Developing Agentry Apps

SAP Mobile Platform 3.0

Contents

Contents

Developing Agentry Apps

Use the Agentry Toolkit to develop metadata-driven applications.

Eclipse Preferences for the Agentry Editor Plug-In

With the release of version 5.2 of the Agentry Mobile Platform, additions have been made to the Eclipse preference pages by the Agentry Editor plug-in. These preferences affect the behavior of the Editor as a whole, or the appearance and behavior of the various views within the Agentry Perspective.

The following sections detail each of these new preference pages. To access these pages, select the menu item **Window | Preferences** in Eclipse. In the Preference screen, select the **Agentry** root node on the left side of the screen.

Compare Preferences
The Compare Preferences page displays the preferences for the Comparison View's appearance. the settings provided allow for the personalization of the colors used to denote the definitions when in different states within this view. There is also a preference for the number of previous revisions to display when the local project is connected to a share repository.

For each of the color preferences, the foreground and background color can be set by selecting the desired color from a color palette. The foreground color setting specifies the text color and the background color specifies the field behind the item. The following is a list of each of these preferences:

- **Selected Definitions:** This preference specifies the color scheme for the currently selected definition in the Comparison View.
- **Matching Definitions:** This preference specifies the color scheme for definitions when the two definitions match exactly in both projects being compared in the Comparison View.
- **Non-Matching Definitions:** This preference specifies the color scheme for definitions that do not match between the two projects being compared in the Comparison View.
- **Non-Matching (Unimportant) Definitions:** This preference specifies the color scheme for definitions that do not match between the two projects being compared in the Comparison View when the difference between the two is an unimportant difference, such as a difference in the descriptions of each.
- **Source-Only and Editor-Only Definitions:** This preference specifies the color scheme for definitions that exist in only one of the projects being compared in the Comparison View.
- **External Compare Tool:** This preference allows for the selection of the external comparison tool used to compare two text files from the Comparison View. By default the

tool Windiff is used. However, another tool can be used provided it can accept two arguments, each representing the files to be compared. This preference should be set to the full path and file name of the executable file of the comparison tool.

Compression Settings

The Compression Settings page displays the preference settings for compressing Agentry definition files during export, export differences, and publish operations. The default is to compress the definition files generated during these operations.

- **Compress Export:** This preference, when selected, results in the creation of compressed export files when the Agentry application project is exported. The file extension for the export file is set to `.agxz`. If the file is not compressed the resulting export file has a file extension of `.agx`.
- **Compress Export Differences:** This preference, when selected, results in the creation of compressed export files when the Agentry application project is exported via an Export Differences operation. The file extension for the export file is set to `.agxz`. If the file is not compressed the resulting export file has a file extension of `.agx`.
- **Compress Publishes:** This preference, when selected, results in the published definitions, stored on the Agentry Server, being compressed. The Server recognizes this compression and automatically decompresses it when loading the definitions. if this option is not selected, definitions are not compressed when published to the Server and the Server will recognize this as well. No change is needed on the Agentry Server to reflect the compression setting for publish.

New Definitions

The New Definitions preference page contains preferences related to the creation of new definitions within the application project. This includes setting reminders on definitions, default names for certain definition types, and the default selected data type for complex table fields. These settings only specify default values initially selected and all values can be changed when actually creating the definitions referenced here. The following list describes each of the preferences found on this page:

- **New Definition Reminders:** Each definition type within the project supports reminders, or text values displayed as informational messages in the Problems View. By default reminders are set manually by the developer. Using this preference setting it is possible to set a default reminder message for all new definitions created in the project, including a default reminder message.
- **Transaction Naming:** The default name of all new transactions can be specified to be either Noun-Verb (default) or Verb-Noun. Regardless of the default set here, names can always be changed when adding a new transaction definition.
- **Rule Naming:** The default name of all new rule definitions can be specified to be either Name-Usage (default) or Usage-Name. Usage refers to the attribute type referencing the rule, e.g. Enable Rule, Initial Value Rule, etc. Regardless of the default set here, names can always be changed when adding a new rule definition.

- **Complex Table Field Type:** This preference allows you to specify the default data type selected when creating a new complex table field definition. This is only the default selection when a new field is first created and can be changed during the creation process to a different data type if needed.

Perspectives

This section specifies the behavior of Eclipse related to the Agentry perspectives when opening an Agentry application project when there is no Agentry perspective currently open. These preferences specify whether to always open an Agentry perspective, never open one, or to prompt you to choose whether or not to open the perspective. Included in these preferences is a list of the available Agentry perspectives where the specific one to open by default is selected.

Project Explorer

The Project Explorer preferences page contains the preferences that affect the appearance and behavior of the Project Explorer View, as well as the Properties View interaction with it. The following list describes each of the preferences found on this page:

- **Link Properties View breadcrumb navigations with the Project Explorer View:** When this preference is set to true, navigating in the Properties View using the navigation buttons for back and forward will automatically select the definitions in the Project Explorer View.
- **Link Properties View parent navigations with the Project Explorer View:** When this preference is set to true, using the parent arrow in the Properties View to navigate to the parent or ancestor definition of the one currently displayed will automatically select that definition in the Project Explorer.
- **Link Properties View hyperlink navigations with the Project Explorer View:** When this preference is set to true, clicking a hyperlink label in the Properties View to display the referenced definition will automatically select that definition in the Project Explorer View.
- **Link Trash Bin definition restores with the Project Explorer View:** When this preference is set to true, the restoration of a definition from the Trash Bin View will automatically select that definition in the Project Explorer View.
- **Link Visual Screen Editor selections with the Project Explorer View:** When this preference is set to true, the selection of a screen control (detail screen field, list column, button, etc.) in the Layout View of a screen definition will result in that same definition being selected in the Project Explorer View.
- **Sort Order:** This preference specifies how the definition type nodes in the Project Explorer View should be sorted. The options are Traditional, which sorts the type nodes as in releases of Agentry prior to 5.2; and Ascending, which sorts the type nodes in alphabetical order.

Tagging Configuration

The Tagging Configuration preferences page contains the preferences for the definition tagging behavior of the project. It is also possible to add new tags to the project within this

preference page. Note the preferences on this page are specific to the currently open Agentry application project. The following list describes the preferences displayed on this page:

- **Show tags bar in the Properties View:** This option shows or hides the tags bar in the Properties View. If the current open Agentry application project is connected to a share, this preference is always true and cannot be changed.
- **Auto-Tagging (Public):** Within this section each of the public tags for the project are listed, represented as a button. One or more of these buttons can be selected to specify that tag should be applied to any definition changed, added, or deleted in the current project. Clicking the tag configuration button within this section will display the tags dialog where additional tags can be added to the project.

Team Configuration
The Team Configuration preference page contains preferences for the team configuration behavior. These preferences are specific to each Agentry application project. These preferences are not available when the current project is not connected to a share. The following list describes these preferences:

- **Auto-Tagging (Private):** Private auto-tagging describes the behavior where the Editor automatically applies a private tag to any definition when it is modified, added, or deleted in the current project. This tag is used in comparisons with the share revisions, and is also displayed in certain operations to allow for the selection of definitions with this tag. The preference setting on the Team Configuration preference page allows for a customized value to be used as the name of this tag. By default, if this preference is blank, the developer's Windows user ID is the tag name.
- **External File Handling:** This preference specifies whether it should be required that the Agentry Development Server is specified before any share operations are executed that involve definitions with an external script normally stored on the Development Server. As examples, when this preference is true, a new project cannot be imported from a share without specifying the Development Server for the new project; a SQL step cannot be updated from the share unless the Development Server has been specified.
- **Share Revisions Menu Count:** This preference specifies the maximum number of revisions to display at one time in the revision menu of the Comparison View. This preference setting only affects the Comparison View's behavior when the open Agentry application project is connected to a share repository.

Agentry Editor and Eclipse Platform Overview

The primary development tool for an Agentry mobile application is the Agentry Editor. The Editor is provided as a plug-in component to the Eclipse development platform. Eclipse is an open source project freely available for download from the project's website, and can also be installed with the the Agentry Editor plug-in. The Agentry Editor is a 4th generation language (4GL) development tool providing point-and-click development of applications. Much of the behavior of the Editor is dictated by the overall behavior of the Eclipse platform.

The following is general information covering the Eclipse platform and the Agentry Perspective within Eclipse. Developers unfamiliar with Eclipse will find this information useful as a starting point. It is also recommended that developers review the help information provided with Eclipse by selecting **Help | Help Contents** and reviewing the *Workbench User Guide* as well as other information provided in the help content.

Specifics on procedures, navigation, and other topics related specifically to the Agentry Editor are covered in detail in subsequent sections.

The Agentry Editor Perspective and Views

The Agentry Editor plug-in includes an Eclipse perspective consisting of several Views. A view is a pane or tab within Eclipse that presents information about and access to various development components. Perspectives are a collection of multiple views. Opening a perspective within Eclipse opens all of that perspective's views.

The views within the Agentry perspective provided by the Agentry Editor plug-in are the primary views for application development within Agentry, but they are not the only views that will be used in most projects. Other views and other perspectives within Eclipse provide features and functionality that match the tasks and development work performed for a given application project. The ability to use the core views within the Agentry perspective in conjunction with views from other tools and plug-ins within Eclipse is one of the main advantages to the Agentry Editor plug-in architecture.

A common perspective used during the development of a mobile application is the Java perspective. This perspective is provided with Eclipse and includes various views and other tools to support Java development work. When working with a mobile application that will synchronize data with a back end system using Java, this perspective is used regularly for development, implementation and configuration tasks.

Agentry Application Project

Development work related to a mobile application's Agentry definitions is organized within the Agentry Editor in an Agentry Application Project. This project contains the definitions that are the mobile application, structured according to the Agentry Application Hierarchy. Within the Agentry Editor perspective this project is presented in the Project Explorer View according to the application hierarchy. Within this view the developer navigates through the application definitions. The project itself is stored as an Agentry application project file with the extension `.apj`.

Eclipse Workspace

The Eclipse platform organizes development projects, interface settings, and most other aspects of the environment into workspaces. This includes the Agentry application project created and managed within the Agentry Editor plug-in. Other related projects are also stored within the same workspace. In most cases it is recommended that a workspace exists for each mobile application. Within the workspace for a given application there should then reside the

Agentry application project, as well as other related items such as Java or Ant projects, where necessary.

An alternative is to create a single workspace for multiple mobile applications that are developed for the same back end system and that have significant overlap in back end processing requirements and methodology. In environments where multiple applications are developed for the same back end system, developers may structure the back end synchronization components into layers that include those common to all mobile applications for the same back end, and those specific to a given mobile application. In such environments it may make sense to have a single workspace containing all components and projects, so that the individual application projects can share the common synchronization resources, as well as contain the individual synchronization projects for each application. This would also include one Agentry application project for each mobile application.

In general, the structure of the workspace is up to the developer or development team, and the best approach for all involved can be used. There is no hard and fast rule concerning the proper structure for a workspace within Eclipse as it relates to the mobile applications developed or modified in the Agentry Editor.

Other information concerning the Eclipse interface is also stored within the workspace. These include the items set via Eclipse's preference settings. These settings can affect the behavior of numerous different views within Eclipse. Changes made to the preferences will be stored in the workspace. This means each workspace within Eclipse can have its own preferences tailored to the needs of that workspace, and switching from one to another will load the new workspace's settings.

Third Party Views and Tools
In addition to the functionality contained within the Agentry Editor plug-in, there are additional tools and views provided by third parties as contributions to the Eclipse Platform. Many of these may be useful in different development environments that involve Agentry application projects. Certain contributions are expected and used by the Agentry Editor, specifically the Data Tools Platform project and the Java Perspective. The Java Perspective is provided with all implementations of Eclipse and is used by the Agentry Editor when the application project includes synchronization with a Java Virtual Machine system connection.

The Data Tools Platform is included in Eclipse as well and is used when the application project includes synchronization with a SQL Database system connection. It is through this package that the Agentry Connector Studio provides its functionality.

The Agentry Perspective in Eclipse

The various views within the Agentry Perspective in Eclipse work in conjunction with one another. The navigation of an application project involves selecting a definition in one perspective and then viewing it in one or more of the other perspectives.

The following are the views found in the Agentry Perspectives provided with the Agentry Editor plug-in for Eclipse:

- Project Explorer
- Properties
- Diagram Views
- Dependency
- Trash Bin
- Problems

Project Explorer View

The Project Explorer View is a part of the core Eclipse platform that is used by the Agentry Editor plug-in. In general, this view displays projects saved within the current Eclipse workspace. These projects are displayed in a tree control, with each root node representing one of the projects. For the Agentry Editor, the Agentry application projects within the current workspace are listed here as well.

An Agentry project is opened or closed within this view. An Agentry project can be navigated in the project explorer view only when it is open, and only one Agentry application project can be open at a given time. Opening one project will close any Agentry project that is currently opened.

The definitions within an Agentry project are displayed in a tree control that matches the Agentry Application Hierarchy. For the open project, any definition listed in the tree control can be selected. This displays the definition in the Properties View, lists its dependency items in the Dependency View, and may display an diagram for the definition in the Diagram View. This last depends on the type of definition selected, as not all definitions have a corresponding diagram.

The following is an example of the Project Explorer View with a sample Agentry application project open within it:
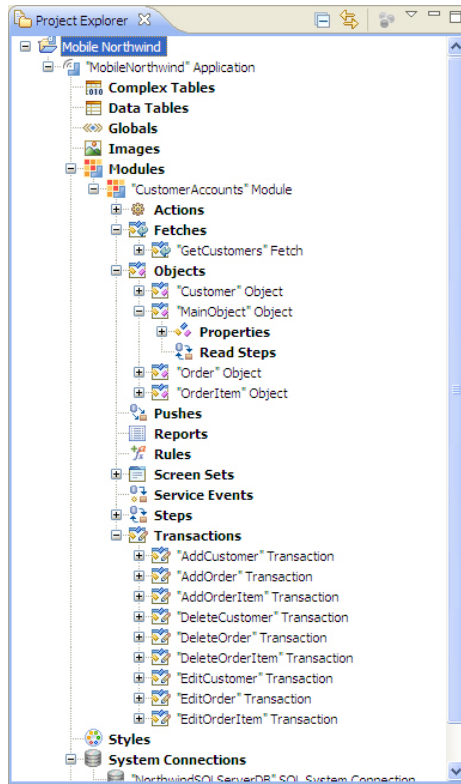
Each of the nodes in this view represents a definition within the application project. Bold nodes represent a set of definitions of the same type, contained within a parent definition. Selecting a bold node will display a list of all definitions of that type within the same parent definition. Selecting a definition node will display information about that definition in the other views within the Agentry Perspective.

Right-clicking a node within the view displays a context menu with options to affect the application project or the selected definition. Which specific options are available depends on which node is selected, or which definition that node represents. Items that may be displayed include displaying the definition in its diagram view, adding a new definition of the selected type, copying or deleting the selected definition, or setting a reminder for the selected definition.

Properties View

The Properties View displays the attributes and child definitions for the currently selected definition within the project. The definition's attributes are edited within this view, and its child definitions can be modified as well, including adding new definitions, deleting a child definition, or selecting a child definition. Selecting a child definition displays it in the Properties View.

Within the Properties view there can be one or more tabs, listed to the left of the view. Each tab displays a set of attributes for the definition, or a list of child definitions. When a tab that displays attributes is selected, there is a single row of toolbar buttons displayed for the view. These buttons pertain to the definition, or to navigation within the application project based on the currently displayed definition.

When the selected tab within the Properties View displays a list of child definitions, a second row of toolbar buttons is displayed that allow for the modification of that list and the child definitions each item in the list represents. This includes adding, editing, deleting, copying, navigation, and other similar options.

The following is an example of the Properties View for a list screen definition, with the List Screen Definition tab selected:
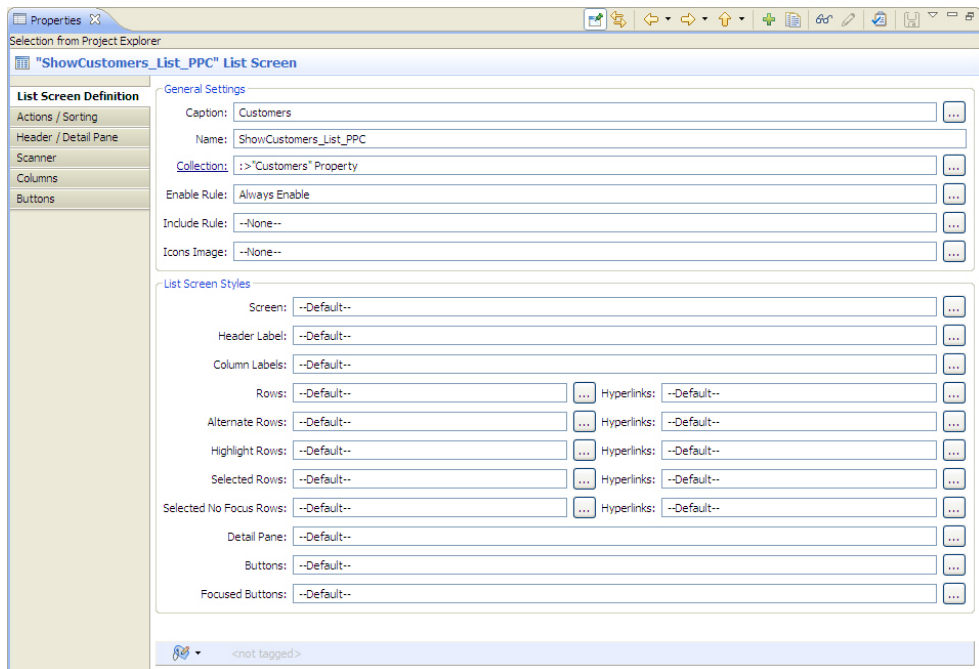
As noted, at the top of this view are the toolbar buttons for working with the definition and its currently displayed attributes. The following is a list of these buttons and their purpose:

Button	Button Icon	Description
Pin Properties View		Pins the Properties View so that it always displays the current definition, regardless of any selections made in any other view. Note that this prevents navigating away from this definition until the button is deselected.

Button	Button Icon	Description
Link With Project Explorer		Selects and highlights the node for the currently displayed definition in the Project Explorer View.
Navigation Buttons		These buttons provide browser-like navigation of the application project, with back, next, and parent navigation possible.
Add Definition		This button adds a definition of the same type, and to the same parent definition, as the one currently displayed in the Properties View; e.g., if currently viewing a screen, this will add a new screen to the same parent screen set.
Copy Definition		This button creates a new definition that is a copy of the currently displayed definition, including all descendent definitions. The new definition is added to the same parent as the original definition.
Diagram View		This button opens the Diagram View for the currently selected definition. This button is disabled for definitions that do not have associated diagram views.
Edit Definition		This button opens the editor specific to the currently displayed definition in the Properties View. Most definitions do not have a type-specific editor, and their attributes are edited within the Properties View. This button is disabled in these cases. The Rule definition is the only definition for which this button is enabled, displaying the Rule Editor when selected.
Set Reminder		This button sets a reminder for the definition displayed in the Properties View. Reminders are listed in the Tasks View within Eclipse, and also result in informational messages in the Problems View during publish and check on publish operations. All definitions support reminders. Setting a reminder does not affect application behavior and serves only to provide notes to developers within the application project on tasks remaining to be accomplished.
Save		This button saves any changes made to the attribute settings currently displayed in the Properties View. Note that the main Eclipse toolbar also contains a save button. Only the save button in the Properties View saves changes made to the definition's attributes. The Eclipse toolbar button is disabled when the Properties View has the current focus.

