.listedDate := Utilities@getServerDate();
    endif;

    // Sanitize Data
    self.fullDescription := self.fullDescription.trimWhiteSpace();
    self.shortDescription := self.shortDescription.trimWhiteSpace();
end;
```

This reimplementation generates a code number, and provides a default date value, as well as sanitizing the data by trimming whitespace.

Base Implementation

```
doPreValidate() updating;

begin

end;
```

doUpdate

The protected **doUpdate** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to update a persisted object and its subordinates.

This method returns **true** if the object and subordinates were updated successfully; otherwise it returns **false** if the object or subordinates failed to be updated.

Base Implementation

```
doUpdate() : Boolean updating, protected, final;

begin
    if not self.hasOnlySubordinatePersistentObjects() then
        self.myModelObject.onUpdate( self );
    endif;

    if not self.updateSubordinateObjects() then
        return false;
    endif;

    return true;
end;
```

doValidate

The public **doValidate** method in the **ModelTA** class returns a Boolean value. This method validates the transaction agent properties before persisting an object.

You should generally reimplement this method on most child classes. Failed validations should add an error to the **allErrors** collection by calling the **ModelTA** class **addError** method.

The following is an example of reimplementing this method.

```
doValidate( pValidationType : Integer ) : Boolean updating;

begin
    inheritMethod( pValidationType );

    // If delete operation then no validation required
    if pValidationType = ValidationType_Delete then
        return true;
    endif;

    self.name := self.name.trimWhiteSpace();

    // validate name attribute
    if self.name = null then
        self.addError( "Name is a required field", Focus_Name );
    endif;

    // validate subordinate
    if self.myAddressTA <> null
    and not self.myAddress.doValidate( pValidationType ) then
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors();
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pValidationType	Integer	The type of validation to be performed. This value should be stored as a global constant.

This method returns **true** if the **allErrors** collection does not contain errors; otherwise it returns **false** if the **allErrors** collection contains errors.

Base Implementation

```
doValidate( pValidationType : Integer ) : Boolean updating;

begin
    return self.hasNoErrors();
end;
```

getFullErrorDetails

The public **getFullErrorDetails** method in the **ModelTA** class returns a formatted string of all the errors in the **allErrors** collection separated by a carriage return / line feed characters.

This method is generally used to populate message boxes or labels with a list of the errors.

Base Implementation

```
getFullErrorDetails() : String;

vars
    errorMessage : String;
    fullErrorDetails : String;

begin
    foreach errorMessage in self.allErrors do
        fullErrorDetails.appendTextWithDelim( CrLf, errorMessage, false );
    endforeach;

    return fullErrorDetails;
end;
```

getModelObject

The public **getModelObject** method in the **ModelTA** class returns the **PersistentModel** object stored in the **ModelTA** class **myModelObject** property.

This method is called when persisting an object to determine if the object should be created or updated. If this method returns null, the object should be created; otherwise the object already exists so it should be updated.

Note This method is generally reimplemented on all transaction agents.

When reimplementing this method, the return type should be type-specific and returned with a type cast of the **inheritMethod** instruction call, as shown in the following example.

```
getModelObject() : Client;

begin
    return inheritMethod().Client;
end;
```

Base Implementation

```
getModelObject() : PersistentModel;

begin
    return self.myModelObject;
end;
```

getModelObjectClass

The protected **getModelObjectClass** method in the **ModelTA** class returns the **Class** type used by the transaction agent, is used when creating a persistent object to determine its class type; for example:

```
create self.myModelObject as self.getModelObjectClass() persistent;
```

The following is an example of the method used to create a persistent object to determine its class type.

```
getModelObjectClass() : Class protected;

begin
    return Client;
end;
```

Base Implementation

```
getModelObjectClass() : Class protected;

begin
    return PersistentModel;
end;
```

hasErrors

The protected **hasErrors** method in the **ModelTA** class returns a Boolean value. This method checks the **allErrors** collection to see if it contains errors.

This method returns **true** if the transaction agent has errors; otherwise it returns **false** if the transaction agent does not have errors.

Base Implementation

```
hasErrors() : Boolean protected, final;

begin
    return not self.hasNoErrors();
end;
```

hasNoErrors

The protected **hasNoErrors** method in the **ModelTA** class returns a Boolean value. This method checks the **allErrors** collection to see if it contains no errors.

This method returns **true** if the transaction agent has no errors; otherwise it returns **false** if the transaction agent has errors.

Base Implementation

```
hasNoErrors() : Boolean protected, final;

begin
    return self.allErrors.isEmpty();
end;
```

hasOnlySubordinatePersistentObjects

The protected **hasOnlySubordinatePersistentObjects** method in the **ModelTA** class returns a Boolean value. This method is used to declare the transaction agent to work with various subordinate objects and not a single persistent object or type. For example, a **ShoppingCartTA** might contain two collections of different types such as **RetailSales** and **Tenders**. These items should be displayed to the user in a single list and checked out in the same transaction.

As these objects should be persisted together and are subordinate objects, the persistence should be performed in the **ModelTA** class **createSubordinateObjects** method in transaction state, as shown in the following example.

```
createSubordinateObjects() : Boolean updating, protected;

vars
    retailSaleTA : RetailSaleTA;
    tenderTA :TenderTA;
    currentTimestamp : TimeStamp;

begin
    inheritMethod();

    // Persist Retail Sales
    foreach retailSaleTA in self.allRetailSaleTAs do
        retailSaleTA.timeStamp := currentTimestamp;
        if not retailSaleTA.persistEntityInTransState( TransactionType_Persist ) then
            self.copyErrors( retailSaleTA );
        endif;
    endforeach;

    // Persist Tenders
    foreach tenderTA in self.allTenderTAs do
        tenderTA.timeStamp := currentTimestamp;
        if not tenderTA.persistEntityInTransState( TransactionType_Persist ) then
            self.copyErrors( tenderTA );
        endif;
    endforeach;

    return self.hasNoErrors;
end;
```

Subordinate objects are persisted using the **persistEntity** method, as follows.

```
if not app.getShoppingCartTA().persistEntity( TransactionType_Persist ) then
    // Display error message / handle error here
    return;
endif;
```

This method needs to be reimplemented only on transaction agents that do not use a specific object type or persist multiple objects. To reimplement this method, simply return **true**, as follows.

```
hasOnlySubordinatePersistentObjects() : Boolean protected;

begin
    return true;
end;
```

Base Implementation

```
hasOnlySubordinatePersistentObjects() : Boolean protected;

begin
    return false;
end;
```

initialize

The public **initialize** method in the **ModelTA** class initializes the transaction agent properties to an uninitialized state (generally null).

This method is generally reimplemented on all transaction agents, as shown in the following example.

```
initialize() updating;

begin
    inheritMethod();

    self.name  := null;
end;
```

It is generally not required to manually call this method. However, one situation in which you may need to call the **initialize** method directly is when performing persistent operations in a loop using the same transaction agent. The transaction agent needs to be initialized at the start of each loop, to ensure no properties exist from the previous iteration.

The following example shows the **initialize** method being called in a client data loader method.

```
create clientTA transient;
create addressTA transient;

while not inputFile.endOfFile() do
    line := inputFile.readLine.trimBlanks();

    if line <> "" then
        pos := FirstIndexInLine;

        clientTA.initialize();
        clientTA.name := self.getNextToken( line, pos );

        addressTA.initialize();
        addressTA.myModelTA := clientTA;

        addressTA.street := self.getNextToken( line, pos );
        addressTA.city := self.getNextToken( line, pos );
        addressTA.country := self.getNextToken( line, pos );
        addressTA.email := self.getNextToken( line, pos );
        addressTA.fax := self.getNextToken( line, pos );
        addressTA.phone := self.getNextToken( line, pos );
        addressTA.webSite := self.getNextToken( line, pos );

        clientTA.persistEntityInTransState( TransactionType_Persist );
    endif;

endwhile;
```

Base Implementation

```
initialize() updating;

begin
    self.allErrors.clear();
    self.allWarnings.clear();

    self.expectedEdition    := null;
    self.focusField         := null;
    self.lockedByProcessId  := null;
    self.lockedTimeStamp    := null;
    self.modificationCode   := null;
    self.myModelObject      := null;
end;
```

lockForCreate

The protected **lockForCreate** method in the **ModelTA** class returns a Boolean value. This method is used to manually place an exclusive lock on objects during a create operation.