Selecting a tab listing child definitions for the current definition in the Properties View displays a second toolbar of buttons that affect the list of child definitions. Following is an example of the Properties View for a list screen with the Columns tab selected, which displays a list of all columns that are child definitions to the list screen:
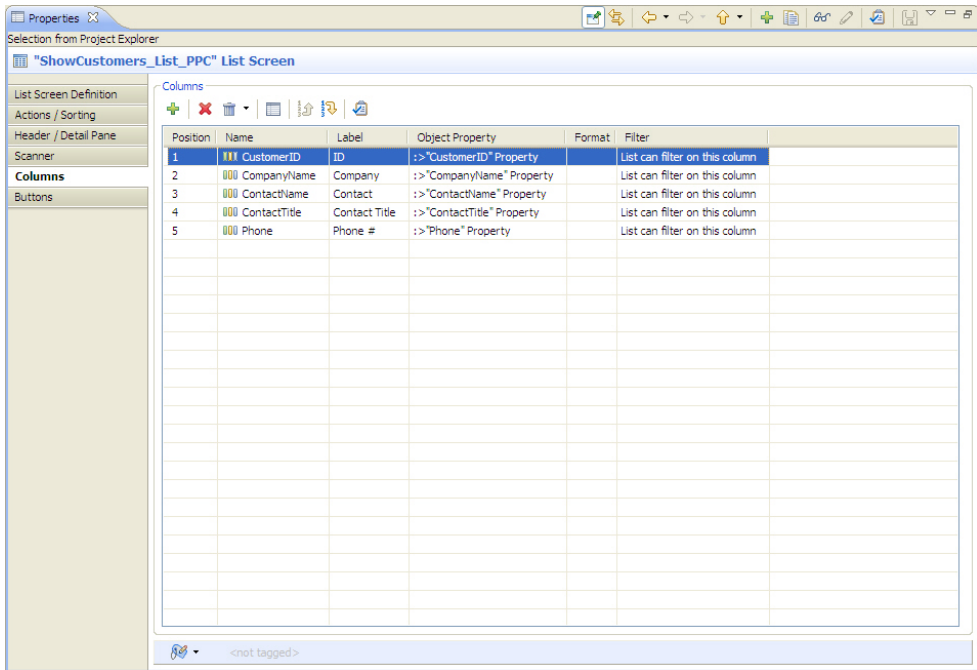
Note the row of buttons directly above the list of child definitions. Following is a description of each of these buttons:

Button	Button Icon	Description
Add Definition		This button adds a new definition of the type currently listed as a child definition to the one currently displayed in the Properties View.
Delete Definition		This button deletes the currently selected child definition in the list.
Trash Bin		This button displays the all definitions that have been previously deleted from the parent definition of the type currently listed; e.g. all columns deleted from the list screen. From this list a deleted child definition can be recovered from the trash bin and placed back in the parent definition.
Display Definition		This button displays the currently selected child definition in the Properties View.

Button	Button Icon	Description
Change Positions		These buttons move the currently selected definition in the list up or down one position in relation to the other child definitions listed. These buttons are only available in child definition lists where the position of the child definitions affect application behavior; e.g., the position of list screen columns in the list dictates the initial display order of those columns on the client at run time, whereas the order in which object properties are listed has no effect on application behavior.
Set Reminder		This button sets a reminder for the currently selected child definition. Reminders are listed in the Tasks View within Eclipse, and also result in informational messages in the Problems View during publish and check on publish operations. All definitions support reminders. Setting a reminder does not affect application behavior and serves only to provide notes to developers within the application project on tasks remaining to be accomplished.

Diagram View

The Diagram View provides a graphical representation of a definition. The layout, organization and appearance of this view is specific to the type of definition being displayed. Many of these are self-explanatory in nature. Not all definitions are displayed in the diagram view, as there is no benefit in doing so. The general rule of thumb is that definitions without child definitions are not displayed in the diagram view. The exception to this is the Rule definition type. Rules do not have child definitions. However they are displayed in the Diagram View, with the structure of the rule logic displayed in a read-only format.

For other definition types, the items in the Diagram View are organized according to the definition type selected. For many definition types this is a graphical representation of the parent-child relationship between the selected definitions and any child definitions. For others, certain organizational information is provided. It is this last group that is explained in more detail here.

Diagram View: Module Transactions
The Diagram View for a module's transactions displays all transactions within a given module. This Diagram View is displayed when the bold Transactions node in the Project Explorer View is double-clicked, or when it is right-clicked and the View Transactions Diagram menu item is selected. The Diagram View displays a module's transactions in a manner organized according to their back end processing definitions, i.e. the step usage definitions within the transaction.
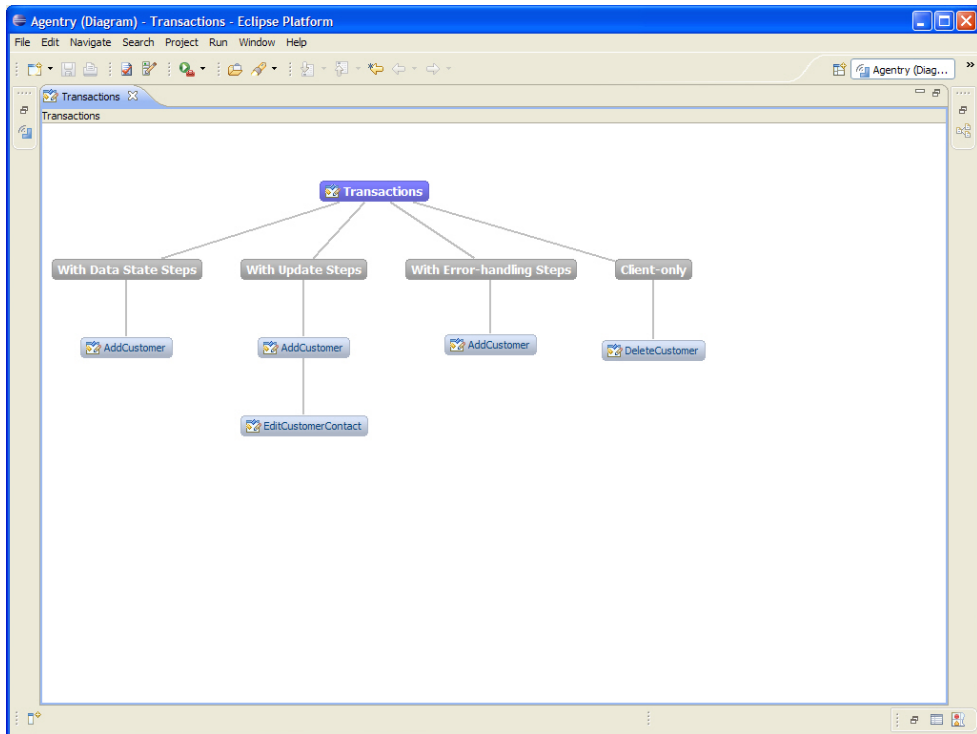
In this simple example there are three transactions displayed. Note that the AddCustomer transaction is listed three times, as it contains server data state steps, server update steps, and error handling steps. EditCustomerContact contains only update steps and is therefore listed only under that node in the diagram. Finally, DeleteCustomer does not have either data state or update steps. Therefore, it is listed under the Client-only node meaning it has no server-side processing.

Diagram View: Actions

The Diagram View for an action definition displays that action in a flow chart view representing the execution flow of the action. This can be a useful view for complex actions, especially those with subaction steps and looping behaviors.

Following is an example of an action with two subaction steps. The first subaction step is defined to execute once and the second is defined to execute iteratively, also known as a looping subaction step:
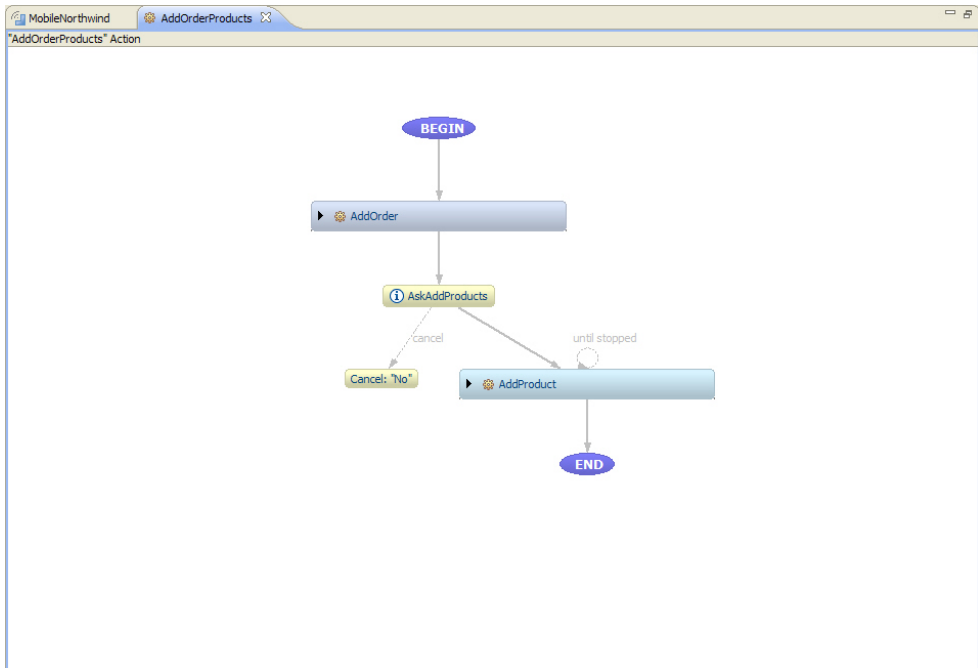
The first subaction step is AddOrder, which executes a second action for an Add transaction. This step is defined to execute once. It is followed by a message step named AskAddProducts, which prompts the user to add products to the order created by the AddOrder step before it. If the user clicks the **No** button in the message, the Action ends execution, which is represented by the node "Cancel: 'No'" in the above example.

If the user clicks the **Yes** button in the message prompt the subaction step AddProduct is executed. As shown in the Diagram View, this step is defined to loop until stopped, meaning the users indicate they are finished. Once this loop ends, the action completes, as represented by the node End in the Diagram View.

The two nodes for the subaction steps can each be expanded to display the steps of the actions they execute. Following is an example of this same action with the subaction steps expanded:
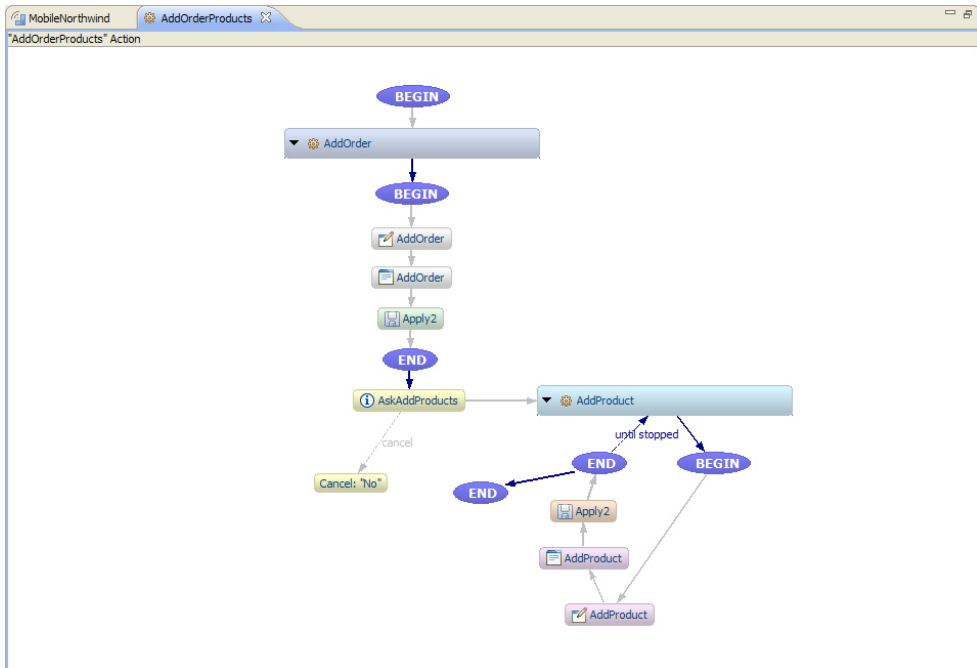
With these subaction steps expanded, the steps of each subaction are displayed. The AddOrder action contains the AddOrder transaction step that displays the transaction in the AddOrder screen set. This is then followed by the Apply step.

Next the Add Products prompt is displayed, which introduces a conditional execution to the action's execution flow. A positive response from the user executes the AddProduct subaction step. Since this step is defined to loop, the steps are presented in the diagram view to represent this behavior. The loop condition is displayed in the flow of this action, with the text "until stopped."

A negative, or "Cancel" response from the user to the Add Products prompt ends the action's execution.

Diagram View: Complex Tables

When displaying a complex table in the Diagram View, the indexes and fields are each displayed as child definitions to the complex table. In addition, the index nodes have one or more child nodes of their own. These child nodes are, first, the field for which the index was created and, second, if it is a child index the field or fields for which the parent indexes were created.
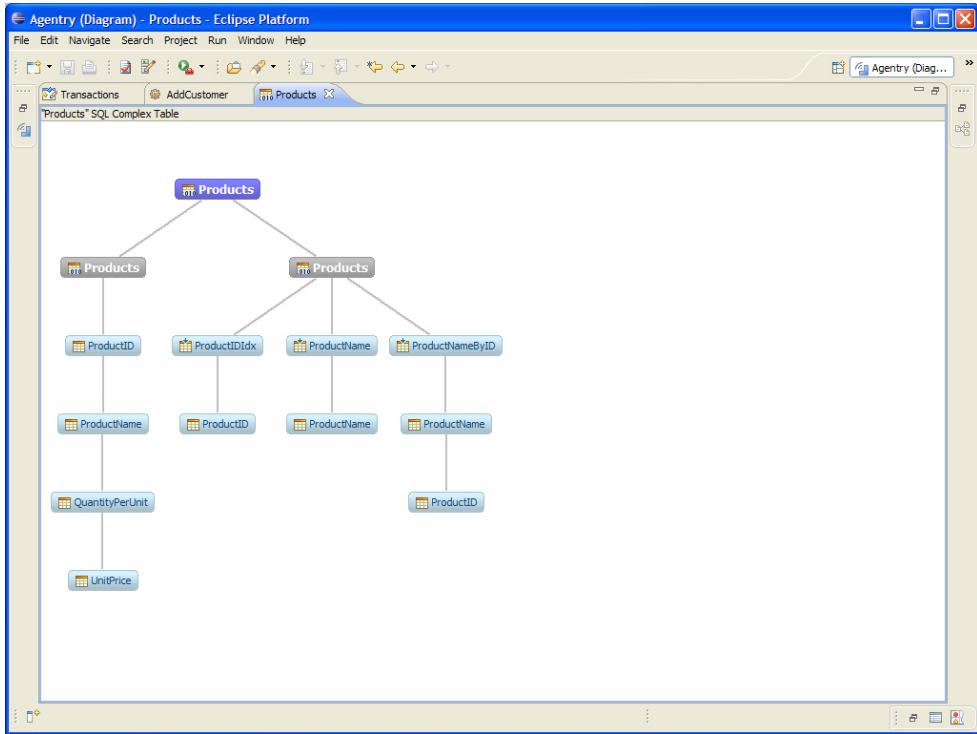
In this example, the index ProductNameByID is a child index to the ProductIDIdx index. This is represented by the Diagram View.

Dependency View

The Dependency View displays the definitions within the application project that reference the currently selected definition in the Project Explorer. This is useful information when editing or removing the selected definition, as you can readily see the definitions that can or will be affected by the change.

The list of definitions in the Dependency View also includes the hierarchy information. A definition can be expanded in the Dependency View to display its parent definition. This information provides the path up to the module level. Any definition in the Dependency View can be selected and displayed it in the Properties View.

The following is an example of the Dependency View for an object definition:
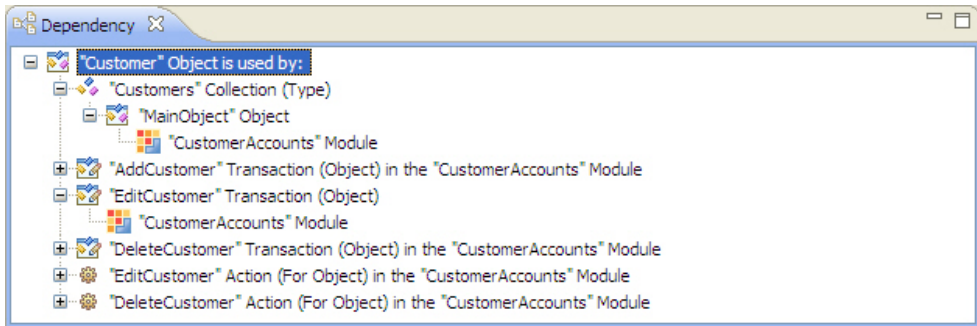
Trash Bin View

The Trash bin View displays a list of all definitions that have been deleted from the Agentry application project. This view provides a holding pen of deleted definitions to allow for their recovery in the event the deletion was in error.

Following is an example of the Trash Bin View:
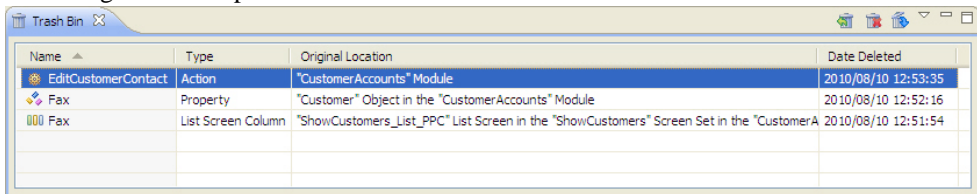
Each of the items in this list represent a definition that has been deleted from the Agentry application project. Within this view a definition can be selected and either restored to its previous location within the project, or permanently deleted from the project. Once a definition is removed from the Trash Bin it cannot be recovered.

Problems View

The Problems View provides informational, warning and error messages related to the current Agentry application project. Following are the three classes of messages displayed:

- **Informational:** These messages are informational in nature and are innocuous in that they do not indicate any issue that may have a detrimental affect on the application project. Examples include information concerning the date and time of the application project's last publish, and a list of the definitions for which reminders have been set.
- **Warning:** These messages indicate a potential issue with the application project. Items resulting in a warning will not prevent an application from being published but may result in issues arising at run-time.
- **Error:** These messages indicate a validity issue with the application project and will prevent the project from being published. Such issues must be rectified prior to publishing the project.

The Problems View is always updated during a publish or a check on publish operation. Additionally, certain items are displayed in the Problems View as soon as a change occurs that indicates an issue.

As an example, if an action definition is currently defined and is used by a control or other definitions, and that action is then deleted from the project, the Problems View will immediately display a list of warning messages indicating those definitions. Following is an example of this messaging for an action that was executed by both a sub-action step in another action and a button on a screen. The action was subsequently deleted from the project:
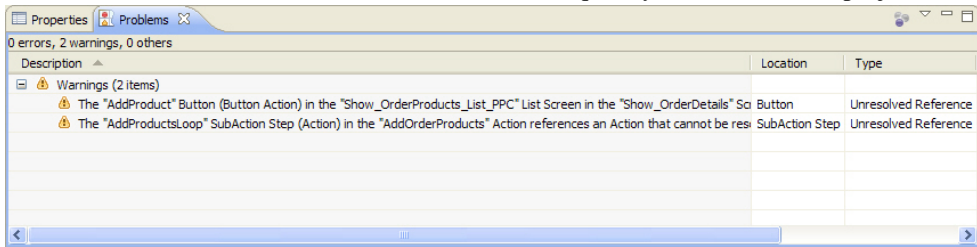
This view is interactive. Double-clicking an item in this list displays in the Properties View the definition with the reported issue so that changes can be made to the definition. Once such changes are made the corresponding message will be removed from the Problems View.

Agentry Views Outside of the Agentry Perspective

The Agentry Editor includes views displayed in the Agentry Perspective as well as views displayed when certain operations are performed or displayed on demand by the developer. The views within Eclipse which may be displayed by the Agentry Editor plug-in, but which are not a part of the default Agentry Perspective, are explained briefly. The information provided here is introductory in nature, providing an overview of the behavior of these views. Details on each are provided in the content discussing the operations to which the views pertain.

Comparison Editor

The Comparison Editor is displayed during import, export, and share repository operations. This view displays the current Agentry application project and a second source project side by side. Definitions within each are compared and those with differences between the two projects, or those that exist in one project but not the other, are highlighted.

For import operations the Comparison Editor is used to select the definitions in the import source to be imported into the current Agentry project. For export operations this view is displayed when the export differences tasks is performed, displaying the differences between the application project and the comparison source, indicating what definitions are to be imported based on differences between the two. For share repository operations, the comparison view is displayed during updates from the repository to the current application project, indicating the differences between the two and the definitions affected by the update.

History View

The History View is displayed when the developer selects the **Team | Show History** menu item in the menu for the Agentry application project. This view is solely used for the team configuration functionality when the current application project is connected to a share repository. This view displays a list of all revisions within the repository. The revision from which the current application project was last updated is highlighted in this list.

The Data Tools Platform: SQL Development Tools

The Data Tools Platform is an open source project for the Eclipse platform that encompasses three separate but related Eclipse projects. The Agentry Editor plug-in for Eclipse make use of two of these projects: the Connectivity Project, and the SQL Development Tools project. Each of these projects provides perspectives and views to the Eclipse platform that can be useful to Agentry development, implementation, or configuration projects involving a SQL Database back end system.

Though these tools are provided by contributors to the Eclipse project and not Syclo, they are used by Syclo's Agentry Editor Eclipse plug-in to facilitate and support common development tasks. Information and instructions for working with and configuring these various tools as it relates to mobile application development and configuration is provided in the document set for the SAP® Mobile Platform. For extensive information on the Data Tools Platform project and its child projects, see the Eclipse help site at:

```
http://help.eclipse.org/galileo/index.jsp
```

Of note for the uninitiated developer is the *Data Tools Platform User Guide*, which is available in electronic form on the Eclipse help site listed above.

Connectivity Project and the Agentry Connector Studio
From the *Data Tools Platform User Guide*:

> The connection-management functionality provided
> in the Connectivity project includes...components
> for defining, connecting to, and working with data
> sources.

One of the features provided by the Agentry Editor Eclipse plug-in is the Agentry Connector Studio. Using this tool an Object Wizard is displayed that allows for the definition of various data-related definitions within a module, with the attributes of those definitions set in part based on available information about the back end system with which the mobile application will synchronize data. The Connector Studio itself can be used with SQL Database, Java Virtual Machine, or HTTP-XML system connections. When working with a SQL Database system connection, the connector studio requires the use of the Connectivity Project tools within the Data Tools Platform.

Within this set of tools, there are specific items used by or required for the Agentry Connector Studio functionality when working with a database system. These include Connection Profiles, Driver Definitions and the related Driver Management Framework, and the Data Source Explorer View. To use the Agentry Connector Studio to create the module data definitions, a Connection Profile must exist for a connection to the database from which the data definitions will be defined. A Connection Profile makes this connection using a Driver Definition that encapsulate the method in which the connection to the profile is made. The Connection Profile then represents the connection to the specific database instance or database server. Connection Profiles created in Eclipse are exposed to the developer in the Data Source Explorer View.

The Agentry Connector Studio can then be accessed from within the Data Source Explorer View, when using the Connector Studio to access a database system. Within the Data Source Explorer, the developer navigates to the specific database table for which an object definition is to be created. Right-clicking on this table displays a context menu, which includes the menu item **Agentry Connector Studio**. Selecting this item will display the Connector Studio Object Wizard, which will walk the developer through the definition of the object and its properties, based on the schema information provided for the database table. It also provides the option for defining transactions for the new object type and SQL step definitions, including basic SQL statements based on the database table. These SQL statements are intended for use by the transaction server update steps, and the object read steps or fetch server exchange steps.

Instructions for creating a Connection Profile and Driver Definition within the Connectivity Project tools can be found in the *Agentry Implementation and Administration Guide*.

SQL Steps and SQL Synchronization Definitions: The SQL Editor View

The Agentry Editor plug-in makes use of the SQL Development Tools, another component of the Data Tools Platform. Specifically the SQL Editor View is the view used to display and edit SQL statements with in the Agentry application project. The SQL Editor is a view that supports the authoring, editing, and testing of a SQL statement. When a definition that includes a SQL statement for synchronization is defined, the default behavior of Eclipse is to display that statement in the SQL Editor. This editor provides several configurable aides in authoring well-formed SQL logic, including helpers such as adding quotes around values that require them automatically, indentation/tab and other "pretty print" functionality, and other similar behaviors, all of which have default behaviors that can be configured and customized to the needs of the developer and the project. In addition, since the SQL Editor is a part of the SQL Development Tools project, and since this project is a part of the larger Data Tools Platform project, the SQL Editor can make use of the features provided by its sibling Connectivity project.

Specifically, if a connection profile has been created within the Connectivity Project tools, the SQL Editor can use that connection profile to open a connection to the database and execute the query within the SQL Editor against that database. This is a useful testing feature that can help developers verify the validity of their SQL statements within the application.

The Java Perspective

The Java Perspective provided with Eclipse is the main interface to the JDT project for Eclipse. This perspective, as well as other tools within the JDT, are used in development of an Agentry application that connects with a Java Virtual Machine system connection to synchronize data.

The Java Perspective is opened by default when Eclipse is started. The views for this perspective include the Package explorer. Within this view are the Java packages for the current project. Also listed in this tab are any Agentry application projects created and saved within the current Eclipse workspace. Selecting and opening this project in this view will open the Agentry perspective within Eclipse, displaying that project.

The Java components to the mobile application should be organized within a Java project. Organized within this same project the packages of the Java Interface through which data will be synchronized, and the Agentry Java API packages provided in the `Agentry-v5.jar` file should also be included.

The Java components of the Agentry application project are created and maintained using the tools and wizards provided with the JDT project. These include the Java Perspective, as well as other tools within the JDT to build and maintain the Java logic. When a definition is created in the Agentry Perspective that contains a Java synchronization component, the option as to the source of the logic for that component is presented. Depending on the selected source, the Java class wizard is displayed, which allows for the selection of the package and parent class for the new class, as well as the package and project placement of the class.

Information covering the Java Perspective, as well as the JDT of which it is a member, provided in the Agentry document set is limited to those areas of functionality in which the two directly relate. This includes guidelines for creating Java projects and packages for a mobile application development project, as well as use of the Java class wizard and some other tools within the JDT. The JDT itself is a robust Java IDE with far more features and functionality than will be covered in this document set. Extensive information can be found on the JDT at the web address:

```
http://help.eclipse.org/galileo/index.jsp
```

The item of note at the above URL for the uninitiated JDT user or Eclipse developer is the *Java Development User Guide*.

Searching Agentry Application Projects

The Agentry Editor to Eclipse includes search functionality. This functionality is supported through the standard search wizard within Eclipse, with the addition of an Agentry Search tab displayed in the Eclipse search wizard.

Within the Agentry Search tab there are numerous options available specific to an Agentry application project. Project definitions can be searched by tags, text contained within the

definition, and searches can be performed within the entire project or within the currently selected definition and its descendent definitions only.

To perform a search of the Agentry project, select the Eclipse menu item Search | Search... This displays the Search wizard, with the Agentry Search tab selected:
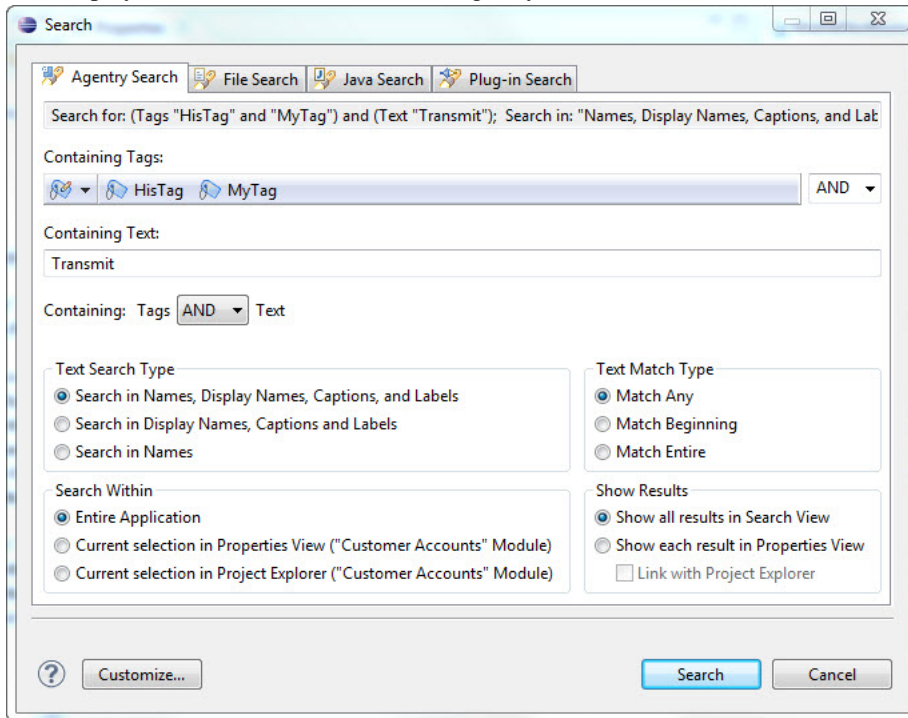
The following items can be selected to perform a search of the project:

- **Containing Tags:** One or more tags from the current project can be selected here, along with the option of definitions with no tag applied. The drop down list at the end of the list of selected tags allows for the selection of an AND or an OR search. AND requires the definition to contain all selected tags, OR returns definitions with any of the selected tags.
- **Containing Text:** Search text contained in the name, display name, caption or label of the definition. This can be further refined with the Text Search Type settings. Selection of the Containing: Tags AND/OR Text option will also impact the search results. AND will require the definition to contain both the selected tags and entered text. OR will return definitions with either the tags or the text or both.
- **Text Search Type:** Allows for the restriction of the search text to be found in one of the three groups of text values of the definitions.
- **Search Within:** The options here allow for the scope of the search to be set. This can include the entire project, the selected definition in the Project Explorer View and its descendents, or the definition displayed in the Properties View and it's descendents.

- **Text Match Type:** These options allow for search behaviors such as matching the whole word only, case sensitivity, and similar options.
- **Show Results:** The option to display all matching definitions in the Search Results View, or to display each definition in turn within the Properties View is set in this section.

Once the search options and criteria have been selected, click the **[Search]** button to search the Agentry application project. Matching definitions will be displayed according to the Search Results settings selected in the wizard. Only the definitions of the currently opened Agentry project are searched.

Agentry Application Projects: Creating, Managing, and Publishing

The Agentry Editor provides several features for creating, managing, and publishing the Agentry application project. An application project can be added to the current Eclipse workspace via an import or a new project created from scratch. Importing can be performed using one of a number of different sources as discussed in detail in the sections on importing. Creating a new project from scratch is performed using the New Application Wizard within the Agentry Perspective in Eclipse.

Project management features include the ability to export definitions from the project to a single file, as well as support for multiple developers through a common repository, or "Team Development," a concept new to the Agentry 5.2 release. Exports can be performed for the entire project, manually selected definitions within the project, or automatically selected definitions based on differences between two different versions of the same project. Additional features include the ability to compare two projects or a project and export file and to selectively import components from a source to the current project.

Publishing is the task performed when modifications within the application project are in a stable state and can then be either tested or deployed to end users. The process of publishing can include development publishes, production publishes to a single Agentry Server instance, or production publishes to a cluster of Agentry Servers. The process of publishing to production for deployment can be performed directly to the Agentry Server(s), or, alternately, may involve an intermediary Agentry Production Server. This depends on the network environment and policies in the implementation environment and is discussed in detail in the sections on publishing to production.

Creating a New Agentry Application Project

Prerequisites

The following items must be addressed prior to performing this procedure:

- The Eclipse environment including the Agentry Editor must be installed.

- The Agentry Perspective must be open within Eclipse.
- The Eclipse workspace to which the new Agentry application project will be added must be open.
- It is recommended, though not required, that the Agentry Development Server is installed to which development publishes will be made.

Task

The following procedure provides instructions on creating a new Agentry application project. Perform this procedure when a new project is needed and that contains no existing business logic. If creating a new project within the current Eclipse workspace based on an existing Agentry application project, export file, or published version residing on an Agentry Server, see the information on importing Agentry application projects.

1. Start the New Application Wizard for Agentry application projects by selecting the menu item **File | New | Project...**

This displays the Eclipse new project dialog where the type of project to create is selected:
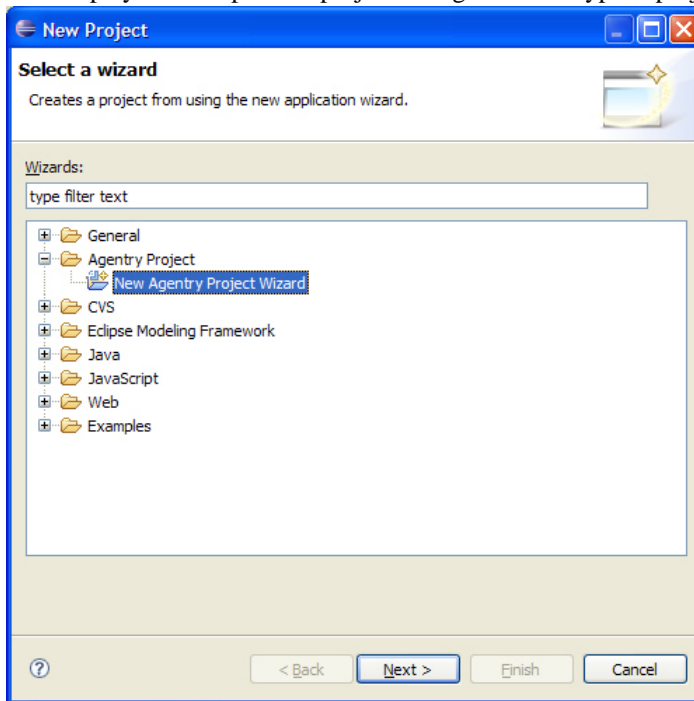
2. Select the item **Agentry Project | New Agentry Project** in the tree control displayed. Click the **[Next >]** button.

The first screen of the New Agentry Project wizard is displayed:
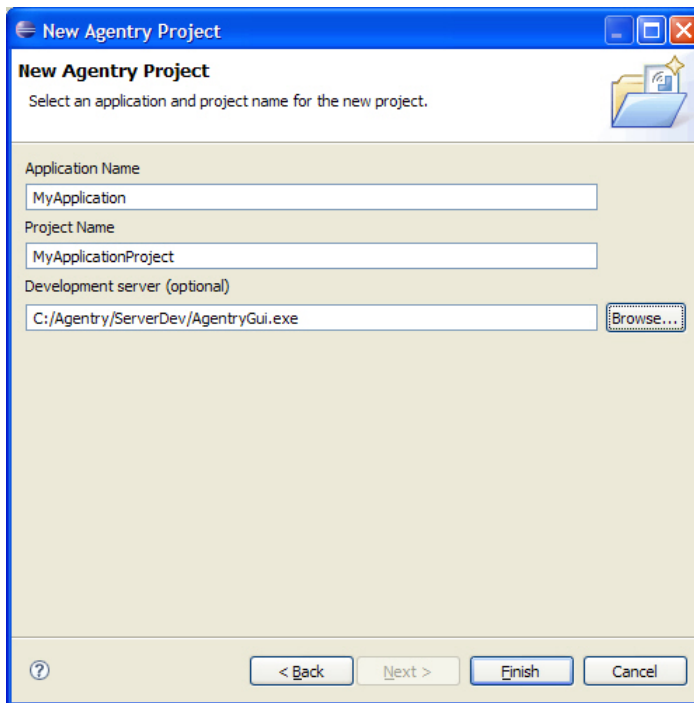
3. In this screen enter the name for the mobile application, the name of the project by which it will be identified in the Eclipse workspace, and optionally the location of the Agentry Development Server that will be used in the development of the project.

The next prompt displayed concerns adding a module to the application. All mobile applications have at least one module:
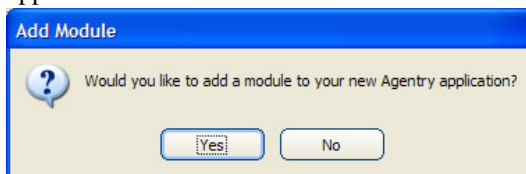
4. To add a module to the new application project, click the **[Yes]** button.

A new module is created with the name NewModule1, and which should be edited to a more meaningful value.

The new application project has been created and stored in the Eclipse workspace. Depending on the selections made during this process, the following definitions will now exist within this project:

• Application
• A set of default transmit configurations:

- • Dialin
- • Network
- • WirelessLan
- • WirelessWAN
- • A module definition
- • The module MainObject, which in turn contains a collection property defined for the BusinessObject object definition
- • A second object deifnition named BusinessObject, which should be renamed to a more meaningful value
- • The module main screen set, defined to display the MainObject
- • The module main fetch, defined to target the collection property within the MainObject.
- • An action named Transmit that includes a single action step of type Transmit.

Next

With the completion of this process the mobile application project is created and development work can begin. In addition, and likely before the development work, additional configuration may be needed of the overall development environment. This can include the following, depending on the nature of the project:

- • If synchronization with a Java Virtual Machine system connection is a part of the mobile application's behavior, create and configure a Java project and include the Agentry Java API packages, and other packages related to the back end system.
- • When the development project is ready for its initial publish, configuration of the Agentry Development Server will begin with the publish, and will then be completed through the Management Cockpit and possibly through modifications to the Server's configuration files.
- • Configuration of the Agentry Development Server to be used in the development process. This can include system connection configuration, logging behaviors, and other similar items.

Agentry Application Export, Import, and Comparison Introduction

A feature of the Agentry Editor is the ability to import and export application project definitions. Exporting from a project can include the entire project or any selected definitions within it down to the module level and will create a single Agentry Export File (`.agx`) containing all exported definitions. The source for comparing and importing to an existing application project can be an Agentry Export File, another application project, or a mobile application as published to the Agentry Server. An additional option for the source of an import is an Agentry 3.x Editor, which will also contain the application project it manages.

The uses for importing and exporting Agentry application projects include:

- Exporting to a single file in support of archiving an application project version control and backup.
- Exporting components of an application that contain differences from a base-line project. This is common when an product application has been configured or customized, and it is desirable to archive those changes for future reuse.
- Importing from an application project source to create a new project in the current Eclipse workspace. This may be done to create an application project from an archived export, or to upgrade an application project to a newer version of the Agentry Mobile Platform. Importing a project, Server published application, or export file created by a previous version will automatically upgrade the application project to the version of the Editor performing the import. Additionally, all products provided by Syclo are delivered with the Server and therefore must be imported to an instance of the Agentry Editor prior to extending or modifying the core functionality of the application.
- Comparing and importing components of another application project or export file to a current project to make use of common customizations or configurations, or to take advantage of previous development work in the current application project.

Import Functionality Overview

The import tools provided by the Agentry Editor provide functionality to support importing application definitions from other projects, published versions from Agentry Servers, and export files. Reasons for importing can include:

- Creating a new application project in the current workspace from an archived project or Agentry Server.
- Upgrading application projects to the current Agentry Mobile Platform version.
- Adding previously developed application definitions and components to the current project.
- Merging separate development work from multiple developers or a share repository created using the Team Development functionality.