The following example shows a reimplementation of this method in a **ClientTA** class. The **app.myCompany.allClients** collection is being locked, because the **Client** object is an inverse of the **Company::allClients** collection.

```
lockForCreate() : Boolean protected;

begin
    if not self.tryLockingObject( app.myCompany.allClients ) then
        return false;
    endif;

    return true;
end;
```

This method returns **true** if the lock was applied successfully; otherwise it returns **false** if the lock was unable to be applied.

Base Implementation

```
lockForCreate() : Boolean protected;

begin
    return true;
end;
```

lockForDelete

The protected **lockForDelete** method in the **ModelTA** class returns a Boolean value. This method is used to manually place an exclusive lock on objects during a delete operation.

The following example shows a reimplementation of this method in a **ClientTA** class. The **app.myCompany.allClients** collection is being locked here because the **Client** object is an inverse of the **Company::allClients** collection.

```
lockForDelete() : Boolean protected;

begin
    if not self.tryLockingObject( app.myCompany.allClients ) then
        return false;
    endif;

    return true;
end;
```

This method returns **true** if the lock was applied successfully; otherwise it returns **false** if the lock was unable to be applied.

Base Implementation

```
lockForDelete() : Boolean protected;

begin
    return true;
end;
```

lockForModify

The protected **lockForModify** method in the **ModelTA** class returns a Boolean value. This method is used to manually place an exclusive lock on objects during a modify operation. A modify operation is a targeted update that sets only some of the properties on an object.

If there are no modify operations being performed, the reimplementation can simply return **true**, as shown in the following method example.

```
lockForModify() : Boolean protected;

begin
    return true;
end;
```

Alternatively, perform a lock on the necessary objects, as shown in the following code fragment.

```
if not self.tryLockingObject( app.myCompany.allClients ) then
    return false;
endif;

return true;
```

This method returns **true** if the lock was applied successfully; otherwise it returns **false** if the lock was unable to be applied.

Base Implementation

```
lockForModify() : Boolean protected;

begin
    return true;
end;
```

lockForUpdate

The protected **lockForUpdate** method in the **ModelTA** class returns a Boolean value. This method is used to place an exclusive lock on objects during a update operation.

The following example shows a reimplementation of this method in a **ClientTA** class.

```
lockForUpdate() : Boolean protected;

begin
    // is the key value on the object changing?
    // if it is, then we need to lock the collection
    if self.name <> self.getModelObject().name then
        if not self.tryLockingObject( app.myCompany.allClients ) then
            return false;
        endif;
    endif;

    return true;
end;
```

In the above method example, the **app.myCompany.allClients** collection is being locked because the **Client** object is an inverse of the **Company** class **allClients** collection. However, we only attempt to lock the object if the attribute used as a key value is being updated.

This method returns **true** if the lock was applied successfully; otherwise it returns **false** if the lock was unable to be applied.

Base Implementation

```
lockForUpdate() : Boolean protected;

begin
    return true;
end;
```

modifyEntity

The protected **modifyEntity** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to modify targeted properties when the system is not in transaction state. Modify is a targeted update operation that sets only some of the properties on an object. The targeted property or subset of properties depends on the modification code passed into the method.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pModification	Integer	The specific property or subset of properties to be modified. Passed in from the caller.

This method returns **true** if the object property or subset of properties was modified successfully; otherwise it returns **false** if the object property or subset of properties failed to be updated.

Base Implementation

```
modifyEntity( pModification : Integer ) : Boolean updating, final, protected;

vars
    transactionImplementor : CommitTransactionTI;

begin
    create transactionImplementor transient;

    return self.modifyEntityWithTransactionImplementor( transactionImplementor, pModification );

epilog
    delete transactionImplementor;
end;
```

modifyEntityInTransState

The protected **modifyEntityInTransState** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to modify targeted properties when the system is in transaction state. Modify is a targeted update that sets only some of the properties on an object. The targeted property or subset of properties depends on the modification code passed into the method.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pModification	Integer	The specific property or subset of properties to be modified. Passed in from the caller.

This method returns **true** if the object properties were modified successfully; otherwise it returns **false** if the object properties failed to be updated.

Base Implementation

```
modifyEntityInTransState( pModification : Integer ) : Boolean updating, final, protected;

vars
    transactionImplementor : NoTransactionTI;

begin
    create transactionImplementor transient;

    return self.modifyEntityWithTransactionImplementor( transactionImplementor, pModification );

epilog
    delete transactionImplementor;
end;
```

modifyEntityWithTransactionImplementor

The protected **modifyEntityWithTransactionImplementor** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to modify targeted properties. A **TransactionImplementor** object reference is passed as an argument to determine if the transaction should be committed to the database. Generally, this is a **CommitTransactionTI** object for general persists or a **NoTransactionTI** object for persisting subordinate objects or collections.

Modify is a targeted update operation that sets only some of the properties on an object. The targeted property or subset of properties depends on the modification code passed into the method.

The parameters for this method are listed in the following table.

Name	Type	Description
pTransactionImplementor	TransactionImplementor	The transaction implementor passed in determines if transaction state changes when a **doBeginTransaction**, **doCommitTransaction**, or **doAbortTransaction** method is called.
pModification	Integer	The specific property or subset of properties to be modified. This value should be stored as a global constant.

This method returns **true** if the object properties were modified successfully; otherwise it returns **false** if the object properties failed to be updated.

Base Implementation

```
modifyEntityWithTransactionImplementor(
        pTransactionImplementor : TransactionImplementor;
        pModification : Integer
        ) : Boolean updating, final, protected;

vars
    transactionCompleted    : Boolean;

begin
    self.clearErrors();
    self.clearWarnings();

    modificationCode := pModification;

    if self.doValidate( ValidationType_Modify ) then
        if self.lockForModify() then
            pTransactionImplementor.doBeginTransaction();
            if self.doModify( pModification ) then
                pTransactionImplementor.doCommitTransaction();
            else
                pTransactionImplementor.doAbortTransaction();
                return false;
            endif;
        else
            pTransactionImplementor.doAbortTransaction();
            self.addError( "Update conflicted with another user.", null );
            return false;
        endif;
    else
        return false;
    endif;

    transactionCompleted := true;

    return true;

epilog
    if not transactionCompleted then
        self.doAbortTransactionCleanup();
    endif;
end;
```

modifySubordinateObjects

The protected **modifySubordinateObjects** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to modify targeted properties on subordinate objects.

Modify is a targeted update operation that sets only some of the properties on an object. The targeted property or subset of properties depends on the modification code passed into the method.

A transaction implementor is passed as an argument to determine if the transaction should be committed to the database. Generally, this is a **CommitTransactionTI** object for general persists or a **NoTransactionTI** object for persisting subordinate objects or collections.

This method should be reimplemented in transactions agents that contain subordinate objects, as shown in the following example.

```
modifySubordinateObjects( pModification : Integer ) : Boolean updating, protected;

begin
    inheritMethod( pModification );

    if self.myAddressTA <> null;
    and not self.myAddressTA.persistEntityInTransState( pModification ) then
        // Error occurred, Copy AddressTa errors to self.allErrors attribute
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors;
end;
```

The parameter for this method is listed in the following table.

Name	Type	Description
pModification	Integer	The specific property or subset of properties to be modified. This value should be stored as a global constant.

This method returns **true** if the subordinate objects property or subset of properties were modified successfully; otherwise it returns **false** if the subordinate objects property or subset of properties failed to be updated.

Base Implementation

```
modifySubordinateObjects( pModification : Integer ) : Boolean updating, protected;

begin
    return true;
end;
```

persistEntity

The public **persistEntity** method in the **ModelTA** class returns a Boolean value. This method is the main method called to persist (create, update, modify, or delete) an object to the database when the system is not in a transaction state.