Import Source
Import Source is the general term used to describe any item used as source for an import operation. Valid import sources within Agentry include:

- **Agentry Application Project Files** (`.apj`)**:** This may be any Agentry project file, including its related definition files, created by the Agentry Editor. This project can be stored in a different Eclipse workspace, or elsewhere on the file system if created by a version of the Agentry Editor from 4.0 through all 5.0.x releases. Note that the `.apj` file is the main project file, but it is not the entire application project. The definition files that are a part of that project must exist in the proper Agentry Editor created file structure in order for this source to be imported. Typically this is a non-issue as there is no valid reason for the definition files within a project to be moved or modified manually.
- **Agentry Export Files** (`.agx`, `.agxz`)**:** This can be any Agentry export file created using the Application Export functionality provided within the Agentry Editor. Note that as of

version 6.0 either standard or compressed export files can be imported. However, compressed export files cannot be imported by versions prior to 6.0.

- **Published Agentry Server Application:** An application published to the Agentry Server, either Development or Production, can be selected as an import source. The selection made for this source is the server's executable file. The application published to that server instance is then used as the import source. For Agentry Production Servers, any published version residing on that server can be an import source. Agentry Development Servers have only a single application version that may be an import source.
- **Agentry 3.x Editor Project:** In the 3.x versions of Agentry, application definitions were not stored in projects, but rather as a part of the data within the Agentry Editor instance. A given Editor contained the definitions for a single mobile application, meaning each mobile application included its own dedicated Agentry Editor. These definitions can be an import source, with the selection of the `AgentryEditor.exe` containing the desired mobile application to be imported.

As a part of the import process it is first necessary to select the type of import source. This information is required at the beginning of the screen flow for the import operation within the Agentry Editor. The above sources are valid import sources for import operations creating new projects within the Eclipse workspace, and for imports that add or replace definitions, known as a "compare and import" operation, within an existing project.

For each of these, the import source must have been created with a version of the Agentry Mobile Platform matching or prior to the version to which the definitions will be imported. There is no "downgrade" functionality provided by the import tools. As an example, it is not valid to select an Agentry export file created by version 5.2 as an import source for an instance of the Agentry Editor delivered in version 5.1. However, the reverse is allowed, importing from version 5.1 to version 5.2. This is, in fact, the proper method for upgrading an application project to a newer release of the Agentry Mobile Platform.

Adding a New Project to the Workspace
When importing to create a new project within the Eclipse workspace, the developer will first select the application source type and the specific source project. The import tool will then read in the source definitions, creating a new application project within the workspace. All definitions from the source are imported.

If the source for the import is not a full application project, or is missing definitions within the application hierarchy, changes will be necessary to the project. As a basic example, if an export file contains only the object definitions from a project, creating a new project by importing the export file will create a new Agentry application project containing an application and module definition. However, these parent definitions to the objects will contain minimal attribute settings with default values. Typically this is not a situation encountered often in real-world development environments.

Adding Definitions to an Existing Project
When performing an import to an existing application project, the current project and the source application for the import will be displayed in the Comparison View within the

Agentry Perspective. This view displays the current project and import source in side-by-side panes. Both are presented according the application hierarchy and are aligned based on the definition type and name. Each definition alignment is denoted as one of the following within this view:

- **No Difference:** The definition in both the project and import source are identical, including child definitions and attribute settings.
- **Unimportant Differences Only:** The definitions in both the project and import source are the same in all areas that would affect run time behavior. Differences were found, but were limited to comments or descriptions only.
- **Exists Only in Source:** The definition exists only in the import source. It is not found in the current project. Such a definition can be selected in the import source pane and imported into the current project. Alternately this definition can be aligned with an existing definition of the same type and with the same parent definition. The source definition will overwrite the existing definition if it is then imported.
- **Exists Only in the Project:** The definition exists in the project but not the source. An import operation will have no effect on such a definition. Alternately this definition can be aligned with a definition in the import source. The source definition will overwrite the existing definition if it is then imported.
- **Differences Exists Between the Definitions:** The definition exists in both the project and the import source, but there are differences between the two definitions. This can include differences in attribute settings or differences in the child definitions. Child definition differences can include attribute differences, or a different set of child definitions.

Within the Comparison View the developer selects the items in the import source to be imported into the current project. This consists of checking and unchecking boxes within the import source indicating the specific definitions and their child definitions should or should not be imported.

The Comparison View
The Comparison View has been added in Agentry version 5.2. Its functionality and behavior is similar to the Comparison Screen that served the same general purpose in prior versions of Agentry. The new Comparison View includes additional functionality in support of the Team Development feature set. It also includes more information concerning the comparison and import, with lists now displayed summarizing the differences found between the project and import source, and a Preview tab listing the potential results of performing the import based on the current selections within the view.

The following is an example of the Comparison View within the Agentry Perspective:
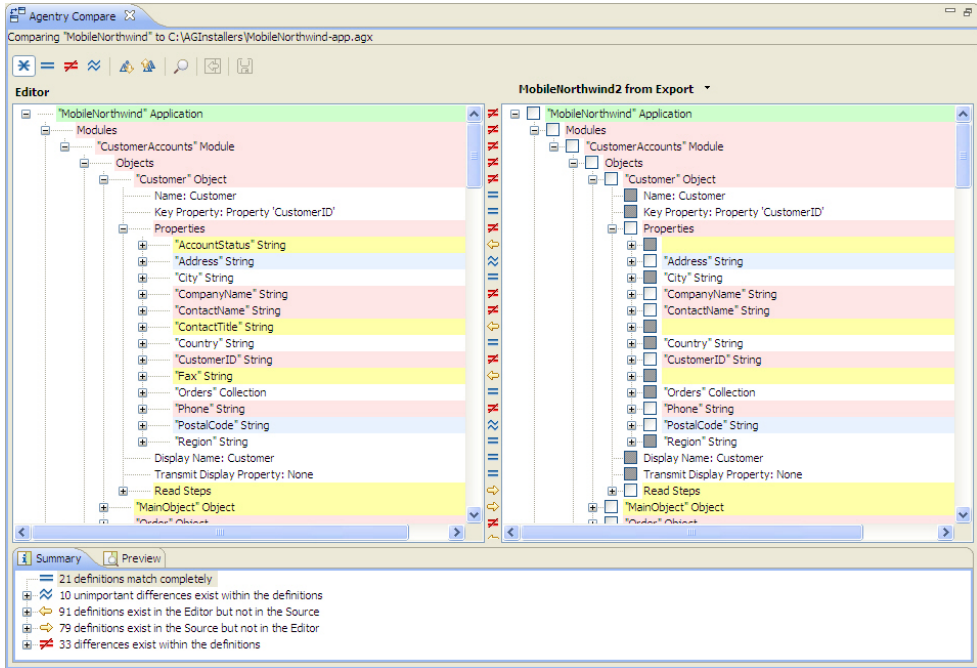
In the pane on the left side of the View is the Agentry application project currently open in the Agentry Editor. On the right are the application definitions in the import source. Between the two are the icons for each definition indicating the comparison status between the project and import source. Following is a list of these icons and their descriptions:

Difference Icon	Description
=	No Differences Found
≠	Differences in Definitions
≈	Unimportant Differences Found
⇦	Exists Only in Project
⇨	Exists Only in Import Source

Importing a New Agentry Project Into the Eclipse Workspace

Prerequisites

Address the following items prior to performing this procedure:

- Determine the source of the application project you are importing in this procedure and that you have access to that source.
- Verify that the application project source was created with the same or earlier version of the Agentry Mobile Platform. You cannot import projects, export files, or published applications created with a later version into an earlier version.
- Verify the workspace in which you are importing the project is the currently opened workspace in Eclipse.
- Determine a name for the project as it will be listed in the Eclipse workspace, as this is required information entered in the import process.
- Determine a value for the **Name** attribute of the application definition. This is information required during the import process.
- Though not required, it is strongly recommended that the Agentry Development Server to which development publishes are performed is installed. While the location of this Server instance is optional during the import, it is necessary information when defining any of the synchronization logic within the project after it is created.

Task

This procedure describes the steps involved in importing an application project into the current Eclipse workspace. When this procedure is complete, a new Agentry application project is created in the current Eclipse workspace. This project contains the definitions and application components found in the import source.

Note that this process excludes any related projects for the source application that may reside in that source project's workspace, such as Java development projects and related packages. Import these related projects and components according to the process that matches that project type, using tools found in Eclipse. Whether the Agentry application project is imported before or after other related projects is unimportant. However, all items must be imported and/or configured before modifications are made.

This procedure accomplishes the following:

- Checks out an Agentry application project from an Agentry share repository and creates a new local project based on the top revision within that repository.
- Migrates an application project from one workspace to another.
- Upgrades an application project or application export file created in a previous release of the Agentry Mobile Platform.
- Restores or recovers an application project from an archived project or export file.
- Creates, restores, or recovers an application project from a mobile application published to the Agentry Server.

1. If not already open, open or create the Eclipse workspace in which to import the new project. Opening or creating a workspace in Eclipse begins by selecting the menu item **File | Switch Workspace** and following the on-screen instructions.
2. Right-click an empty area in the Project Explorer View and select the menu item **Import...**. Alternately, select the menu item **File | Import...** in the Eclipse main menu.

The Select Import Source screen displays.
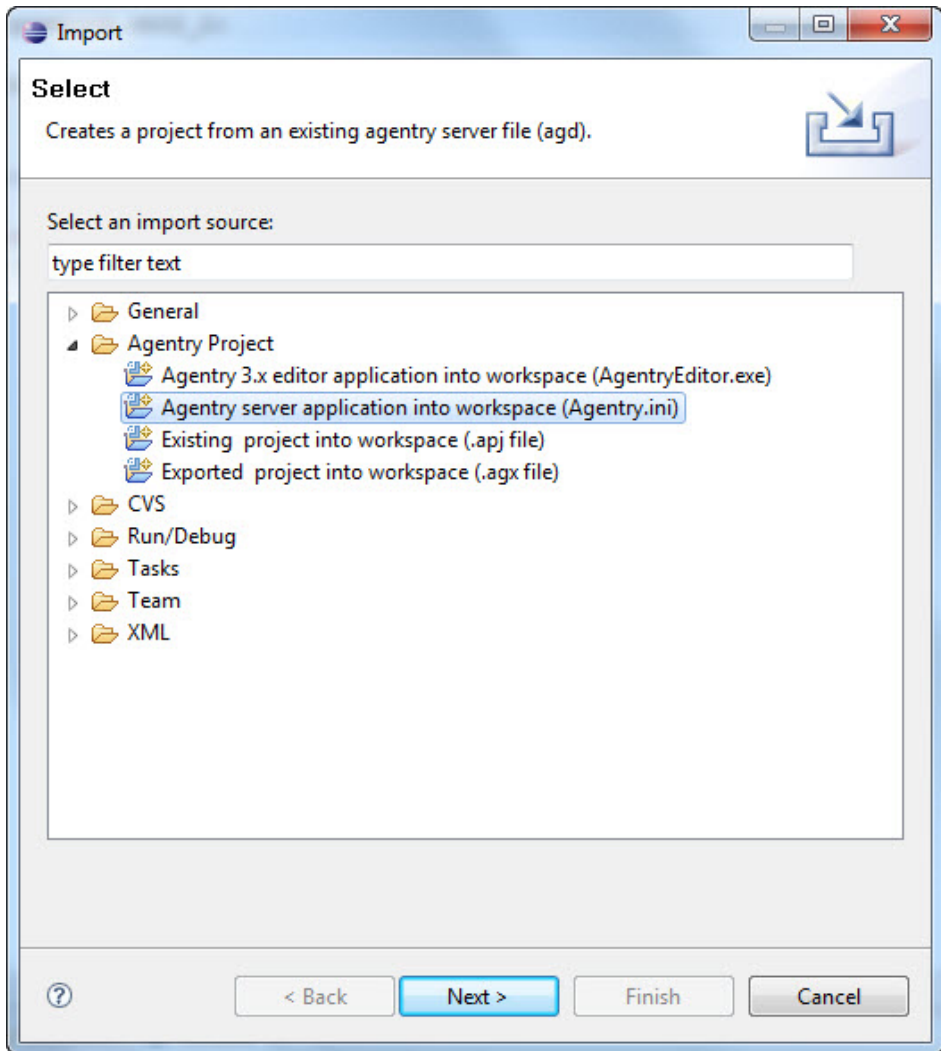
3. On this screen are the different import sources for Eclipse. Two of these pertain to Agentry application projects: **Agentry Project** and **Agentry Share**. Select the desired source by expanding one of these nodes and selecting the appropriate item under it. Click **[Next]** to continue.

The Select Source screen displays. This screen will be slightly different depending on the selected import source type. The following example is for a source type of Agentry Server application:

4. In this screen the information entered is dependent on the source type selected in the previous step. Enter the information according to the following:

a) The specific item selected for the source is different based on the type. Select the source by clicking the **[Browse]** button.

b) The **Source Application** box is only displayed when the source type is an Agentry Server. This lists the name of the application published to that Server instance. If the selected server instance is a production Server, each published version currently residing on that Server is listed and you can import any one of those versions.

c) The **Application Name** is the name given to the mobile application that is created by the import. This is set as the value of the Name attribute within the application definition and can contain no white space. This field is read-only if the selected import source is an Agentry share repository.

d) The **Project Name** is the name for the project within the Eclipse workspace. This must be a unique project name for the workspace and white space is allowed.

e) The **Development Server (optional)** is the Agentry Development Server for the application project being created. Any script files contained in the source project are copied to this Server. By default, this field is set to the Server from which the project is imported, if that Server instance is a development Server. Leave the option set as-is if this is the development Server for the new project, or change to a different development Sever if necessary.

5. Verify the information entered is accurate and complete. Click **[Finish]** to perform the project import.

   A new project is created by importing the definitions from the selected import source. The project is listed in the Project Explorer View and is automatically opened.

After this process is complete, the new project is added to the Eclipse workspace. The project is opened and displayed in the Agentry Perspective within Eclipse. The application name and project name match those values entered in the Import wizard. If the application project source was created using a previous version of the Agentry Mobile Platform, the new project was upgraded during the import to the version of the current Agentry Editor.

If the selected import source was an Agentry share repository, the new project contains the definitions found in the top revision of that repository. The revision is checked out to the current user. Subsequent changes made to the application project are tagged with that user ID. The project is connected to the selected share repository and you can update from it. Commit changes made locally to this repository.

Compare and Import Into an Agentry Application Project

Prerequisites

Before importing an application project source into an existing application project, the following items must be addressed:

- Determine the import source and verify access to that source.
- Verify the import source was created with the same or earlier version of the Agentry Mobile Platform. Export files or Server published applications created with a later version cannot be imported into an earlier version.
- Verify the correct Agentry application project to which definitions will be imported is the one currently open in the Agentry Perspective within Eclipse.
- Back up the current project by exporting the entire project, or by committing any changes to the Agentry share repository prior to beginning the import.

Task

This procedure describes the steps involved in importing definitions from an import source into the current Agentry application project. This process includes comparing the two projects and selecting those components to import to the current project. When completed, any definitions selected in the import source will be added to the current application project. The source application will be unaffected by the procedure. This process may be cancelled at any time.

This procedure should be performed to:

- Compare and import from the currently connected share repository. This is a feature included in the Team Development support added to the system with the release of Agentry version 5.2. This includes manual compare operations where the share repository is selected, as well as updates from the share that result in conflicts.
- Make use of archived customizations or components in the current application project.
- Merge development work performed by multiple developers into a single master project. Note that this is *not* a part of the Team Development support and feature set. It is recommended that the Team Development features be used when coordinating work among multiple developers for the same application. Importing from an export file of another developer's work can still be performed, but should be limited to the scenarios of handing off responsibility for development, or when making use of modifications made to one project that are needed in another application project.

1. If not already open, select and open the Agentry application project to which definitions are to be imported in the Project Explorer View.
2. Right click the root folder for the application project in the Project Explorer View and select the menu item **Compare With | Compare with other Agentry project**.

   This will open the Comparison View in the Agentry Perspective, with no project items yet displayed. It will remain blank until the import source is selected for comparison:
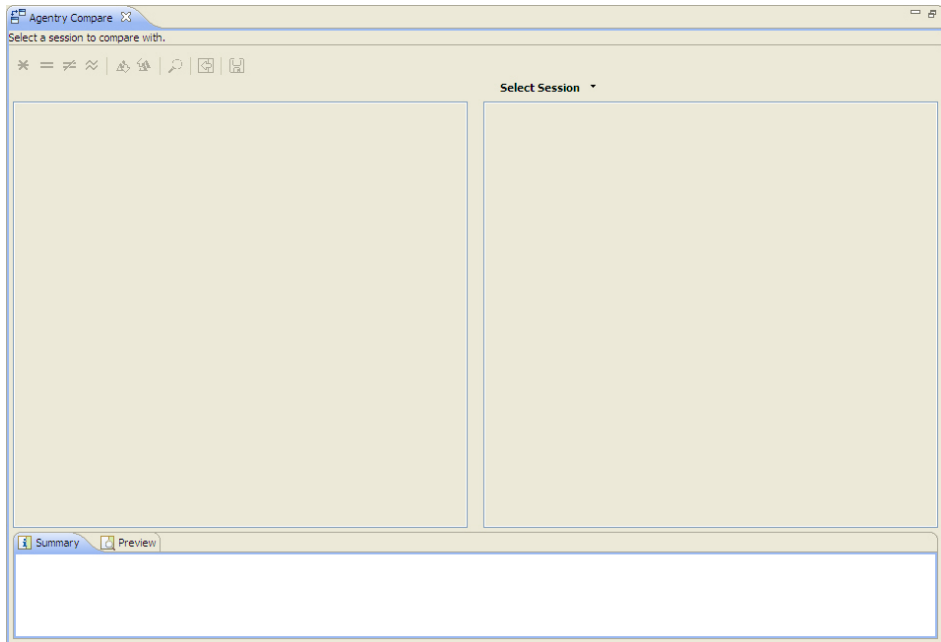
3. Within this view, above the pane on the right side, is the button **[Select Session]**. This button allows for the selection of the import source. Its behavior and the proper selection to make depend on whether or not the open project is currently connected to a share repository:

   • If connected to a share repository, clicking the button will automatically open the share revision matching the last revision to which the project was updated. The share is then set as the import source, and other revisions can be selected within the share to compare and import from within the share.

   • If not connected to a share, clicking this button opens the Manage Share Sessions screen. Within this screen either a previous import session can be reopened, or a new import source can be selected by clicking the **[Browse]** button below the list of save sessions:
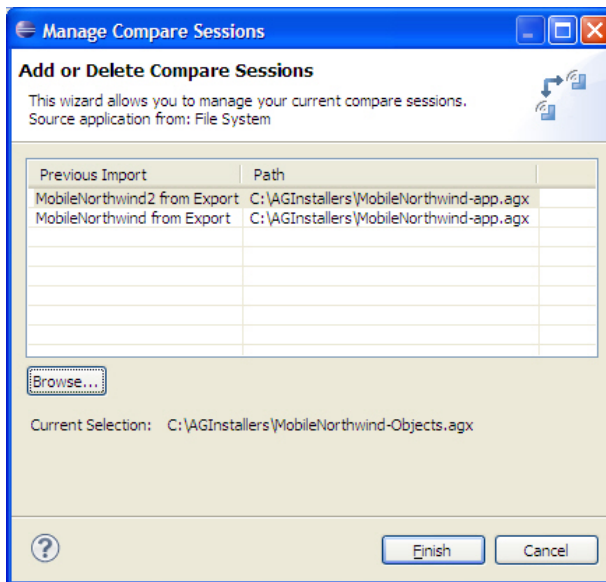
- Alternately, whether connected to a share or not, clicking the drop down arrow for this button lists the import sources appropriate to the current project's connection status. Always listed are any previous import sessions, as well as a menu item to display the Manage Compare Sessions screen. If connected to a share, an additional menu item is displayed that opens a sub-menu listing all revisions of the share to which the project is connected.

Once the import source is selected, that source and the current project are opened in the Comparison View and displayed side-by-side:
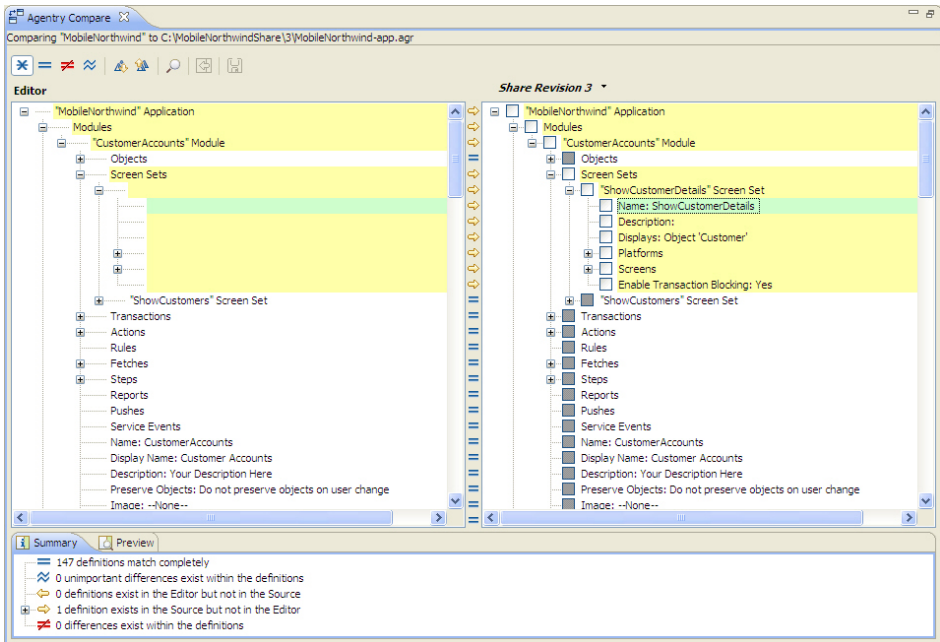
4. Within the comparison screen now displayed, the pane on the left is currently opened application project. The pane on the right is the import source. Differences between the two are highlighted. Items can be selected in the import source to import to the current project by checking the boxes for those item's nodes. Definitions are aligned by default in the two panes based on definition name and type. To force two definitions of the same type but with different names to be aligned, right click either one and select the menu item **Align With |** *definition name*, where *definition name* is one of the possible definitions with which the selection can be aligned. Once the all of the definitions to be imported from the import source have been selected, click the button Import in the view's toolbar. To save the changes, click the apply button in the toolbar.

   The definitions will be moved to the left side pane indicating where they will be placed in the application project once imported.

5. Close the Comparison View by clicking the **X** button on the view's tab.

Once this procedure is complete, the selected definitions in the import source are now a part of the open application in the Agentry Perspective. They may be modified further within the open project or otherwise used as needed. If the session was cancelled at any point, any applied imports will remain. Any imports that were not applied will be rolled back.

Next

Whenever importing into an existing application project, it is always recommended that a check is performed of the resulting project to verify the new definitions are defined as needed. Any modifications can be made according to normal processes. All imports, as with any other

application change, should be thoroughly tested before being published to a live production environment.

Export Functionality Overview

Export operations within the Agentry Editor are performed to store multiple application definitions, including a complete application project, in a single file known as an Agentry Export File (`.agx`, `.agxz`). When performing an export operation it is possible to manually select the definitions to be exported from a given application project, or to export the differences between a project and some comparison source project. Available with Agentry version 5.2 it is also possible to select the definitions to export based on one or more tags having been previously applied to those definitions.

When a definition is exported, the definition's attributes and child definitions are included by default. It is possible to select a definition for export and then deselect one or more of its child definitions. The resulting definition in the export file will only contain the selected child definitions.

Exporting Definitions - Manual Selection

When exporting definitions from an Agentry application project, it is possible to manually select the individual definitions to be saved in the export file. This can include selecting the entire application project, or any individual definitions within it. Also, it is possible to select to export definitions based on the tags that have been applied to them within the application. During the export process, one or more tags can be selected and those definitions that have been tagged accordingly are selected for export.

It is a common practice to export an entire application project to a single export file as a backup prior to making significant changes to a stable release of the application. Note that this same result can be accomplished using the Team Development feature set available in Agentry version 5.2 by committing a project to the share repository as a new revision prior to making modifications to the project. Either method of backup is acceptable and the one best suited to a given developer's environment or preference should be used.

Exporting Definitions - Exporting Application Differences

Exporting application differences is functionality that can be useful when making modifications to a core application for implementation-specific needs. Once such changes are complete, exporting the definitions involved in just those modifications can be accomplished by comparing the modified version of the application project with the original version. It is then possible to store such changes for later uses, including importing the modifications into the same core application at a different implementation with similar requirements.

Exporting differences involves comparing the Agentry application project with either an Agentry export file or an application as published to an instance of the Agentry Server. This comparison will determine which definitions exist in one version but not the other, and which definitions exist in both but are different from each. During the export process the source for the export is then selected, either the project or the comparison source. The resulting Agentry

export file will contain only those definitions found to be different as they exist in the selected export source. Adhering to this practice whenever implementation-specific modifications are made to a standard product application can result in a robust library of common customizations that can be imported into future implementations for less labor and time intensive implementation projects.

When performing an export of differences, a comparison is always made between the open Agentry application project within the Agentry Editor and a comparison source, which cannot be an Agentry application project. Valid comparison sources include:

- **Published Agentry Server Application:** An application published to the Agentry Development Server can be selected as a comparison source. Production server applications cannot be selected. The selection made for this source is the server's `Agentry.ini` configuration file. The application published to that Server instance is then used as the comparison source.
- **Agentry Export Files (`.agx, .agxz`):** This can be any Agentry export file created using the Application Export functionality provided within the Agentry Editor. Note that compression of export files is available in the 6.0 version and later. Compressed export files (`.agxz`) cannot be used as a comparison source in versions prior to 6.0.

Once a valid comparison source is selected, that source is compared to the open Agentry application project, with differences between the two highlighted in a comparison screen. Within this screen the export source, i.e., either the project or comparison source, is selected. When the export file is created, it contain the definitions found to be different within the selected export source.

Exporting Agentry Application Project Definitions

Prerequisites

The following items must be addressed prior to performing this procedure:

- The application project to be exported must be open in the Agentry Perspective.
- Identify the location of the export file to be created by this process and verify read-write and network access to that location.
- Determine the desired file name and a brief description for the export file that will be useful for later reference and identification. This information is entered during the export process.
- If exporting a subset of the definitions within the application project, identify and take note of those definitions before proceeding.

Task

This procedure describes the steps necessary to export definitions from an Agentry application project. During this process it is possible to select definitions with the application project to be exported, or to export the entire application. When selecting individual

definitions, all child definitions to the one selected will also be selected by default. It is possible to export definitions down to the module-level.

This procedure should be performed to:

- Archive an application project for version control and back up purposes.
- Export and archive components of a project for version control and back up purposes, or to make available for merge into a master application in a multi-developer effort (see information provided on Team Configuration for an alternative to this manual procedure).

With the release of the Agentry Mobile Platform 6.0 the default behavior is to create a compressed export file (.agxz). The preference pages in Eclipse for the Agentry Editor plug-in provide the ability to change this default behavior to create standard export files. The process for creating an export file is the same regardless of whether or not the file is compressed.

1. To begin the export process, right click the root project node in the Project Explorer view of the open Agentry application project. In the context menu select the item **Export...**

This displays the Export type selection screen:
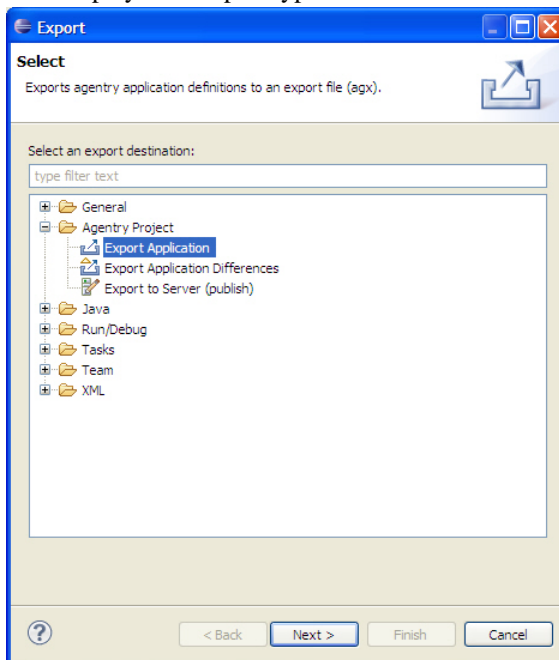
2. Select the **Agentry Project | Export Application** item in this screen. Click the **[Next >]** button.

This will display the first screen of the Export Wizard where the application definitions to export can be selected:

---
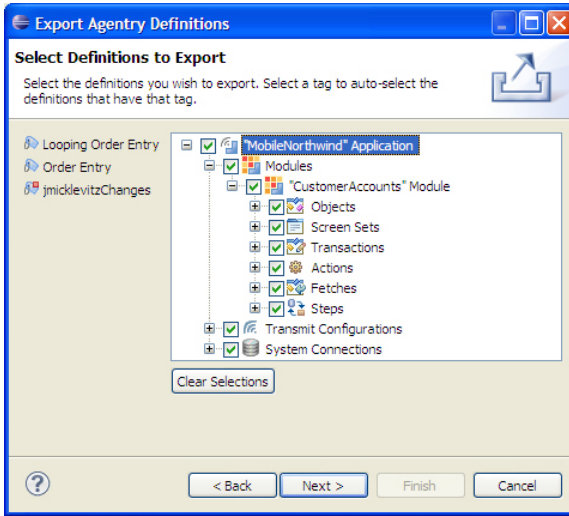
3. Within this screen the definitions of the application project are displayed and can be selected for export by checking the associated box for each. Checking a given definition will automatically select all child definitions. Alternately, one or more of the tag buttons to the left of the project can be selected, which will then automatically select all definitions with the associated tag to be exported. To export an entire application project, simply check the root Application node in this screen. Click the **[Next >]** button to proceed.

This will display the next screen of the Export Wizard where the name and location for the export file is entered, as well as an optional comment:
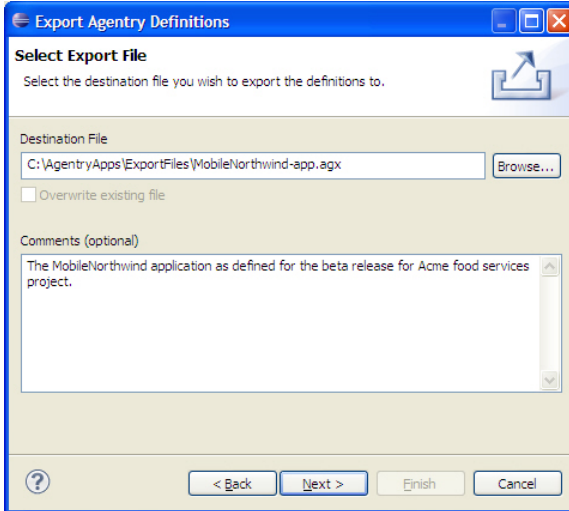
4. In this screen select the location and file name for the export file to be created. Ensure the selected location is one to which the Windows user has read-write privileges and network access where applicable. A comment can also be entered at this time. The contents of the

comment field are displayed as tool tip for the export file once it is created. Information concerning the date and time of creation and the version of Agentry are always a part of the tool tip for the file and need not be a part of the comments. Click the **[Next >]** button to proceed.

The export will now begin. When completed, the following summary screen is displayed:
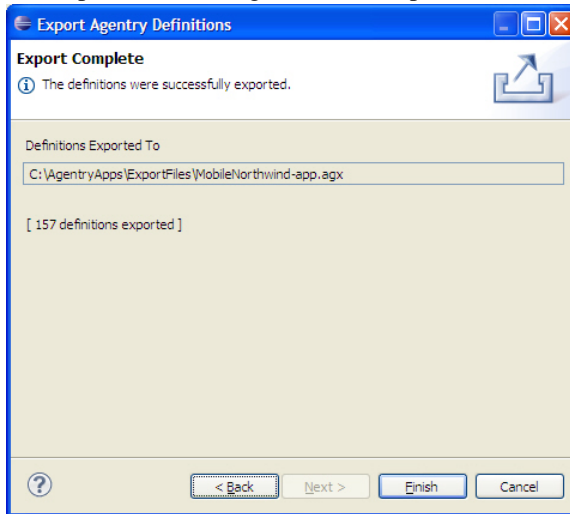
5. Click the **[Finish]** button to close the Export Wizard.

Completion of this procedure results in the creation of an Agentry Export File containing the selected definitions from the Agentry application project. This file can now be archived in a version control system, made available to other developers for import, or moved or copied to any desired location. It can be used as an import source to create new Agentry application projects or to import definitions into another project where needed.

Exporting Agentry Application Project Differences

Prerequisites

The following items must be addressed prior to performing this procedure:

- The Agentry application project to be compared against a comparison source must be open in the Agentry Perspective within Eclipse.
- The comparison source must be accessible to the Windows user and Eclipse.
- The name and location for the export file to be created should be determined and read-write and network access to this location should be confirmed.
- A comment for the export file should be determined that will be useful for later reference.

Task

This procedure describes the steps necessary to export the differences between an Agentry application project and a comparison source. This process will create an Agentry Export File

containing the definitions from either the open application project or comparison source deemed different from the other. This includes definitions found in one but not the other, or definitions found in both but that contain attribute differences. Before the export proceeds, those definitions to be exported are highlighted in the Comparison View.

This procedure should be performed to:

- Capture the differences between one application version and another for archive purposes.
- Capture differences made for implementation-specific configuration or customization. Such changes can be archived for later import into other implementations with similar functionality requirements.
- Other use cases where it is desired to export the differences between two Agentry application projects.

With the release of the Agentry Mobile Platform 6.0 the default behavior is to create a compressed export file (`.agxz`). The preference pages in Eclipse for the Agentry Editor plug-in provide the ability to change this default behavior to create standard export files. The process for creating an export file is the same regardless of whether or not the file is compressed.

1. To begin the export process, right click the root node of the open Agentry application project in the Project Explorer View and select the item **Export...** in the context menu. Alternately, select the menu item **File | Export...** in the Eclipse menus.

   This will display the Export Type Selection screen, where the type of export is selected:
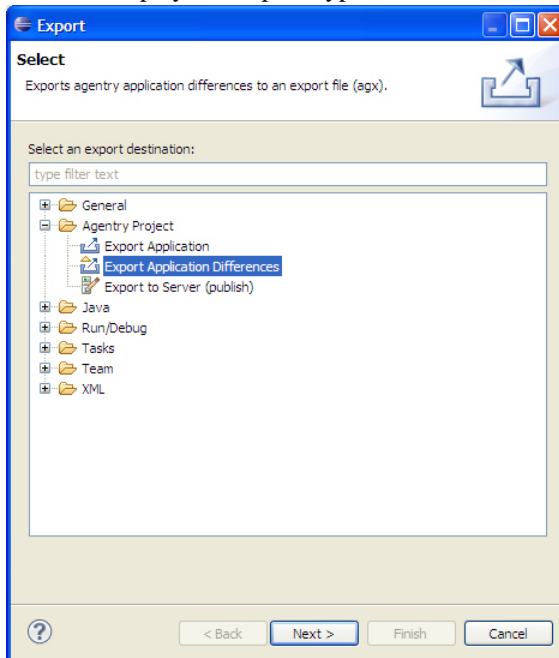
2. To export the differences between two projects, select the item **Agentry Projects | Export Application Differences**. Click the **[Next >]** button to proceed.

This will display the first screen of the Export Differences wizard:
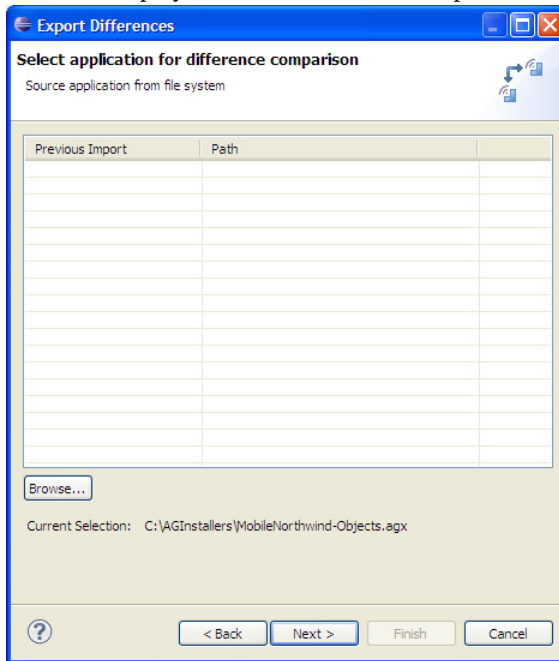
3. The list control on this screen displays the previous export differences sessions. Items can be selected in this list as a comparison source. Alternately, to select a difference source click the **[Browse]** button below the history list to display a Windows file dialog where the comparison source can be selected. Valid options are either an Agentry export file or an instance of the Agentry Server. Once the comparison source is selected, click the **[Next >]** button.

If a new source is selected, a prompt is displayed to enter a name for the history list. Enter this now if necessary. The next screen displayed is the comparison screen:
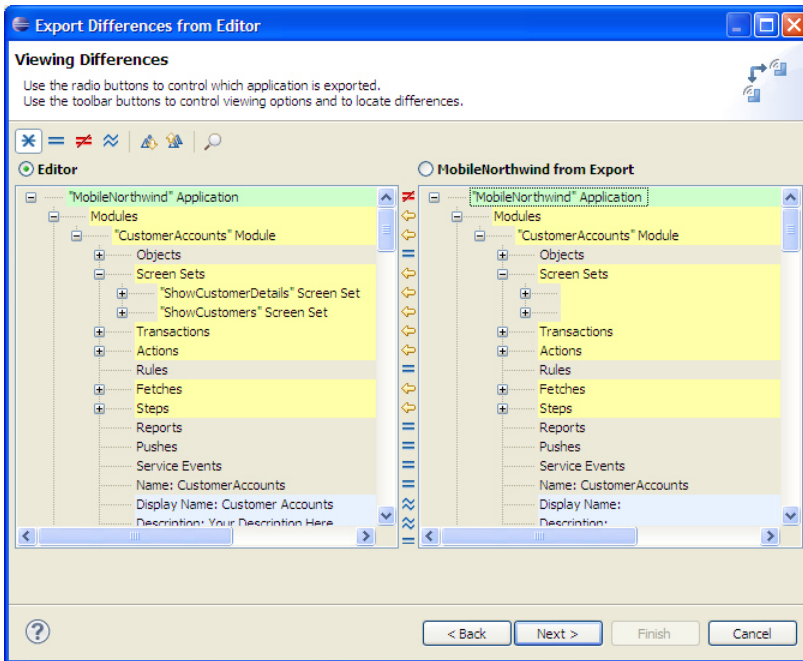
4. This screen displays the Agentry application project on the left, and the comparison source on the right. Differences between these are highlighted. Above each is a radio button that can be selected to indicate that the definitions from that source, either the Application project in the workspace, or the comparison source, will be exported. The definitions exported from the selected source will be those found to be different from, or that do not exist in the other project. Once the selection has been made, review the definitions to be exported and then click the **[Next >]** button.