The following code fragment is an example of creating or updating a **Client** object.

```
clientTA.persistEntity( TransactionType_Persist );
```

The following code fragment is an example of deleting a **Client** object.

```
clientTA.persistEntity( TransactionType_Delete );
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTransactionType	Integer	The type of transaction to perform; that is: ■ A **TransactionType_Delete** triggers a delete transaction. ■ A **TransactionType_Persist** triggers a create or update transaction. ■ A transaction type with values larger than a persist transaction indicate a modification. Modification codes should be stored as global constants.

This method returns **true** if the object was persisted successfully; otherwise it returns **false** if the object failed to be persisted.

Base Implementation

```
persistEntity( pTransactionType : Integer ) : Boolean updating, final;

begin
    // Check for Full_delete first (aka -1)
    if pTransactionType = TransactionType_Delete then
        if self.getModelObject() = null then
            // Rather than raise an exception, just treat this as a
            // no-op instead and return true.
            return true;
        endif;

        return self.deleteEntity();

    // Check for Full_update next (aka 0)
    elseif pTransactionType = TransactionType_Persist then
        if self.getModelObject() = null then
            return self.createEntity();
        else
            return self.updateEntity();
        endif;

    // Check for a modification number
    elseif pTransactionType > TransactionType_Persist then
        return self.modifyEntity( pTransactionType );

    else
        return false;
    endif;
end;
```

persistEntityInTransState

The public **persistEntityInTransState** method in the **ModelTA** class returns a Boolean value. This method is the main method called to persist (create, update, modify, or delete) an object to the database when the system is in a transaction state.

This method is commonly used to persist subordinate objects or when persisting multiple objects in a single transaction (for example, in a collection of objects).

The following code fragment is an example of creating or updating an **Address** object from a **ClientTA** object.

```
self.myAddressTA.persistEntityInTransState(TransactionType_persist);
```

Tip Because the **Address** object (child) is inversed to the **Client** (parent), we do not need a **persistEntityInTransState** method to perform a delete operation, as child objects are automatically deleted when the parent object is deleted. This is one huge benefit of using inverses.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTransactionType	Integer	The type of transaction to perform; that is: ■ A **TransactionType_Delete** triggers a delete transaction. ■ A **TransactionType_Persist** triggers a create or update transaction. ■ A transaction type with values larger than a persist transaction indicate a modification. Modification codes should be stored as global constants.

This method returns **true** if the object was persisted successfully; otherwise it returns **false** if the object failed to be persisted.

Base Implementation

```
persistEntityInTransState( pTransactionType : Integer ) : Boolean updating, final;

begin
    // Check for Full_delete first (aka -1)
    if pTransactionType = TransactionType_Delete then
        if self.getModelObject() = null then
            return true;
        endif;

        return self.deleteEntityInTransState();

    // Check for Full_update next (aka 0)
    elseif pTransactionType = TransactionType_Persist then
        if self.getModelObject() = null then
            return self.createEntityInTransState();
        else
            return self.updateEntityInTransState();
        endif;

    // Check for a modification number
    elseif pTransactionType > TransactionType_Persist then
        return self.modifyEntityInTransState( pTransactionType );

    else
        return false;
    endif;
end;
```

populateFromObject

The public **populateFromObject** method in the **ModelTA** class copies the **PersistentModel** object properties into the properties of the transaction agent.

This method is generally used on all transaction agents and should be reimplemented accordingly. In addition, the parameter name and type should be changed to be more specific.

The following reimplementation example shows the properties in a **Client** object being copied to the **ClientTA** properties.

```
populateFromObject( pClient : Client ) updating;

begin
    inheritMethod( pClient );

    self.name := pClient.name;
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pModelObject	PersistentModel	The object from which to copy properties. The object and the transaction agent should both have the same properties.

Base Implementation

```
populateFromObject( pModelObject : PersistentModel ) updating;

begin
    self.myModelObject  := pModelObject;

    self.populateSubordinateObjects( pModelObject );
end;
```

populateSubordinateObjects

The protected **populateSubordinateObjects** method in the **ModelTA** class copies the **PersistentModel** object properties of the subordinate objects into the respective subordinate transaction agents.

Reimplement this method in all transaction agents that contain subordinate transaction agents. In addition, the parameter name and type should be changed to be more specific.

The following example shows an implementation of this method where an **Address** is being referenced to a **Client** object.

```
populateSubordinateObjects( pClient : Client ) updating, protected;

vars
    addressTA : AddressTA;

begin
    inheritMethod( pClient );

    create addressTA transient;
    addressTA.populateFromObject( pClient.myAddress );
    addressTA.myModelTA := self;
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pModelObject	PersistentModel	The object from which to copy properties. The object and the transaction agent should both have the same properties.

Base Implementation

This method is intentionally left blank, as not all transaction agents have subordinate objects.

```
populateSubordinateObjects( pModelObject : PersistentModel ) updating, protected;

begin

end;
```

tryLockingObject

The protected **tryLockingObject** method in the **ModelTA** class returns a Boolean value. This method attempts to place an exclusive lock on the object passed in as the parameter.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pObject	Object	The object to be exclusively locked

This method returns **true** if the object was successfully locked; otherwise it returns **false** if the object was unable to be locked.

Base Implementation

```
tryLockingObject( pObject : Object ) : Boolean protected, final;
// Try to get a lock on the Object, or return false

begin
    return self.tryLock( pObject, Exclusive_Lock, Transaction_Duration, LockTimeout_Server_Defined );
end;
```

updateEntity

The protected **updateEntity** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to update an object when the system in not in transaction state.

Note This method should not be reimplemented.

This method returns **true** if the object was updated successfully; otherwise it returns **false** if the object failed to be updated.

Base Implementation

```
updateEntity() : Boolean updating, final, protected;

vars
    transactionImplementor  : CommitTransactionTI;

begin
    create transactionImplementor transient;

    return self.updateEntityWithTransactionImplementor( transactionImplementor );

epilog
    delete transactionImplementor;
end;
```

updateEntityInTransState

The protected **updateEntityInTransState** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to update an object when the system is in transaction state.

Note This method should not be reimplemented.

This method returns **true** if the object was updated successfully; otherwise it returns **false** if the object failed to be updated.

Base Implementation

```
updateEntityInTransState() : Boolean updating, final, protected;

vars
    transactionImplementor  : NoTransactionTI;

begin
    create transactionImplementor transient;

    return self.updateEntityWithTransactionImplementor( transactionImplementor );

epilog
    delete transactionImplementor;
end;
```

updateEntityWithTransactionImplementor

The protected **updateEntityWithTransactionImplementor** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to update an object. A **TransactionImplementor** is passed as a parameter to determine if the transaction should be committed to the database. Generally, the transaction implementor is a **CommitTransactionTI** for general persists or a **NoTransactionTI** for persisting subordinate objects or collections.

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTransactionImplementor	TransactionImplementor	The transaction implementor passed in determines if transaction state changes when the **TransactionImplementor** class **doBeginTransaction**, **doCommitTranasction**, or **doAbortTransaction** method is called

This method returns **true** if the object was updated successfully; otherwise it returns **false** if the object failed to be updated.

Base Implementation

```
updateEntityWithTransactionImplementor(
        pTransactionImplementor : TransactionImplementor
        ) : Boolean updating, final, protected;

vars
    validateRtn : Boolean;
    transactionCompleted    : Boolean;

begin
    self.modificationCode:= null;
    self.clearErrors();
    self.clearWarnings();

    validateRtn := self.doValidate( ValidationType_Update );

    if validateRtn then
        if self.lockForUpdate() then
            if not self.checkEdition( self.expectedEdition ) then
                self.addError( "Another user updated this entity", null );
                return false;
            endif;
            pTransactionImplementor.doBeginTransaction();
            if self.doUpdate() then
                pTransactionImplementor.doCommitTransaction();
            else
                pTransactionImplementor.doAbortTransaction();
                return false;
            endif;
        else
            pTransactionImplementor.doAbortTransaction();
            self.addError( "Update conflicted", null );
            return false;
        endif;
    else
        return false;
    endif;

    transactionCompleted    := true;

    return true;

epilog

    if not transactionCompleted then
        self.doAbortTransactionCleanup();
    endif;
end;
```

updateSubordinateObjects

The protected **updateSubordinateObjects** method in the **ModelTA** class returns a Boolean value. This method is programmatically called to update subordinate objects.

A **TransactionImplementor** is passed as a parameter to determine if the transaction should be committed to the database. Generally, the transaction implementor is a **CommitTransactionTI** for general persists or a **NoTransactionTI** for persisting subordinate objects or collections.

Reimplement this method in transaction agents that contain subordinate objects, as shown in the following example.