This displays the destination and comment screen:
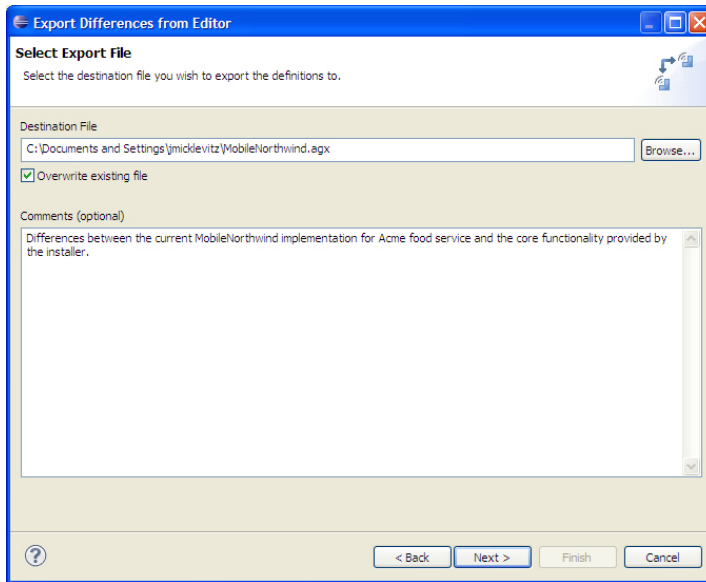
5. Within this screen select the location and file name for the export file to be created by this process. If the file exists, check the Overwrite existing file check box to replace the existing export file. Optionally, enter a comment for the export file. This comment is displayed for the file as a tool tip in any Windows Explorer or File Dialog. Information concerning the date and time the file is created and the version of Agentry that created it are automatically a part of this tool tip and need not be a part of the comments. Click the **[Next >]** button to proceed with the export.

The export is performed and the export file is created. The following summary screen is displayed:
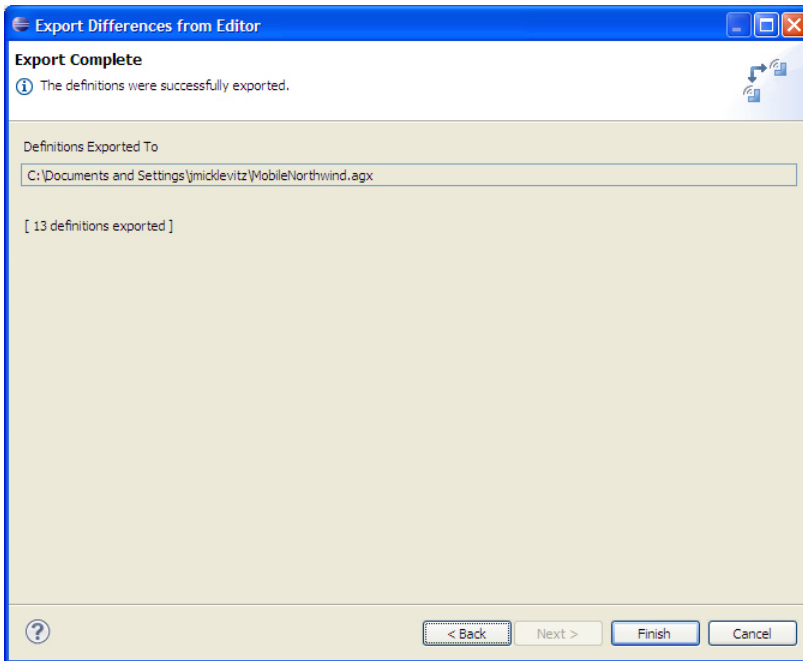
6. Click the **[Finish]** button to close the Export Differences Wizard and return to the Agentry Perspective in Eclipse.

Completion of this procedure results in the creation of an Agentry Export File containing the definitions from the selected source, either the open project or comparison source, found to be different. This file can now be archived in a version control system or other repository, made available to other developers for import, or moved or copied to any desired location. It can be used as an import source to add these definitions to another Agentry application project where similar functionality is needed.

Publishing Applications Overview

In order to test application projects or to deploy applications to users, the application project must be published from the Agentry Editor to the Agentry Server. If you are ready to distribute the application to the end users, it must then be deployed to the SAP Mobile Platform Server, after it has been published. At this point it will then be made available to and downloaded by those Agentry Clients that synchronize.

There are two types of publishes that can be performed, Development and Production. When publishing an application from the Agentry Editor, there are various aspects that will affect the type of publish performed and which options to select during the process. Additionally, the first time an application project is published there is some initial configuration performed by the publish process that will affect the Agentry Server.

Other options exist for the publish process, including the creation of files used for localization of the application. When performing a production publish, the publish version of the application must be set. Also during a production publish the option exists to delay the deployment from the Agentry Server to Agentry Clients until a specified date and time.

For all publish operations the application data is copied to the Agentry Server, with transformation of the application data into the format for the Agentry Server. This format will vary depending on whether the target Agentry Server is configured for development or production. The location of the application data will be in the Agentry Server's configuration directory, in either the subdirectory `Application\Development` for a development publish, or `Application\Production\Version` for a production publish. To perform a publish it is necessary to have read-write access to this location.

Publishing During the Development Cycle

When developing a new application or when customizing an existing one, it will be necessary to perform several publishes to an instance of the Agentry Server provided within the SAP Mobile Platform Server. This Agentry Server instance should be installed as a development server; that is, the Configuration property Development Server should be set to true. Note that it is false by default and must be modified after the SAP Mobile Platform Server has been installed.

Publishing to Production for End Users

When development and testing are complete, the application can be published to the SAP Mobile Platform Server. This is accomplished by performing a production publish to the Agentry Server configuration directory within the SAP Mobile Platform Server installed for end users. A production publish includes specification of a publish version number, which dictates certain application update behaviors performed by during the next transmit, and as described in the information provided specific to production publishing.

Compression of Published Files

The files generated during a publish are compressed by default. This behavior can be altered using the preference settings in Eclipse for Agentry. There is no difference in the process of publishing the application based on whether or not the files are compressed. The Agentry Server automatically determines whether or not the files are compressed and will process them accordingly.

Development Publish Description and Overview

When performing development work, customizations, or configuration to an application project, the development and unit testing is normally performed in the Agentry Development Environment with components installed from the SAP Mobile SDK. This includes the Agentry Development Server to which the application will be published, and the Agentry Test Environment, used for development and other testing of client functionality. The Agentry Server is installed within the SAP Mobile Platform Server and must be configured to be a development server. See the administration content on the SAP Server for details on changing

this configuration. When publishing to a Development Server, the publish performed is a development publish. A development publish should only be performed to an Agentry Development Server.

A development publish will transfer the definition files in the application project from the Agentry Editor to the Agentry Development Server. The files are stored on the file system at the installation location of the SAP Mobile Server; specifically within the configuration directory of the Agentry Server in the sub-directory `Application\Development`. A development server has no concept of application versions. When a development publish is performed, the previous version of the application on the Server is overwritten.

In a development publish, SQL scripts, file system scripts or batch files are written to the file system as separate, editable files. This allows for modification to these synchronization components on the Server without the need for a publish. Any other application modifications made require a publish to the Server to be available to the Agentry Clients or Agentry Test Environment.

The options for a development publish are limited to selecting the Agentry Development Server to which the application will be published, and whether or not to create the localization base files. Delayed deployment and version information cannot be set.

The main reasons for performing a development publish are to perform development and unit testing of an application during the development process. Additional testing should also be performed in a Production environment established for the purpose of quality assurance and user acceptance testing.

Publishing to Development

Prerequisites

Before publishing an application project to the Agentry Development Server, the following items must be addressed:
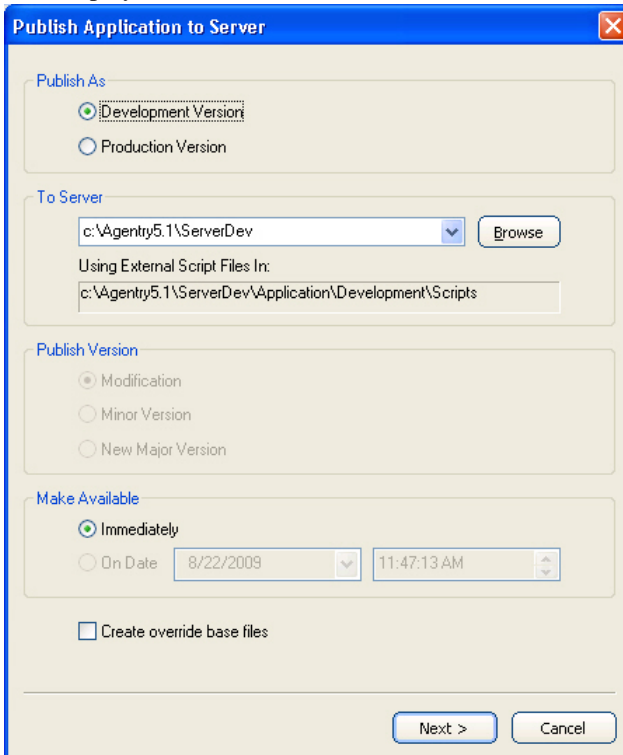
- Verify the Windows user for the Agentry Editor host system has read-write and network access (where applicable) to the installation location of the Agentry Development Server.
- Confirm the application project is in a state for which a publish is reasonable and ready for testing.
- During the publish process checks will be made as to the overall integrity of the application project. Errors or warnings will be listed in the Problems View in Eclipse. Prepare for the possibility of needing to correct these issues in order to perform a successful publish. An application can be published with warnings, but not with errors.
- Verify the version number of the Agentry Development Server and the Agentry Editor are the same, excluding any patch or point releases. Publishing from an Editor from one major or minor release to a Server from another release is not supported.
- The Agentry application project must be open in the Agentry Perspective.

Task

This procedure describes the steps necessary to perform a development publish from the Agentry Editor to the Agentry Development Server. When this process is complete, the application project will exist on the Server and will be deployed to any Agentry Clients that perform a transmit with that Server. This process is performed whenever it is desired to publish an application project for the purposes of development or unit testing in advance of eventual deployment to a production environment.

1. Begin the publish by clicking the toolbar Publish button in Eclipse.

   This displays the first screen of the Publish Wizard:

2. All options affecting the publish, including the type of publish, are set on this screen. For a development publish the following options are set:

   • The **Publish As** option is set to Development Version.
   • The **To Server** option lists the most recently selected Development Server, or the Development Server specified when the application project was created. Any other Development Servers to which a publish has been performed will be listed in this drop down. Select the Development Server in this field by selecting it from the drop down or by navigating to the Server's installation location using the **[Browse]** button.

• The **Create override base files** option can be set to generate the files used in support of localization. These files will contain all the display strings within the application project, with each display string including an identifier. The contents of these files can be translated and reincorporated in the Server to localize the application. See available information on localizing an Agentry application on the next steps after these override base files have been generated.

Once these options are set click the **[Next >]** button to begin the publish. The publish status screen will be displayed:
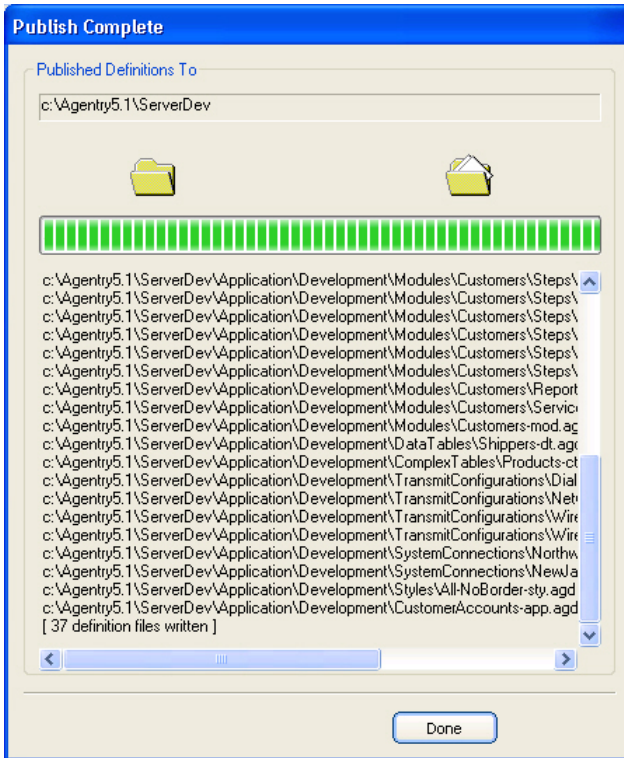
3. The total count of definitions published to the Server is now displayed. Click the **[Done]** button to close the Publish Wizard and return to the Agentry Perspective in Eclipse.

Performing a development publish results in the business logic encapsulated in the definitions being transferred to the Agentry Development Server where they can be deployed to clients connecting to that server instance. Any previous version of the application published to the same server instance is overwritten by subsequent publishes.

Production Publish Description and Overview

A production publish is performed to an instance of the Agentry Production Server within the SAP Mobile Platform Server, typically when it is time to deploy the application to users. Additionally, production environments are normally established for the purposes of quality

assurance testing prior to deployment. This can include publishing a new application project or publishing modifications to an existing project to create a new application version.

A production publish will transfer the application project from the Agentry Editor to the Agentry Server. The location of the project on the file system for the Agentry Server is the Agentry Server configuration directory in the sub-directory `Application \Production`. The entire project from the Agentry Editor is written to a single file at this location with an extension of `.agpz`. This file encapsulates the application for the version published. Note that in addition to this file, it may be necessary to manually transfer additional resources, where needed, depending on the nature of the application's synchronization logic. Examples include `.jar` files containing Java resources, `.dll` files, or other similar resources. Such files are completely dependent on the mobile application's architecture.

During the production publish the version number and deployment date and time can be specified. The specified version number can affect the behavior of the application during client-server synchronization of the new version, with a change to the major version number having one affect and a change to the minor or modification number another.

Production Publish Version Number Selection
As a part of the process of performing a production publish, the version number for the application must be specified. The first time a production publish is performed to the Agentry Server, the version number will be 1.0. During subsequent production publishes to the same Agentry Server instance, the version number is changed based on the selection made in the Publish Wizard.

An Agentry project's publish version contains the three components major, minor, and modification number. In the publish version 3.2 mod 1, the 3 is the major version, 2 is the minor version, and mod 1 is the modification number. The Publish Wizard will contain a section where the developer will specify which of these should be changed. This selection impacts the behavior of the application during the next synchronization by the Agentry Clients.

The specific difference is related to the processing of transactions sent from the Agentry Client to the Agentry Server. Depending on the nature of the change made to the application, it is possible that changes captured on the Agentry Client in transactions using the previous application version will be incompatible with the new version of those same transactions. When the synchronization definitions of the transaction (server data state or update steps) perform processing that will not work with the previous version of the application, it is necessary to process the pending transactions with the previous application version. To force this behavior, change the production publish version by incrementing the major version number.

When a new major version is specified, the Agentry Server will process each transaction sent by a Client using the old version of the application. Once all transactions have been processed, the new version of the application will be sent to the Agentry Client and dictate any subsequent synchronization behaviors. When either the minor or modification numbers are changed, the

new version of the application takes effect immediately, before the pending transactions on the Agentry Client are processed.

Because of this behavior, it is the responsibility of the developer performing the publish to determine which behavior is desired, and to then select the proper change to the publish version of the application.

Publishing to Production

Prerequisites

The following items must be addressed prior to performing this procedure:

- Confirm the application project is in a state for which a publish is reasonable and ready for production use or user acceptance and quality assurance testing.
- Obtain the directory within the SAP Mobile Server to which the project should be published from the system administrator.
- Verify the Windows user for the Agentry Editor host system has read-write and network access to the configuration directory of the Agentry Server defined within the SAP Mobile Platform Server.
- During the publish process checks will be made as to the overall integrity of the application project. Errors or warnings are listed in the Problems View in Eclipse. Prepare for the possibility of needing to correct these issues in order to perform a successful publish. An application can be published with warnings, but not when any errors are found. It is recommended that any warnings are corrected prior to performing a production publish, especially when deploying to end users.
- Verify the version number of the Agentry Server and the Agentry Editor are the same, excluding any patch or point releases. Publishing from an Editor from one major or minor release to a Server from another release is not supported.
- The Agentry Application Project must be open in the Agentry Perspective.
- Determine which of the major, minor, or modification number of the publish version should be changed, based on the desired transaction processing behavior related to Agentry Client transmits.
- Determine if the application should be immediately available, or if deployment to the Agentry Client should be delayed to a future date and time.
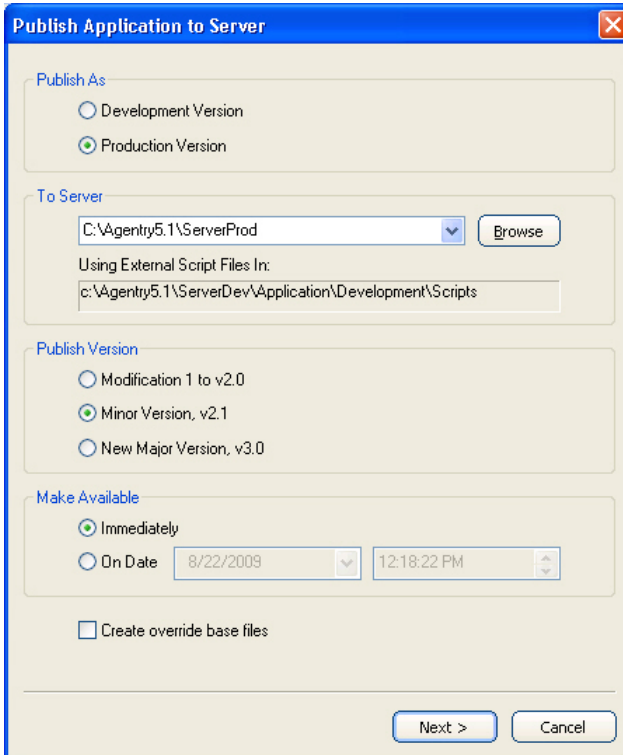
Task

This procedure describes the steps necessary to perform a production publish from the Agentry Editor to the Agentry Server. When this process is complete, the application project will exist on the Agentry Server. From here it can be packaged, along with other resources, for deployment to the SAP Mobile Platform Server.

In addition, Agentry Clients can also synchronize with this server instance for testing purposes, if desired.

**1.** Begin by clicking the toolbar Publish button in Eclipse.

This displays the first screen of the Publish Wizard:

2. All options affecting the publish, including the type of publish, are set on this screen. For a production publish the following options are set:

- The **Publish As** option is set to Production Version.
- The **To Server** option lists the most recently selected production server. Any other production servers to which a publish has been performed are listed in this drop down. Select the production server in this field from this drop down, or by navigating to the Server's installation location using the **[Browse]** button. The proper location is the configuration directory of the Agentry Server defined within the SAP Mobile Platform Server for the mobile application.
- The **Publish Version** is specific to a production publish. The option selected here increments one of the Major, Minor, or Modification numbers of the publish version on the Server. Increment the Major version when it is necessary to process transactions using the previous published version before providing Clients with the new version. Increment the Minor or Modification number when this processing is unnecessary.
- The **Make Available** option is specific to a production publish. This option can specify whether to make this published version of the application available to Clients immediately, or at a future date and time. When a date and time is specified, the

application is published immediately, but will not be deployed to Clients until after the specified date and time.

- The **Create override base files** is set to generate the files used in support of localization. These files contain all the display strings within the application project. The contents of these files can be translated and reincorporated in the Server to localize the application.

If this is the first time an application has been published to the Agentry Server a prompt will be displayed when advancing the wizard indicating changes will be made to the Agentry.ini file. These changes include the addition of configuration sections related to the specific system connections defined within your project. Accept this message to proceed as these sections are needed.