```
updateSubordinateObjects() : Boolean updating, protected;

begin
    inheritMethod();

    // Update the Address object
    if self.myAddressTA <> null
    and not self.myAddressTA.persistEntityInTransState( TransactionType_Persist ) then
        // Error occurred, Copy AddressTA errors to self.allErrors attribute
        self.copyErrors( self.myAddressTA );
    endif;

    return self.hasNoErrors;
end;
```

This method returns **true** if the object was updated successfully; otherwise it returns **false** if the object failed to be updated.

Base Implementation

```
updateSubordinateObjects() : Boolean updating, protected;

begin
    return true;
end;
```

TransactionImplementor Class

The TransactionImplementor (TI) classes contain specific behaviors to perform at different stages of a persistent transaction.

TI classes are injected into the ***xxxxxxx*EntityWithTransactionImplementor** methods (for example, **createEntityWithTransactionImplementor**) in the **ModelTA** class as a dependency, allowing the methods to call the same functions but perform different behaviors specific to the transaction implementor provided. This technique is referred to as *Dependency Injection*.

Each transaction implementor has different behavioral implementations for begin, commit, and abort operations.

Beginning a Transaction

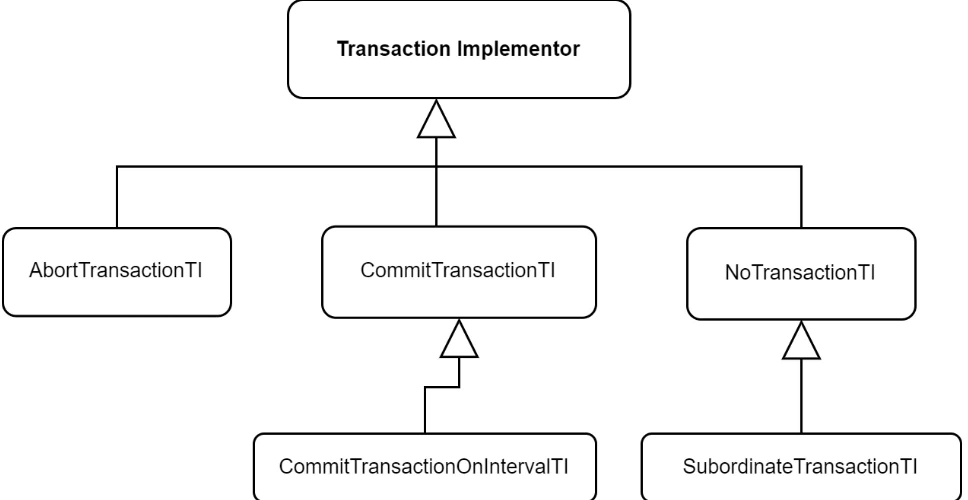
To perform a persistent operation in Jade (saving, updating, or deleting objects from a database), the system must first be placed into a transaction state. This is generally done by calling the **beginTransaction** instruction prior to performing persistent operations. This behavior is handled by the **doBeginTransaction** method in the TI.

Committing a Transaction

To commit a transaction to the database, Jade uses the **commitTransaction** instruction. This functionality is handled by the **doCommitTransaction** method in the TI.

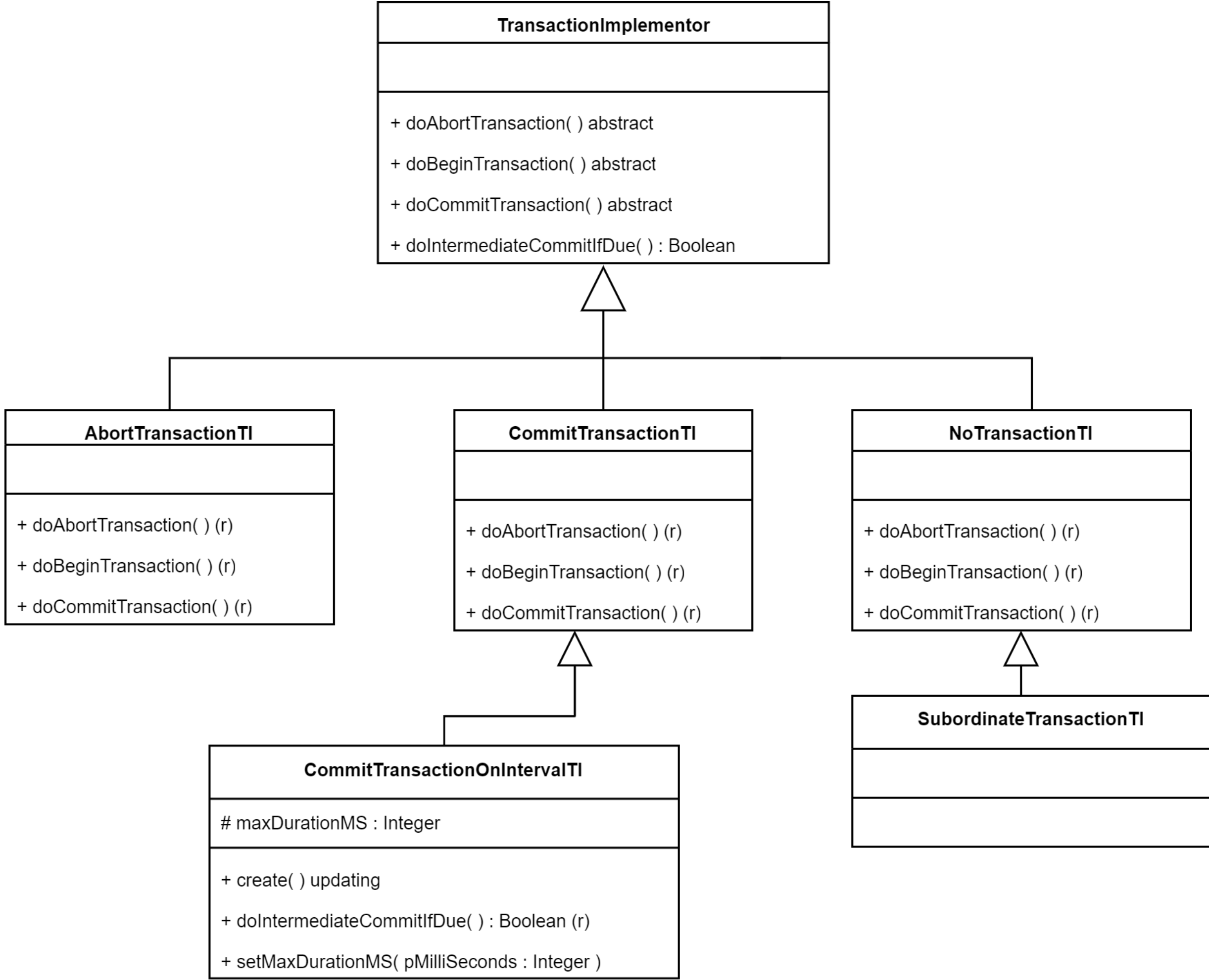
Aborting a Transaction

To abort a transaction (in the case of an error or other reason), Jade uses the **abortTransaction** instruction. This functionality is handled by the **doAbortTransaction** method in the TI.



TransactionImplementor Class Diagram

The following diagram shows the properties and methods defined in the **TransactionImplementor** class.



TransactionImplementor Abstract Class

The abstract transaction implementor class is reimplemented in child classes to ensure that the transaction methods are available in each child class.

The **TransactionImplementor** abstract class public methods are summarized in the following table.

Method	Return Type	Description
doAbortTransaction	Not applicable	Generally called to abort a transaction
doBeginTransaction	Not applicable	Generally called to start a transaction
doCommitTransaction	Not applicable	Generally called to commit a transaction
doIntermediateCommitIfDue	Boolean	For use on longer background tasks to commit periodically after an elapsed time

doAbortTransaction (Abstract)

The public **doAbortTransaction** abstract method in the **TransactionImplementor** abstract class is generally called to abort a transaction.

Base Implementation

```
doAbortTransaction() abstract;
```

doBeginTransaction (Abstract)

The public **doBeginTransaction** abstract method in the **TransactionImplementor** abstract class is generally called to start a transaction.

Base Implementation

```
doBeginTransaction() abstract;
```

doCommitTransaction (Abstract)

The public **doCommitTransaction** abstract method in the **TransactionImplementor** abstract class is generally called to commit a transaction.

Base Implementation

```
doCommitTransaction() abstract;
```

doIntermediateCommitIfDue

The public **doIntermediateCommitIfDue** abstract method in the **TransactionImplementor** abstract class is for use on longer background tasks to commit periodically after an elapsed time, and it is reimplemented only in **CommitTransactionOnIntervalTI** abstract classes.

This method (in a reimplementation only) returns **true** if the elapsed time has passed and therefore should commit. By default, this method returns **false**; that is, it does nothing.

Base Implementation

```
doIntermediateCommitIfDue() : Boolean;

begin
    return false;
end;
```

AbortTransactionTI Class

The **AbortTransactionTI** transaction implementor class will go into a transaction state but will never commit, as the **doCommitTransaction** method will abort the transaction.

This transaction implementor class can be used for:

- Unit testing
- Previewing code (where the preview is doing some persistent updating to get the result for displaying and then aborting)

The **AbortTransactionTI** class public methods are summarized in the following table.

Method	Description
doAbortTransaction	Aborts the transaction by calling the **abortTransaction** instruction
doBeginTransaction	Places the system into a transaction state
doCommitTransaction	Called to commit a transaction; however, the transaction will not commit because the **abortTransaction** instruction will be called instead

doAbortTransaction

The public **doAbortTransaction** method in the **AbortTransactionTI** class aborts the transaction by calling the **abortTransaction** instruction.

Base Implementation

The following example calls a method in the **app** object to perform the abort action because the same logic may be used in other abort methods. In addition, we want to call our **app** notification manager to clear any scheduled notifications.

```
doAbortTransaction();

begin
    app.erewhonAbortTransaction();
end;
```

The **erewhonAbortTransaction** method in the following example is responsible for aborting the transaction and clearing any scheduled notifications in the notification manager. This logic could be placed directly in the **doAbortTransaction** method; however, because the **app** shares this logic, we have separated it into its own method.

```
erewhonAbortTransaction();

begin
    abortTransaction;

    if self.myNotificationManager <> null then
        self.myNotificationManager.clearScheduledNotifications();
    endif;
end;
```

doBeginTransaction

The public **doBeginTransaction** method in the **AbortTransactionTI** class places the system into a transaction state.

Base Implementation

```
doBeginTransaction();

begin
    beginTransaction;
end;
```

doCommitTransaction

The public **doCommitTransaction** method in the **AbortTransactionTI** class is called to commit a transaction; however, the transaction will not commit because the **abortTransaction** instruction will be called instead.

Base Implementation

```
doCommitTransaction();

begin
    // this implementor will NEVER commit
    app.erewhonAbortTransaction();
end;
```

The **erewhonAbortTransaction** method in the following example is responsible for aborting the transaction and clearing any scheduled notifications in the notification manager. This logic could be placed directly in the **doCommitTransaction** method; however, because the **app** shares this logic, we have separated it into its own method.

```
erewhonAbortTransaction();

begin
    abortTransaction;

    if self.myNotificationManager <> null then
        self.myNotificationManager.clearScheduledNotifications();
    endif;
end;
```

CommitTransactionTI Class

The **CommitTransactionTI** transaction implementor class is used for normal persistence of objects during create, update, modify, and delete operations.

The **CommitTransactionTI** class public methods are summarized in the following table.

Method	Description
doAbortTransaction	Aborts the transaction by calling the **abortTransaction** instruction
doBeginTransaction	Places the system into a transaction state
doCommitTransaction	Commits a transaction and therefore takes the system out of transaction state

doAbortTransaction

The public **doAbortTransaction** method in the **CommitTransactionTI** class aborts the transaction by calling the **abortTransaction** instruction.

Base Implementation

The following example calls a method in the **app** object to perform the abort action because the same logic may be used in other abort methods. In addition, we want to call our **app** notification manager to clear any scheduled notifications.

```
doAbortTransaction();

begin
    app.erewhonAbortTransaction();
end;
```

The **erewhonAbortTransaction** method in the following example is responsible for aborting the transaction and clearing any scheduled notifications in the notification manager. This logic could be placed directly in the **doAbortTransaction** method; however, because the **app** shares this logic, we have separated it into its own method.

```
erewhonAbortTransaction();

begin
    abortTransaction;

    if self.myNotificationManager <> null then
        self.myNotificationManager.clearScheduledNotifications();
    endif;
end;
```

doBeginTransaction

The public **doBeginTransaction** method in the **CommitTransactionTI** class places the system into a transaction state.

Base Implementation

```
doBeginTransaction();

begin
    beginTransaction;
end;
```

doCommitTransaction

The public **doCommitTransaction** method in the **CommitTransactionTI** class commits a transaction and therefore takes the system out of transaction state.

Base Implementation

```
doCommitTransaction();

begin
    app.erewhonCommitTransaction();
end;
```

The **erewhonAbortTransaction** method in the following example is responsible for committing the transaction and sending any scheduled notifications using a notification manager. This logic could be placed directly in the **doCommitTransaction** method, however because the **app** shares this logic, we have separated it into its own method.

```
erewhonCommitTransaction();

begin
    commitTransaction;

    if self.myNotificationManager <> null then
        self.myNotificationManager.sendScheduledNotifications();
    endif;
end;
```

CommitTransactionOnIntervalTI Class

The **CommitTransactionOnIntervalTI** class behaves exactly the same as the **CommitTransactionTI** class, but there is an additional **doIntermediateCommitIfDue** method that can be used when a longer background task is running and you want to commit periodically.

Note This implementor performs a commit action only if the system has been in a transaction state for longer than 3,000 milliseconds (that is, 3 seconds) from the time the system was placed into a transaction state.

It is up to the developer to call the **doIntermediateCommitIfDue** method at an appropriate point in the processing, as the interim begin and commit actions do not occur automatically. If you want a different interval instead of the default value of **3** seconds, use the **setMaxDurationMS** method to override the default value.

Tip The specified value should be suitable to avoid contention with other users and to avoid the transaction getting very large so that it has a negative impact on performance, but also not so short that it has a negative impact from the associated overhead when doing commits to the database.

The protected property in the **CommitTransactionOnIntervalTI** class is listed in the following table.

Property	Description
maxDurationMS	Integer value that specifies the number of milliseconds from when the system was placed in a transaction state until a commit transaction can occur

The public methods in the **CommitTransactionOnIntervalTI** class are summarized in the following table.

Method	Description
create	Executed when the object is first created
doIntermediateCommitIfDue	Commits periodically after an elapsed time for longer background tasks
setMaxDurationMS	Sets the maximum number of elapsed milliseconds after which to commit the action

create

The public **create** method in the **CommitTransactionOnIntervalTI** class is executed when the object is first created.

This method sets the value of the **maxDurationMS** property to the default maximum duration value of **3,000** milliseconds (that is, 3 seconds).

Base Implementation

```
create() updating;

begin
    self.maxDurationMS := Default_MaxDuration;
end;
```

doIntermediateCommitIfDue

The public **doIntermediateCommitIfDue** method in the **CommitTransactionOnIntervalTI** class returns a Boolean value. This method is used on longer background tasks to commit periodically after an elapsed time.

If the system has been in a transaction state for longer than the maximum amount of the time specified in the **maxDurationMS** property, the system commits the transaction and sets the system back into a transaction state ready for the next commit interval, as shown in the following example.