Once these options are set click the **[Next >]** button to begin the publish. The publish status screen is displayed:
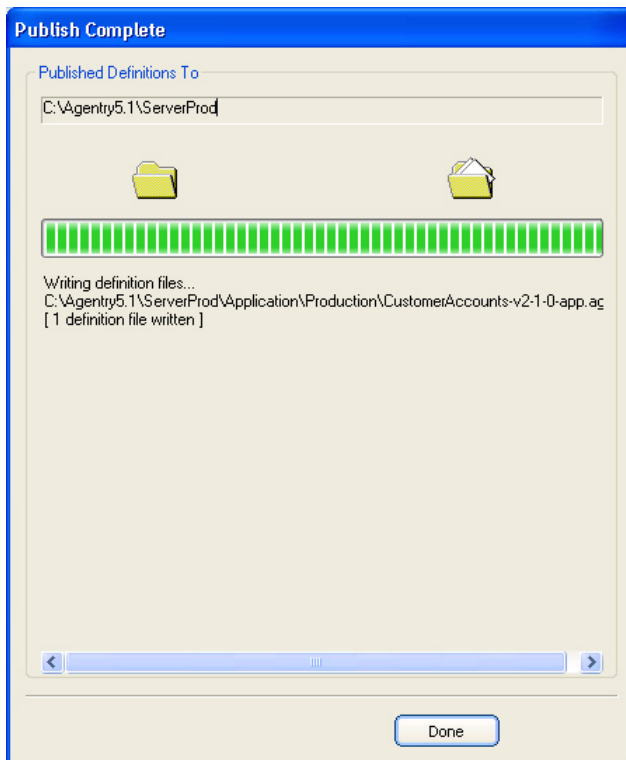
3. The total count of definitions published to the Server is displayed at the end of the publish. For a production publish this is always one file, regardless of the number of changes made. Click the **[Done]** button to close the Publish Wizard and return to the Agentry Perspective in Eclipse.

Performing a production publish results in the transfer of the application logic to the Agentry Production Server. The publish version of the application on that server instance is changed

according to the selection made in the Publish wizard. The business logic contained in the `.agp` or `.agpz` file, along with the accompanying `.ini` file, can now be bundled with the ZIP archive for deployment to the SAP Mobile Platform Server.

Introduction to Definition Tags

As of version 5.2 of the Agentry Mobile Platform the concept of Definition Tags is available. Definition tags, or simply tags, are a way to mark definitions of any type with a consistent tag for organizational purposes. Tags can be public or private. Public tags are associated with definitions and remain a part of those definitions during export, import, and share repository operations. Private tags are primarily for use with team configuration functionality, applied to definitions whenever they are changed, and are not included in any export, import, or share operations. Private tags are stripped from the definitions in the local project before they are committed to a share revision or exported to an Agentry export file.

Public Tags

Public tags are created and maintained by the developer within the Agentry application project. The developer can create as many tags as needed for the project, and a given definition can have multiple tags applied to it. When an application project is exported, committed to a share repository, or when a share repository is created from the local project, the public tags are included in the information written to those destinations. Imports from the export files, or updates from the share repository retrieve the tags for the definitions along with the definitions themselves and are displayed in the local project.

By default, public tags are manually applied to a definition by the developer. When a tag is applied to a definition, it can be applied to just that definition, or recursively applied to the selected definition and its descendents. Public tags can be removed from any definition that currently has a tag, and can be recursively removed from the definition and all its descendents that have the same tag.

As an optional behavior it is possible to set preferences in Eclipse to automatically apply one or more tags to any definition modified by the developer. Within the preference page "Tagging Configuration" for Agentry, one or more public tags can be selected for auto-tagging. This results in the selected tags being applied to any definition modified by the developer in any way. This continues until the auto-tagging is disabled for the previously selected tags.

Auto-tagging can be a useful feature when implementing a feature set or custom functionality in an existing product or previously deployed application. A tag can be created to mark those definitions that have been modified or added specifically in support of the new functionality. During subsequent export operations it is then possible to select these definitions by their tags, creating an export file containing just the definitions with the selected tag.

Private Tags

Within a local Agentry application project, there can be one designated private tag. The private tag is stripped from the definitions before they are committed to the share or before they are

exported to an Agentry export file. Private tags are primarily intended for use with the Team Configuration functionality.

The project's private tag is automatically applied to definitions when they are modified and only when the project is connected to a share repository. The private tag is used by the Agentry Editor during commit operations, with the definitions containing the private tag being those compared to the share revision to determine if there are differences.

Note that a definition with a private tag does not guarantee it will be committed to the share repository. If a definition is modified in such a way that at the time of commit it exactly matches the same definition in the share's tip revision, the local definition will not be committed to the share.

As an example, if a developer modifies the minimum length of a the string property City in the Customer object to a value of 5 and then commits, a new tip revision is created in the share. If a second developer that has not yet updated the local project to this new tip revision makes the same change to the City property in his or her local project, it will have the private tag applied to it. When the second developer then commits, however, the City property will not be sent to the share as the Agentry Editor recognizes that the two definitions are the same. If this is the only change made to the project, the commit will not proceed. Other changes will be committed if present.

The name of the private tag can be edited within the "Team Configuration" Agentry preference page. By default, if no name is specified, the default private tag name is *username*Changes, where *username* is the Windows user ID of the developer.

The private tag cannot be manually added to or deleted from definitions. If the Agentry application project is not currently connected to a share repository, there is no private tag available.

Tagging: Creating New Public Tags

Prerequisites

The following items must be addressed prior to performing this procedure:

• The Agentry application project must be open in the Agentry Perspective.

Task

This procedure describes how to create new public tags within an Agentry application project. When complete a new public tag will exist within the project and be applied to that project's definitions.
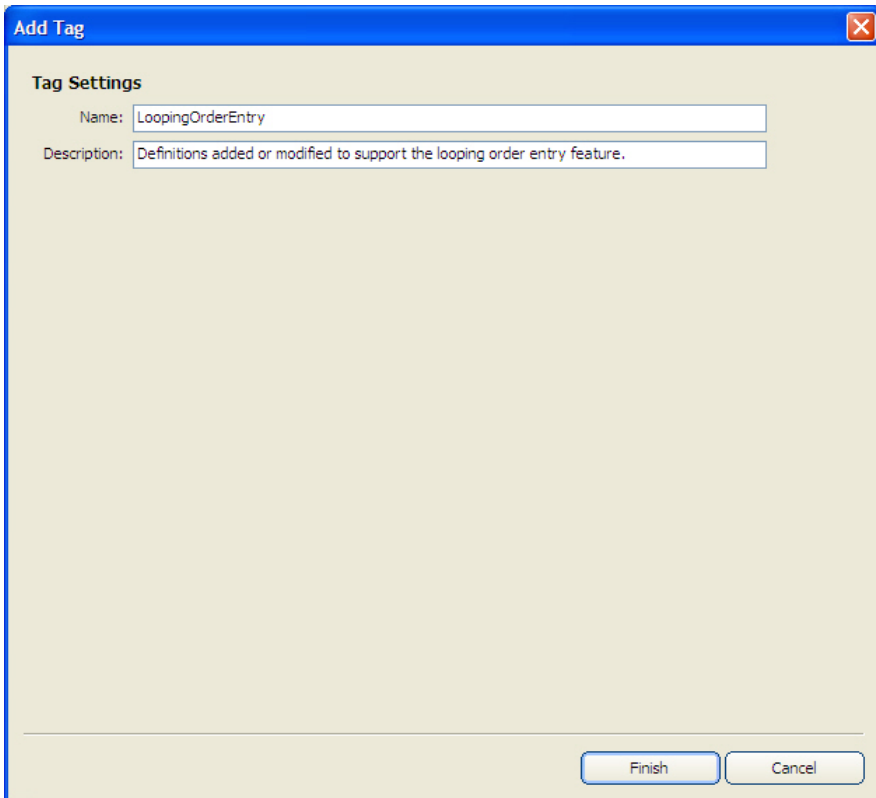
1. Begin by clicking the tag button in the Properties View for any definition.

   This displays the Tag Browser screen listing all current public tags for the project. This can also be used to edit an existing tag or delete a tag from the project:
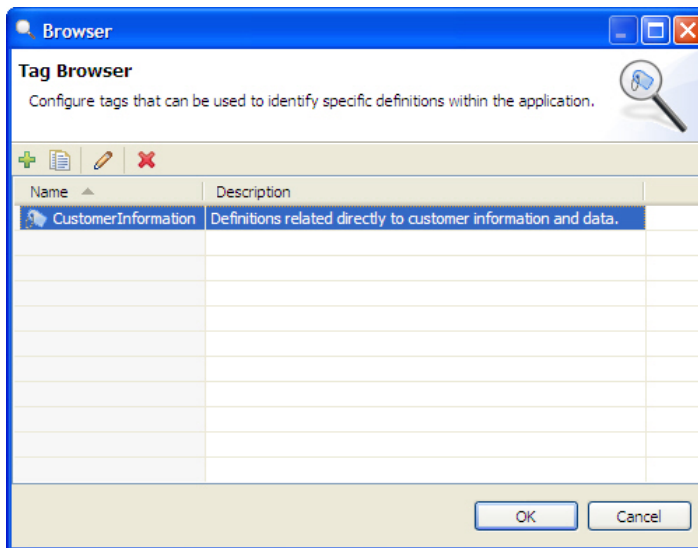
2. Click the add button above the list of public tags.

This displays the Add Tag wizard screen:

Add Tag

Tag Settings

Name: LoopingOrderEntry

Description: Definitions added or modified to support the looping order entry feature.

Finish   Cancel

3. Enter a name and description for the new tag. Click the **[Finish]** button when complete to create the new tag for the project.

The tag is added to the project and listed in the Tag Browser:
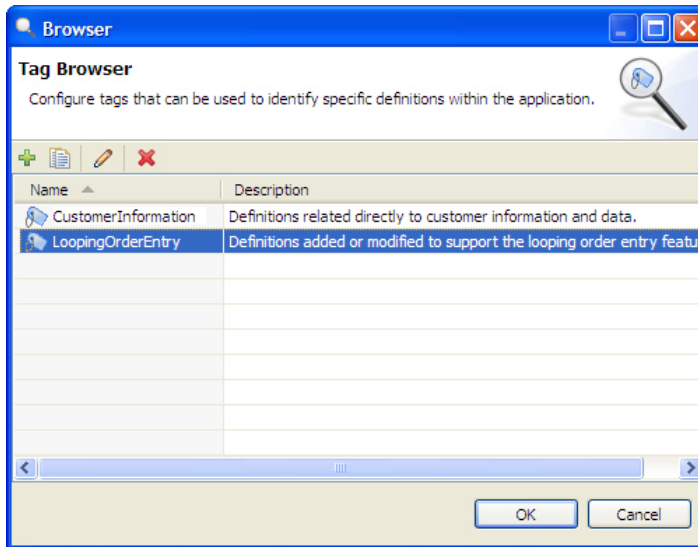
4. To immediately add this tag to the currently selected definition in the project, double-click it in this list. To close the Tag Browser screen click the **[OK]** button.

A new public tag has been added to the Agentry application project. This tag can be applied to definitions manually or via the auto-tagging feature.

Next

The tags name and description can be edited by returning to the Tag Browser at any time and editing the selected tag in the list. Edits update all definitions to which the tag has been previously applied. The tag can be deleted from the project in the tag browser, removing it from all definitions to which it was previously applied.

Tagging: Applying Public Tags to Definitions

Prerequisites

The following items must be addressed prior to performing this procedure:

- The Agentry application project containing the definitions to be tagged must be open in the Agentry Perspective.
- The tag to apply must exist within the Agentry application project.
- The tag cannot currently be applied to the definition.

Task

This procedure describes the steps necessary to apply a public tag to a definition within the Agentry application project. It also describes the process of recursively applying the same tag

to multiple definitions. When this procedure is complete the definition(s) will include the selected tag and can be organized or selected in various operations by this tag.

1. Display the definition to be tagged in the Properties View. Click the down arrow for the tag button.

   This displays a menu of options related to tagging the current definition and includes a list of all public tags for the current project:
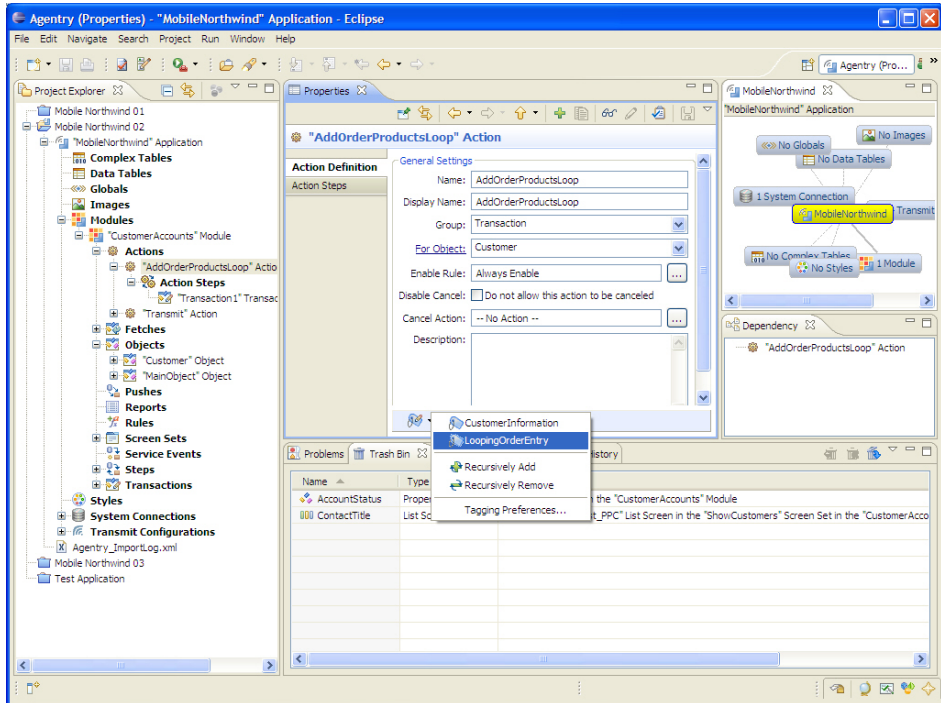
2. A tag can now be applied to the current definition, or to the current definition and all its descendents.

   a) To apply a tag to the current definition only, select it in the menu.

      The tag is now applied to the current definition and displayed in the Tag Bar at the bottom of the Properties View.
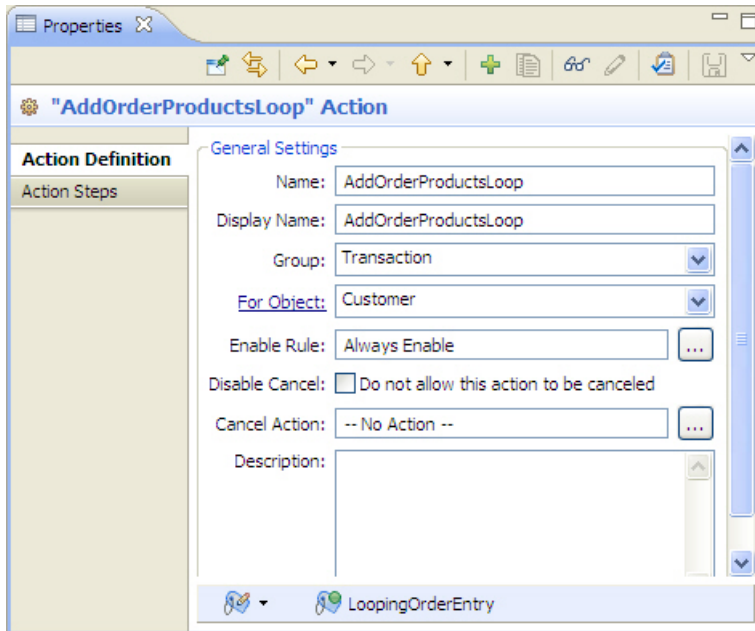
b) To apply a tag recursively to the current definition and all its descendents, select the menu item **Recursively Add**.

This displays the Tag Definitions screen listing the available tags on the left and the definitions under the current definition. Below the list of definitions is the check box to Select all dependencies of the current definition. Selecting this option will tag not only all descendents of the current definition, but also all definitions dependent on it, i.e., those definitions that reference it in some way. Note that this same screen is displayed when choosing to recursively remove tags:
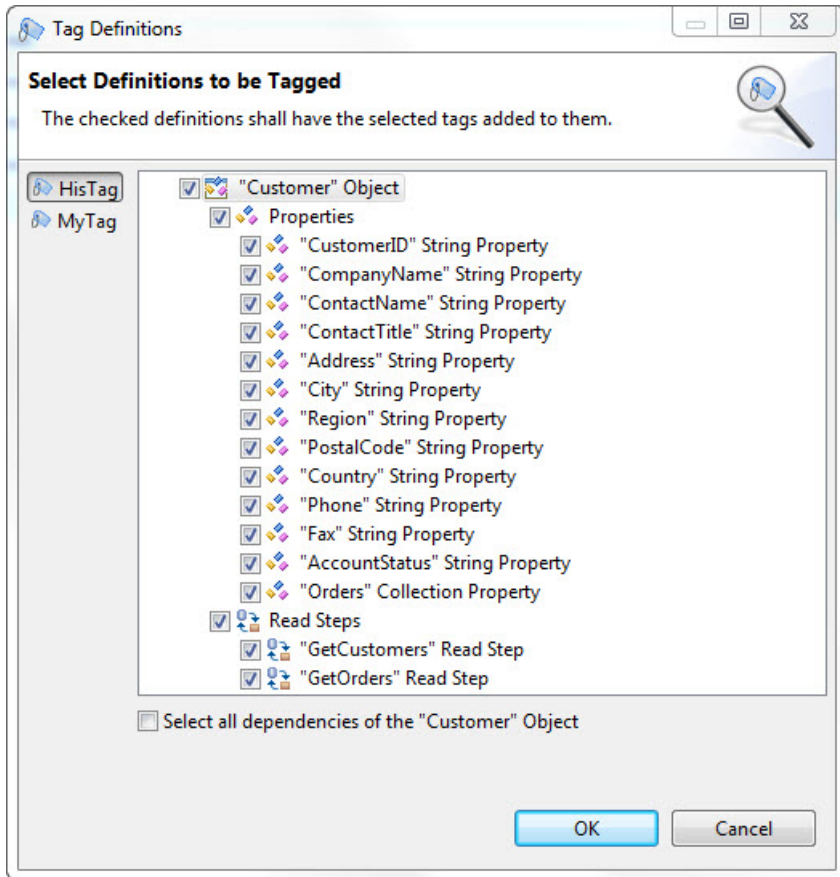
c) Select the tag to apply on the left of the screen. In the tree control, the default selection is the current definition and all its descendents. Definitions can be unchecked to not apply a tag to that definition. Once the tag and definitions are selected, click the [OK] button to apply the tag recursively.

The selected tag(s) is applied to all the selected definitions. The Properties View for all selected definitions now displays the tag in the Tag Bar at the bottom of the view:
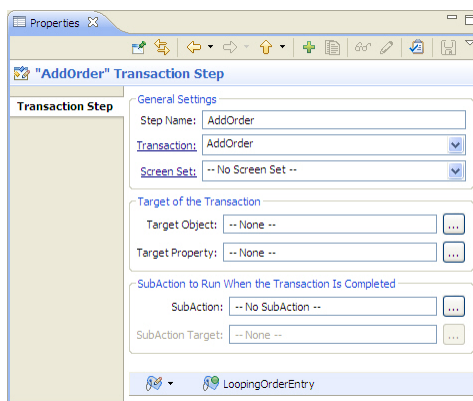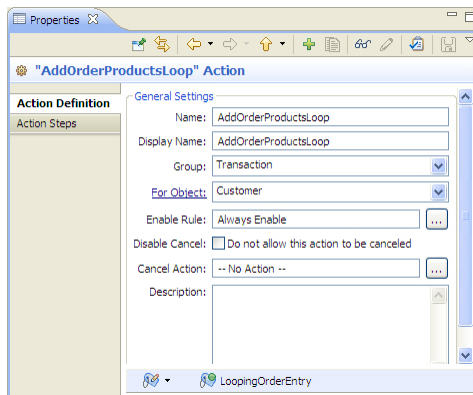
The selected definition or definitions now contain the selected public tag or tags. These public tags are displayed at the bottom of the Properties View for all affected definitions.