```
doIntermediateCommitIfDue() : Boolean;

begin
    if system.getTimeInTransactionState( process ) > self.maxDurationMS then
        self.doCommitTransaction();
        self.doBeginTransaction();
        return true;
    endif;

    return false;
end;
```

This method returns **true** if the maximum duration has been exceeded and the transaction has been committed; otherwise it returns **false** if the duration has not been exceeded.

Base Implementation

```
doIntermediateCommitIfDue() : Boolean;

begin
    return false;
end;
```

maxDurationMS

The protected **maxDurationMS** property in the **CommitTransactionOnIntervalTI** class has an Integer value that specifies the number of milliseconds from when the system was placed in a transaction state until a commit transaction can occur.

The default value of **3,000** milliseconds (3 seconds) is set in the **CommitTransactionOnIntervalTI** class **create** method. To override this default value, call the **setMaxDurationMS** method.

setMaxDurationMS

The public **setMaxDurationMS** method in the **CommitTransactionOnIntervalTI** class sets the value of the **maxDurationMS** property to a different interval (in milliseconds).

The default interval is **3,000** milliseconds (that is, **3** seconds) set by the **create** method.

Base Implementation

```
setMaxDurationMS( pMilliSeconds : Integer) updating;

begin
    self.maxDurationMS := pMilliSeconds;
end;
```

NoTransactionTI Class

The **NoTransactionTI** transaction implementor class does not perform any **beginTransaction**, **commitTransaction**, or **abortTransaction** instruction behavior and is primarily used when creating subordinate objects for the parent object while in a transaction state.

The parent object is responsible for beginning, committing, and aborting the transaction.

The public methods in the **NoTransactionTI** class are summarized in the following table.

Method	Called when the...
doAbortTransaction	Transaction should abort
doBeginTransaction	System should be in placed transaction state
doCommitTransaction	Should commit

doAbortTransaction

The public **doAbortTransaction** method in the **NoTransactionTI** class is called when the transaction should abort. As this implementor is used for subordinate objects, the parent object is responsible for aborting the transaction, so this method does nothing.

Base Implementation

```
doAbortTransaction();

begin
    // do nothing
end;
```

doBeginTransaction

The public **doBeginTransaction** method in the **NoTransactionTI** class is called when the system should be in transaction state. As this implementor is used for subordinate objects, the parent object should already be in a transaction state, so this method does nothing.

Base Implementation

```
doBeginTransaction();

begin
    // do nothing
end;
```

doCommitTransaction

The public **doCommitTransaction** method in the **NoTransactionTI** class is called when the transaction should commit. As this implementor is used for subordinate objects, the parent object is responsible for committing the transactions, so this method does nothing.

Base Implementation

```
doCommitTransaction();

begin
    // do nothing
end;
```

SubordinateTransactionTI Class

The **SubordinateTransactionTI** class is for use by subordinate TA methods where the parent TA is responsible for controlling the transaction state.

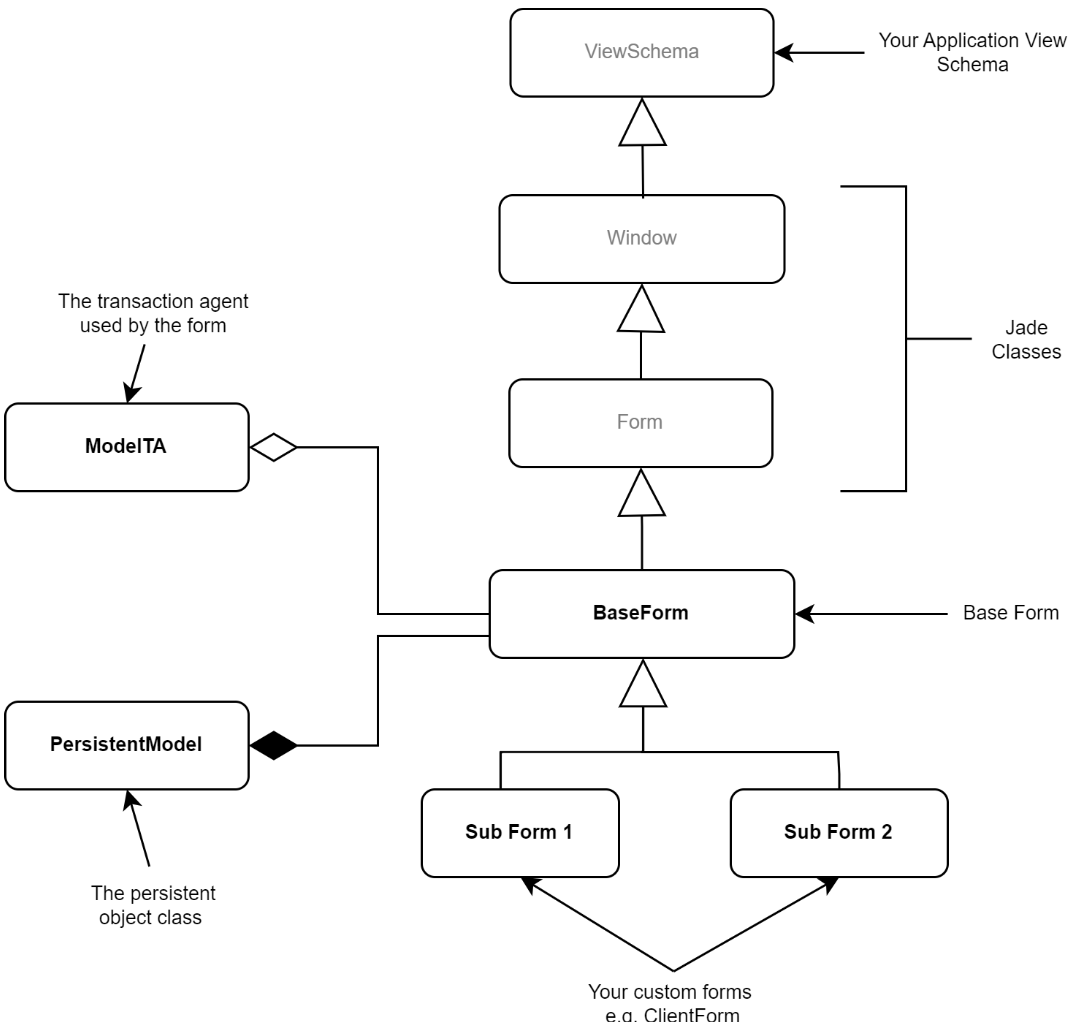
The inherited **doBeginTransaction**, **doCommitTransaction**, and **doAbortTransaction** methods do nothing!

This class behaves the same as the **NoTransactionTI** class. The class name implies a more-specific usage.

BaseForm Class

Most of the time, the user interacts with the system through forms. These forms need to work with the transaction agent through a **BaseForm** class.

The **BaseForm** class should be the parent class of all forms that use the transaction agent framework, and it generally does not contain any controls or menus but it *does* contain the properties and methods for interacting with the transaction agent.



The above diagram shows the **BaseForm** class as a child of the Jade **Form** object. By convention, all user interfaces and views should be a subclass of a ***YourAppNameViewSchema*** schema.

BaseForm Class Diagram

The **BaseForm** class contains a reference to the **ModelTA** transaction agent and a **PersistentModel**. The form data is populated in the **PersistentModel** and passed to the **ModelTA** for processing.

BaseForm

+ myCurrentObject : PersistentModel
+ myTA : ModelTA

displayErrors( )
displayObject(pObject : PersistentModel)
doDelete( ) : Boolean
doSave( ) : Boolean
formLoad( )
formUnload( )
+ getCurrentObject( ) : PersistentModel
+ getTA( ) : ModelTA
# getTAClass( ) : Class
# populateTAFromForm(pTA : ModelTA input) : Boolean
# processAfterDelete(pTA : ModelTA input)
# processAfterSave(pTA : ModelTA)
+ setContextObject(pContext : PersistentModel)

EditClientForm

displayObject(pClient : Client)
+ getCurrentObject( ) : Client
+ getTA( ) : ClientTA
# getTAClass( ) : Class
# populateTAFromForm(pClientTA : ClientTA input) : Boolean
# processAfterDelete(pClientTA : ClientTA input)
# processAfterSave(pClientTA : ClientTA)

EditClientForm Code Implementation Examples

The **EditClientForm** class reimplements the methods summarized in the following table. For code implementation examples of these methods, see the following subsections.

Method	Description
displayObject	Populates the form controls using the properties of the **pClient** parameter
getCurrentObject	Gets the **Client** object stored in the **myCurrentObject** property
getTA	Gets the **ClientTA** transaction agent instance used by the form
getTAClass	Gets the class type of the **ClientTA** transaction agent
populateTAFromForm	Populates the **ClientTA** transaction agents from the respective form control values
processAfterDelete	Tasks to perform after a successful delete action
processAfterSave	Tasks to perform after a successful save action

displayObject

The **displayObject** method in the **EditClientForm** class populates the form controls with the values contained in the **pClient** parameter.

```
displayObject( pClient : Client ) updating, protected;

vars
    address : Address;

begin
    inheritMethod( pClient );

    if pClient = null then
        self.caption := $Add & " " & self.caption;
    else
        self.caption := $Edit & " " & self.caption;
        self.txtName.text := pClient.name;

        address := pClient.myAddress;

        self.txtStreet.text := address.street;
        self.txtCity.text := address.city;
        self.txtCountry.text := address.country;
        self.txtPhone.text := address.phone;
        self.txtFax.text := address.fax;
        self.txtEmail.text := address.email;
        self.txtWebSite.text := address.webSite;
    endif;
end;
```

getCurrentObject

The **getCurrentObject** method in the **EditClientForm** class returns the object stored in the **myCurrentModel** reference returned by the **inheritMethod** instruction and typecasts it to the more-specific **Client** type.

```
getCurrentObject() : Client;

begin
    return inheritMethod().Client;
end;
```

getTA

The **getTA** method in the **EditClientForm** class gets the transaction agent stored in the **myTA** property returned by the **inheritMethod** instruction and typecasts it to the more-specific **ClientTA** type.

```
getTA() : ClientTA;

begin
    return inheritMethod().ClientTA;
end;
```

getTAClass

The **getTAClass** method in the **EditClientForm** class gets the **Class** class instance for the **ClientTA** transaction agent used by the form.

```
getTAClass() : Class protected;

begin
    return ClientTA;
end;
```

populateTAFromForm

The **populateTAFromForm** method in the **EditClientForm** class populates the transaction agent properties with the respective values from the controls of the form.

```
populateTAFromForm( pClientTA : ClientTA input ) : Boolean protected;

vars
    addressTA : AddressTA;

begin
    if not inheritMethod( pClientTA ) then
        return false;
    endif;

    pClientTA.name := self.txtName.text.trimWhiteSpace();

    addressTA := pClientTA.myAddressTA;

    // create the addressTA if it does not exist
    if addressTA = null then
        create addressTA transient;
        addressTA.myModelTA := pClientTA;
    endif;

    addressTA.street := self.txtStreet.text.trimWhiteSpace();
    addressTA.city := self.txtCity.text.trimWhiteSpace();
    addressTA.country := self.txtCountry.text.trimWhiteSpace();
    addressTA.phone := self.txtPhone.text.trimWhiteSpace();
    addressTA.fax := self.txtFax.text.trimWhiteSpace();
    addressTA.email := self.txtEmail.text.trimWhiteSpace();
    addressTA.webSite := self.txtWebSite.text.trimWhiteSpace();

    return true;
end;
```

processAfterDelete

The **processAfterDelete** method in the **EditClientForm** class performs tasks after a delete operation has been successfully performed. In this case, we are showing a message box to alert the user that the **Client** object was deleted.

```
processAfterDelete( pClientTA : ClientTA input ) protected;

begin
    inheritMethod( pClientTA );

    app.msgBox( 'Client ' & pClientTA.name & ' has been deleted', 'Success', MsgBox_OK_Only );
end;
```

processAfterSave

The **processAfterSave** method in the **EditClientForm** class performs tasks after a save operation has been successfully performed. In this case, we are showing a message box to alert the user that the **Client** object was saved.

```
processAfterSave( pClientTA : ClientTA ) updating, protected;

begin
    inheritMethod( pClientTA );

    app.msgBox( 'Client ' & pClientTA.name & ' has been saved', 'Success', MsgBox_OK_Only );
end;
```

BaseForm Properties

The public properties defined in the **BaseForm** class are summarized in the following table.

Property	Value	Stores the...
myCurrentObject	PersistentModel	Object for which the form and transaction agent are responsible
myTA	ModelTA	Transaction agent responsible for persisting and displaying the **PersistentModel** object

BaseForm Methods

The methods defined in the **BaseForm** class are summarized in the following table.

Method	Description
displayErrors	Displays the errors stored in the **allErrors** collection after a failed delete or save operation
displayObject	Displays the persistent object properties on the form
doDelete	Performs a persistent delete of the object stored in the **myCurrentObject** property
doSave	Performs a persistent save of the object stored in the **myCurrentObject** property
formLoad	Performs specific form load behavior
formUnload	Performs specific form unload behavior
getCurrentObject	Gets the **PersistentModel** object stored in the **myCurrentObject** property of the form
getTA	Returns the transaction agent stored in the **myTA** property
getTAClass	Returns the type of transaction agent used by the form
populateTAFromForm	Populates the transaction agent of the form with the form control values
processAfterDelete	Performs tasks after a successful delete operation
processAfterSave	Performs tasks after a successful save operation
setContextObject	Populate the **myCurrentObject** property of the form with the **PersistentModel** specified in the **pContext** parameter

displayErrors

The protected **displayErrors** method in the **BaseForm** class displays the errors stored in the **allErrors** collection after a failed delete or save operation.

Parameters

The parameters for this method are listed in the following table.

Name	Type	Description
pTA	ModelTA	The transaction agent being used to store the errors
pTitle	String	Title for the message box

Base Implementation

```
displayErrors( pTA : ModelTA; pTitle : String ) updating, protected;

vars
    msg : String;
    error : String;
    title : String;

begin
    if pTA <> null then
        foreach error in pTA.allErrors do
            msg.appendTextWithDelim( CrLf, error, false );
        endforeach;
    endif;

    title := pTitle;
    if title = null then
        title := "Error during save";
    endif;

    //set focus field when encountered an error
    self.setFocusField( pTA.focusField );

    app.msgBox ( msg , title , MsgBox_Information_Icon + MsgBox_OK_Only );
end;
```

displayObject

The protected **displayObject** method in the **BaseForm** class displays the persistent object properties on the form.

The reimplementation of this method should provide a more-specific parameter. In the following example, a **Client** instance has been provided as the parameter.

```
displayObject( pClient : Client ) updating, protected;

vars
    address : Address;

begin
    inheritMethod( pClient );

    if pClient = null then
        self.caption := $Add & " " & self.caption;
    else
        self.caption := $Edit & " " & self.caption;
        self.txtName.text := pClient.name;

        address := pClient.myAddress;

        self.txtStreet.text := address.street;
        self.txtCity.text := address.city;
        self.txtCountry.text := address.country;
        self.txtPhone.text := address.phone;
        self.txtFax.text := address.fax;
        self.txtEmail.text := address.email;
        self.txtWebSite.text := address.webSite;
    endif;
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pClient	Client	Persistent client containing the properties to be displayed

Base Implementation

```
displayObject( pObject : PersistentModel ) updating, protected;

vars
    msgReturn   : Integer;

begin
    if pObject <> null and pObject.lockedByProcessId <> null then
        msgReturn := app.msgBox(
                            "This " & self.thing() & " is currently being edited.
                            Would you like to send a 'Save' request so that changes made are saved?",
                            self.thing() & " already being dited", MsgBox_Question_Mark_Icon + MsgBox_Yes_No );

        if msgReturn = MsgBox_Return_Yes then
            pObject.causeEvent( Erewhon_Event_SaveOutstandingChanges, true, Erewhon_Event_SaveOutstandingChanges );
        endif;
    endif;

    self.myCurrentObject := pObject;
end;
```

doDelete

The protected **doDelete** method in the **BaseForm** class returns a Boolean value. This method is called by the form to perform a persistent delete of the object stored in the **myCurrentObject** property.

The method in the following example shows the **doDelete** method being called when the **Remove** button is clicked.

```
btnRemove_click( btn : Button input ) updating;

begin
    app.mousePointer := Busy;

    if lstClients.listObject <> null then
        self.myCurrentObject := lstClients.listObject.Client;
        self.doDelete();
    endif;

epilog
    app.mousePointer := Idle;
end;
```

This method returns **true** if the delete operation was successful; otherwise it returns **false** if the delete operation was unsuccessful.

Base Implementation

```
doDelete() : Boolean updating, protected;

vars
    ta  : ModelTA;

begin
    ta := self.getTA();

    if ta = null then
        create self.myTA as self.getCurrentObject().getTAClass();
        ta := self.myTA;
    endif;

    ta.populateFromObject( self.getCurrentObject() );

    if not ta.persistEntity(TransactionType_Delete) then
        self.displayErrors( ta, null );
        return false;
    endif;

    self.processAfterDelete( ta );

    // Clean up references to invalid objects
    delete self.myTA;
    self.myCurrentObject := null;

    return true;
end;
```

doSave

The protected **doSave** method in the **BaseForm** class returns a Boolean value. This method is called by the form to perform a persistent save of the object stored in the **myCurrentObject** property.

The method in the following example shows the **doSave** method being called when the **Ok** button is clicked. If the method returns **false**, we exit the button **click** method by returning early.

```
btnOk_click( btn : Button input ) updating;

begin
    app.mousePointer := Busy;

    if not self.doSave() then
        return;
    endif;

    self.modalResult := ModalOK;
    self.unloadForm();

epilog
    app.mousePointer := Idle;
end;
```

This method returns **true** if the save operation was successful; otherwise it returns **false** if the save operation was unsuccessful.

Base Implementation

```
doSave() : Boolean updating, protected;

vars
    ta  : ModelTA;

begin

    ta := self.getTA();

    if ta = null then
        create self.myTA as self.getTAClass();
        ta := self.myTA;

        if self.getCurrentObject() <> null then
            ta.populateFromObject( self.getCurrentObject() );
        endif;
    endif;

    if not self.populateTAFromForm( ta ) then
        return false;
    endif;

    if not ta.persistEntity(TransactionType_Persist) then
        self.displayErrors( ta, null );
        return false;
    endif;

    self.processAfterSave( ta );

    return true;
end;
```

formLoad

The protected **formLoad** method in the **BaseForm** class is a proxy method to perform specific form load behavior. This method is called by the inbuilt Jade **load** method for forms.

This method should be used to disable or enable buttons or other initial setup that does not rely on the specific values from the transaction agent, as shown in the following example.

```
formLoad() updating, protected;

begin
    inheritMethod();

    self.lstClients.listCollection( app.myCompany.allClients, true, 0 );

    self.btnEdit.enabled := false;
    self.btnRemove.enabled := false;
end;
```

Base Implementation

```
formLoad() updating, protected;

vars
    keys  : IntegerArray;

begin
    create keys transient;
    self.formGetRegisterKeys( keys );
    self.registerFormKeys( keys );
epilog
    delete keys;
end;
```

formUnload

The protected **formUnload** method in the **BaseForm** class is a proxy method to perform specific form unload behavior. This method is called by the inbuilt Jade **unload** method for forms.

This method can be used to automatically save an object or unsubscribe from notifications. For example, the following logic can be used to delete transient objects and end notifications.

```
formUnload() updating, protected;

begin
    inheritMethod();

    delete self.myItemSearch;

    self.endNotificationForSubscriber( self );
end;
```

Base Implementation

```
formUnload() updating, protected;

begin
    if self.myCurrentObject <> null and self.myCurrentObject.isSoftLockedByMe() then
        if self.myTA = null then
            create self.myTA as self.getTAClass();
            self.myTA.populateFromObject( self.myCurrentObject );
        endif;

        self.myTA.persistEntity( TransactionType_Modify_UnlockEntity );
    endif;

    self.endNotificationForSubscriber( self );
    delete self.myTA;
end;
```

getCurrentObject

The protected **getCurrentObject** method in the **BaseForm** class returns a **PersistentModel** value. This method, is used to get the **PersistentModel** object stored in the **myCurrentObject** property of the form.

When reimplementing this method, the return type should be more-specific and the inherited method type cast with the specific **PersistentModel** subclass, as shown in the following example.

```
getCurrentObject() : Client;

begin
    return inheritMethod().Client;
end;
```

Base Implementation

```
getCurrentObject() : PersistentModel;

begin
    return self.myCurrentObject;
end;
```

getTA

The protected **getTA** method in the **BaseForm** class is used to return the transaction agent stored in the **myTA** property.

When reimplementing this method, the return type should be more-specific and the inherited method type cast with the specific **ModelTA** subclass.

```
getTA() : ClientTA;

begin
    return inheritMethod().ClientTA;
end;
```

Base Implementation

```
getTA() : ModelTA;

begin
    return self.myTA;
end;
```

getTAClass

The **BaseForm** class protected **getTAClass** method is used to return the type of transaction agent used by the form.

The reimplementation of this method should return a specific transaction agent (**ModelTA**) subclass, as shown in the following example.

```
getTAClass() : Class protected;

begin
    return ClientTA;
end;
```

Base Implementation

```
getTAClass() : Class protected;

begin
    return ModelTA;
end;
```

populateTAFromForm

The protected **populateTAFromForm** method in the **BaseForm** class returns a Boolean value. This method is used to populate the forms transaction agent with the form control values.

The following reimplementation example shows a **ClientTA** object being populated with form data.

```
populateTAFromForm( pClientTA : ClientTA input ) : Boolean protected;

vars
    addressTA : AddressTA;

begin
    if not inheritMethod( pClientTA ) then
        return false;
    endif;

    pClientTA.name := self.txtName.text.trimWhiteSpace();

    addressTA := pClientTA.myAddressTA;

    // create the addressTA if it does not exist
    if addressTA = null then
        create addressTA transient;
        addressTA.myModelTA := pClientTA;
    endif;

    addressTA.street := self.txtStreet.text.trimWhiteSpace();
    addressTA.city := self.txtCity.text.trimWhiteSpace();
    addressTA.country := self.txtCountry.text.trimWhiteSpace();
    addressTA.phone := self.txtPhone.text.trimWhiteSpace();
    addressTA.fax := self.txtFax.text.trimWhiteSpace();
    addressTA.email := self.txtEmail.text.trimWhiteSpace();
    addressTA.webSite := self.txtWebSite.text.trimWhiteSpace();

    return true;
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The transaction agent used by the form to populate the data. Reimplementation of this method should use a specific **ModelTA** subclass parameter value.

This method returns **true** if the operation was successful; otherwise it returns **false** if the operation was unsuccessful.

Base Implementation

```
populateTAFromForm( pTA : ModelTA input ) : Boolean protected;

begin
    pTA.myModelObject := self.getCurrentObject();

    return true;
end;
```

processAfterDelete

The protected **processAfterDelete** method in the **BaseForm** class performs tasks after a successful delete operation.

The following example shows how you can display a message box to the user or refresh a list.

```
processAfterDelete( pClientTA : ClientTA input ) protected;

begin
    inheritMethod( pClientTA );

    app.msgBox( 'Client ' & pClientTA.name & ' has been deleted', 'Success', MsgBox_OK_Only );
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The transaction agent used by the form

Base Implementation

```
processAfterDelete( pTA : ModelTA input ) protected;

begin

end;
```

processAfterSave

The protected **processAfterSave** method in the **BaseForm** class performs tasks after a successful save operation.

The following example shows how you can display a message box to the user or refresh a list.

```
processAfterSave( pClientTA : ClientTA ) updating, protected;

begin
    inheritMethod( pClientTA );

    app.msgBox( 'Client ' & pClientTA.name & ' has been saved', 'Success', MsgBox_OK_Only );
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pTA	ModelTA	The transaction agent used by the form

Base Implementation

```
processAfterSave( pTA : ModelTA ) updating, protected;

begin
    if self.myCurrentObject = null then
        self.myCurrentObject    := pTA.getModelObject();
    endif;

    self.displayObject( self.myCurrentObject );

end;
```

setContextObject

The protected **setContextObject** method in the **BaseForm** class is used to populate the **myCurrentObject** property with the **PersistentModel** specified in the **pTA** parameter.

The following example shows how you can populate the new form with a specified **Client** object to perform an update operation.

```
btnEdit_click( btn : Button input ) updating;

vars
    clientForm : EditClientForm;

begin
    app.mousePointer := Busy;

    if lstClients.listObject <> null then
        create clientForm;
        clientForm.setAddressableEntity( lstClients.listObject.Client.myAddress );
        clientForm.setContextObject( lstClients.listObject.Client );
        clientForm.showModal;
    endif;
epilog
    app.mousePointer := Idle;
end;
```

Parameter

The parameter for this method is listed in the following table.

Name	Type	Description
pContext	PersistentModel	The persistent object to be set as **myCurrentObject**

Base Implementation

```
setContextObject( pContext : PersistentModel ) updating;

begin
    self.myCurrentObject := pContext;
end;
```