Next

The tags can be selected in export operations to select the definitions to export by public tag. Commit operations to a share repository will include this tag information.

Introduction to Team Configuration

With the release of the Agentry Mobile Platform version 5.2 a new feature set collectively called Team Development has been implemented. These features are provided to support multiple developers performing work on the same application project. These features include the following:

- A common share repository for storing work from multiple developers and capable of tracking multiple revisions of a given project
- New import behavior related to the share repository

- Extended export functionality supporting the export of definitions below the module level
- Definition tagging for various organizational purposes
- Replacement of the Compare Dialog with the Compare View, which includes additional functionality related to the share repository

The general approach of team development is to provide a central share repository to all developers working on a common application project. Each developer then creates a project within the local Eclipse workspace based on the contents of the share repository. Developers then modify their local versions of the project and periodically commit their changes to the share, and also update their local projects from the share repository.

In support of this workflow, several new operations have been added to the Agentry Editor specifically for working with the share repository. Additionally the import operation has been augmented to support working with a share repository. Also, the concept of definition tagging has been added to the Agentry application project and Agentry Editor. The Agentry application project itself can be connected to a share, which then allows for the tracking of changes and revision history, and the share repository operations to update and commit from and to the repository.

The following sections provide overview information on the different functional areas and concepts related to the team configuration behaviors. Each of these is covered in more detail in subsequent sections, including instructions and requirements where applicable.

Share Repository

At the center of the team development architecture is the share repository, or simply "share." This share is placed in a location common to and accessible by all developers on a team. Changes can be committed to this share from each developer's local application project and from this share other developers can then update their local projects to retrieve committed changes.

The location of the share repository must be one that is accessible to all developers on a team, and should also be one that is backed up in some manner, usually via a version control or source control system. This location is typically a file server common to all members of the development team and to which each member has read-write privileges.

Stored within this repository is the common project share by the development team. Each developer has a local Agentry application project within the Eclipse workspace created from this share. Developers can work with the local project, defining behaviors just as with any project. When a stable point is reached in the development work, the developer commits the changes made back to the share. From here, other developers update their local projects from the share. The share itself maintains multiple revisions of the project, one for each commit. From this share a developer can update to the latest, or "tip" revision, as well as to earlier revisions when necessary.

Share Repository Compression

The compression of the share repository is a behavior added in version 6.0 of the Agentry Mobile Platform. This is the standard behavior of both creating a new share repository, and

when committing new revisions to an existing repository. For share repositories created with a version of Agentry prior to 6.0, the existing revisions in the repository will remain uncompressed and subsequent revisions to it will be compressed. Once such a revision is committed, the entire repository can only be used by versions of the Agentry Editor (6.0 and later) that support compressed share repository revisions.

Share Repository Operations

Operations related to the share repository are begun by right-clicking the root project node in the Project explorer view. This must always be the open Agentry application project. In the context menu displayed there is a sub-menu **Team**. Within this menu there are several operations related to the share repository. If the open project is not currently connected to a repository, the only option available is to create a new share. This operation should only be performed when creating a new share for other developers to connect with, typically at the beginning of the project.

If no project is currently connected to a share, then a new project can be created from a share using the Import operation, which is separate from those in the Team menu. This is the same Import operation as others previously available. The selected import source is a share repository available to the Agentry Editor.

Once a project is connected, the Team sub-menu for the open project includes several options. Note that the first menu item is Apply Patch... This item is not a part of the team development functionality for an Agentry project and does not apply to these discussions.

The remaining menu options do apply and include the following:

- **Commit:** A commit operation updates the local Agentry project to the share. Changes between the local and share projects are determined, and the new or changed definitions from the local project are updated to the share. Likewise, definitions removed from the local project are removed. A new revision is created in the share as the tip revision for others to retrieve via an Update operation.
- **Revert:** The revert operation allows for the local project to be reverted to an earlier version of the project within the repository. Note that a commit of any local changes must be performed before the local project is reverted. When a revert operation completes, the local project matches the selected revision in the repository.
- **Update:** The Update operation updates the local project to the tip revision in the repository. As a part of this update process checks are made for conflicts between the local project and the tip revision in the share. When conflicts exist the developer is informed and given options on resolving them.
- **History:** The history menu item opens an additional History View within the Agentry Perspective. This view lists all revisions within the repository. Each item in the list includes the revision number, the description entered when that revision was committed, and the author and date and time of the commit.

- **Disconnect:** The disconnect operation removes the link between the local project and the share. Changes made subsequent to a disconnect operation are not tracked in relation to the share. Updates and commits can no longer be performed.

Update Conflicts and Resolution

When working with a share repository the possibility exists that changes committed to that share and changes made by individual developers conflict with one another. These conflicts become prevalent when a developer attempts to update the local project from the tip revision in the share.

The processing logic within the update operation includes checks for such conflicts. These items are then either resolved automatically by the update operation or are noted by the operation and the developer is informed. In the latter case, the developer is presented with options to resolve the issue. The specific options depend on the nature of the conflict.

Whether the conflict is handled automatically or via manual intervention by the developer, the goal of the operation is to update the local project to match the tip revision, or to modify it in such a ways that the next commit performed by the developer updates the share so that no further conflicts exist.

Import

The import operations have been augmented in the 5.2 release of Agentry with the addition of a new import source that is the share repository. This source is selected in an import operation to create a new local Agentry project based on the tip revision of a selected share. When completed a local project is added to the Eclipse workspace that matches the tip revision in the share repository.

Within the import wizard the share repository is selected. Note that this operation always creates a new project. Imports cannot be performed from a share to an existing project. Rather, this operation requires the developer to perform an update operation for a project already connected to a share.

Tagging Definitions

Any definition within the application project can have one or more tags added to it. Tags are created by developers within the project and include a name and description. Tags do not affect the mobile application at run time.

Certain operations, such as exporting, allow for definitions to be selected by the associated tags. As an example, if implementing a new behavior in an existing project, the definitions added or modified to support that behavior can all be given the same tag. During an export, the developer can easily select all these definitions to be exported by simply selecting that tag, which is displayed in the export wizard. In support of such behavior it is possible to enable auto-tagging. This feature allows the developer to select one or more tags to be automatically applied to definitions are modified or added.

The impetus for implementing tags in the 5.2 release of Agentry is in support of the team development functionality. However, they are not tied solely to this functional set and can be used in any manner found useful by the developer.

Team Configuration: Share Repository Requirements and Operations

The share repository, or simply "share", is the central component to the Team Configuration functionality available as of the 5.2 release of Agentry. The share is the common project storage location of work performed by all developers for a single Agentry application project. When working with a share the Agentry project must be connected to that share. This then links the project with the share, tracking the changes made locally with the project as it exists in the share repository.

Share Repository Requirements and Details

The basic requirements for a share repository are that it be stored in a location to which all developers on the team have read-write access to that location. Typically this is a common file server or equivalent that is accessible to all developers and is linked to each developer's workstation as a mapped network drive in Windows. The directory in which the share is placed must exist prior to sharing the project. Multiple share repositories cannot be created in the same base directory. However, multiple shares can have a common ancestor directory.

As an example, a share can be created in the directory `M:\SharedProjects\MobileNorthwindCRM`. A second share can be created in the folder `M:\SharedProjects\MobileNorthwindInventory`. However, it is not allowed to create two shares in the directory `M:\MobileNorthwindApps`.

Each developer will perform operations related to the share that include reading from and writing to the share's directory, and therefore each must have permissions to perform these operations on all files within the share location.

When a share is created (see "Creating a Share Repository" for details) a local project is first selected. A directory is then selected to store the share. Within this directory a file named `share.ini` is created containing information about the share. This file should never be manually modified unless directed by a Syclo support specialist. When checking out from a share to create a new local project based on that share's tip revision, the `share.ini` file is selected as a part of that operation.

The initial revision in the share will then be the definitions in the selected local project. These items are written to the directory named `1`. Subsequent commits to this share for developers create additional directories, each numbered to match the share revision created by that commit operation, e.g. `2, 3, 4`.... As with the `share.ini` file, the contents of these sub-directories should never be modified manually unless under the specific direction of a Syclo support specialist.

Share Operation: Share Project

The Share Project operation is the first step in creating a team environment for a common Agentry application project. This operation creates a share repository at a designated location. The new share contains a single, initial revision, i.e., revision 1. The contents of this revision match the contents of the local Agentry application project open within the Agentry Perspective when the share project operation is executed. This operation can only be executed on Agentry application projects not currently connected to a share repository.

The new share should be created in a location common to all developers on the team and according to the requirements of the share repository. The local project should be in a state in which it makes sense to share the project contents. This state will vary from one project to the next and depends on the division of work among the developers. The only requirements from a technical standpoint are that an Agentry application project exists within the Eclipse workspace and that project is open. No validation or check on publish is performed as a part of the share project operation. This means the project need not be in a publishable state prior to creating the share.

In practice, it is likely desirable that some useful functionality exist prior to creating the share. In many use cases the first revision of the share is the standard implementation of a product application, such as those provided by Syclo. For new application projects the functionality need not be nearly as robust, or even completely implemented before creating the share. Rather, the core pieces to the project, such as objects and their properties, fetches and pushes that may or may not yet contain step usage definitions, and screen sets with or without platforms or screens may all be a part of the initial revision of the share when created.

Typically when planning and creating a share, the developer responsible for creating these core definitions should perform their initial work (though it need not be the final planned result) and then create the share from their local project. This developer, as well as the rest of the team, can continue to work with the definitions once the share is created.

Share Operation: Checkout (Import from Agentry Share)

Once a share is created, other developers can access its contents for their own portion of the work for the project. To begin this work the developers will each need to check out the tip revision of the share repository. This is performed via the Import operation. During this operation the import wizard is displayed, the first screen of which provides the developer with the list of choices for the import source. One of these options is **Agentry Share | Checkout Project from an Agentry Share (share.ini)**. Selecting this option indicates a new project is to be created in the local Eclipse workspace by checking out the tip revision from the share repository.

Once the share is selected as the source of the import, and other information is provided, the import operation creates a new Agentry application project within the workspace by importing the definitions within the share's tip revision. When the operation is complete, the developer can modify and extend this project for their portion of the overall implementation. Typically a checkout is performed only once to create the local project. After this point, the

developer performs commit operations to commit changes made to the local project to the share; and update operations to retrieve changes committed to the share by other developers.

Share Operation: Commit

When working with a project connected to a share, the developers on a team must perform the commit operation to commit changes made in their local projects to the share repository. A commit operation results in the addition of a new revision to the share. This new share then becomes the tip share that other developers receive when performing updates until a subsequent commit is performed by any developer connected to the share.

During the commit operation, the wizard screens displayed include a comments field. Within this field comments are automatically added to note all changes made to the local project. The comments reflect the type of change made, which can be add, edit, or delete, and the definition modified. These default comments can be edited prior to performing the actual commit. The final contents of this comment field are then the comments for the revision created by the commit, and will be viewable by all developers in the History View.

When performing a commit, the Agentry Editor first checks the local project for changes as compared to the share's tip revision. If no changes exist, the commit will not be performed. The commit wizard's OK button is disabled. The summary view within this wizard indicates no differences exist between the local project and current tip revision.

Another of the share operations is revert. Using this operation it is possible to revert the local project to a share revision earlier than the tip revision. When the local project is reverted to a previous revision and subsequent changes are made to the local project, a commit operation will display a warning message indicating the difference in revisions. Note that the current state of the local project will be committed to the share as the new tip revision. Any changes made and committed to the share between the reverted revision and the current tip revision will be lost when the new tip revision is committed. For this reason, reverting to a previous revision and then committing should only be performed in rare circumstances.

Share Operation: Update

The update operation is performed by the developer to update the local Agentry application project to the tip revision of the share repository. This allows the developer to retrieve changes made by other developers working on the same project. During an update a check is first made for differences between the share revision and the local project. If changes exist for the same definition in both the local and share projects, a conflict exists. This requires manual resolution by the developer. The specific behavior of the update and the resolution depend on the nature of the conflict. When a conflict does occur, and there are other definitions in the share that should be imported that are not in conflict with the local project, those definitions are updated to the local project, leaving only the conflicted definitions in need of resolution. See the information on "Update Conflict Resolution" for details.

Share Operation: Revert

The Revert Operation for a share repository replaces the local project with the specified share revision. This revision can include the tip revision when the local project is at an earlier

revision. The difference between a revision and an update is the, first, a specific revision can be selected, and, second, there is no conflict detection performed. This last revert behavior is important to note as it means that any uncommitted changes made to the local project are lost when the revert operation completes. One use for the revert operation can be to remove unwanted changes from the local project.

Note that while the tip revision can be selected in a revert operation it should never be used in place of the update operation. Reverting to the tip revision should only be performed when it is desired to remove all local changes. In such situations the developer should be careful to verify all local changes should be removed before proceeding.

As an option to a revert operation it is possible to create a new local project based on the selected share revision. This can be useful when wanting to branch development from an earlier revision of the repository for a separate development effort. In such a situation, the proper overall procedure is as follows:

1. Execute the revert operation, selecting the earlier revision from the share, and selecting the option to create a new local project.
2. The new project is connected to the share repository from which it was imported. Disconnect from this share.
3. To support team development with the new local project, create a new, separate share by performing the share project operation.

Share Operation: Show History

The Show History operation does not affect either the local Agentry application project or the share repository. This operation opens the History View within which each repository revision is listed. The revision of the repository from which the last update to the local project was performed is highlighted.

Within this view the revision number, the date it was created, the user that created it, and the revision description as entered during the commit operation are listed. This is a read-only view intended to provide information about the local project as it relates to the share, as well as information about the share itself. This view can be refreshed at any time and will display any new revisions added to the share since the last refresh.

The history view should be reviewed prior to performing commit or update operations to understand the current state of the share before changes are made to it or the local project.

Share Operation: Disconnect

The Disconnect Operation disconnects the local Agentry project from the share repository. Once a project is disconnected it is no longer tied in any way to the share. Subsequent changes to the local project are made out of synch with the share. Such changes are not privately tagged by default.

The disconnect operation should only be performed on a project that should no longer be a part of the team efforts. This may be useful when it is desired to retrieve a project from an existing share by performing a checkout but for which changes to that project should not be included in the share. This procedure would involve the following steps:

1. Perform an import from an existing share repository, create a local Agentry application project, which is currently connected to the share. The project contains the tip revision from the share.
2. If a previous revision from the share is desired, revert the local project to that revision.
3. Once the desired share revision has been imported into the local project, disconnect the local project from the share.
4. Optionally, create a new share from the local project to support team efforts.

This is only one scenario for disconnecting a project from a share. Others may exist, and the operation can be performed to meet any needs found by the developer.

Update Conflicts and Conflict Resolution

When performing an update operation the local project is updated to match the tip revision of the share repository. When the update is performed, however, it is possible the state of the local application project is in conflict with the state of the share's tip revision. This occurs, potentially, when the same definition is deleted or modified in some manner in both the local project and share.

During the update operation, the Agentry Editor checks for any conflicts. If found, it may be necessary for the developer to manually resolve these conflicts. This manual resolution is performed in the Comparison View in the Agentry Perspective. This view is displayed whenever an update operation is performed. If a conflict is found, a message is displayed after the update is finished indicating there is an issue. Any definitions in the tip revision not in conflict with the local project are imported. Those in conflict are not imported, but rather are highlighted as differences in the comparison view.

This manual conflict resolution is similar in behavior to a compare and import from some other import source, such as an export file. The local project and the share's tip revision are displayed in the comparison view, with the share on the right side as the comparison source. The developer can then select the individual definitions within this view to be imported from the share, or leave them as is within the local project. This selection can be different for each definition found to be in conflict.

When the conflict resolution is completed by the developer, the definitions selected in the share are imported into the local project. Any conflicted definitions not imported from the share are left unchanged in the local project. At this point a commit can be performed and those definitions not imported from the share are committed to the share as a part of the new revision.

The nature of the conflict and how it should be resolved depends on the type of changes made to the definitions in both places. The following list describes the potential conflicts requiring manual resolution by the developer. This list is then followed by sections describing the resolution choices for each:

- **Share definition edited - local definition edited:** In this situation, both the share and local definitions' attributes have been modified in some manner. Since the nature of the change

to both may not be compatible in one manner or another when merged, this change is left to the developer to resolve.

- **Share definition deleted - local definition modified:** If the local definition has been edited by the developer, and the same definition is deleted from the share revision, the developer must specify whether to keep the local definition, or to import the deleted definition from the share.
- **Share definition deleted - local definition modified and deleted:** If the share definition has been deleted (contained in the trash bin) and the local definition was modified and then deleted, a conflict exists. While the share definition is simply added to the trash bin if brought down during the update, there is a question as to which version of this definition should be stored in the trash bin for possible later recovery, the local version or the share version. Therefore, the developer is required to make this selection.

Share Definition Edited - Local Definition Edited

If both the local definition and the definition in the share have been edited, a conflict exists and must be resolved manually. Edited share and local definitions includes changes to the definitions' attributes, whether or not it is the same attribute. Changes to child definitions are not considered conflicts.

When such a conflict occurs, the developer must manually specify in the Comparison View which definition to keep. If the share definition is selected in this view, it will be imported into the local project, replacing the local definition. If the share definition is not selected, the local definition will remain. During the next commit from the local project, the local definition that was in conflict will be treated as a changed definition and committed to the share.

Share Definition Deleted - Local Definition Modified

If the definition in the share has been deleted, and the local definition has been modified, a conflict exists. Any change to any attributes in the local definition constitute a change. The share definition is considered deleted if it has been removed but remains in the trash bin. Permanently deleted definitions in the share repository, that is, those that have been removed from the trash bin, and those that have been edited in the local project are not conflicted.

Share Definition Deleted - Local Definition Modified and Deleted

If the share definition has been deleted and resides in the trash bin, and the local definition has been modified and then subsequently deleted prior to committing, a conflict exists. In this situation, the developer must compare the two definitions and determine which should reside in the local trash bin.

If the share definition is selected in the Comparison View, the local definition in the trash bin is replaced with the share definition. Otherwise the local definition remains in the trash bin and will then be added to the next commit performed from the local project.

Automatically Resolved Conflicts

Additional conflicts may occur during an update from the share's tip revision that will not require intervention on the part of the developer. These may not even be considered conflicts,

but rather the behavior of the update operation under certain conditions other than when the local definition has not been modified and the share definition has.

The following list describes these situations and the resulting behavior of the update:

- **Share definition deleted in trash bin - local definition not modified:** If the share definition has been deleted and currently resides in the trash bin, and if the local definition has not been modified in any way and matches the share definition, the local definition is removed from its parent and placed in the trash bin.
- **Share definition deleted - local definition deleted, not modified:** If the share and local definition have both been deleted, and if there is some difference with the local definition that is not tagged, meaning it was not made since the last commit, the share definition will replace the local definition in the trash bin.
- **Share definition deleted - local definition does not exist:** If the share definition is deleted and resides in the trash bin, and there is not corresponding local definition, the share definition is added to the local project and resides in the trash bin.
- **Share definition does not exist - local definition exists, not modified:** If the local definition exists and has not been modified, and the share does not have a corresponding definition, the local definition is not modified or removed. This is listed as a difference in the Comparison View and can be removed here or by simply deleting the definition. If the definition is not removed from the local project it will be added to the share during the next commit operation.

Share Operation: Creating a Share Repository

Prerequisites

The following items must be addressed prior to performing this procedure:

- The directory in which the new share will be placed must exist prior to creating the share.
- The directory for the share must be empty.
- The directory should be in a location accessible by all developers on the team. This includes both read and write access to the directory and its contents.
- The Agentry application project from which the share is to be created must be the open project in the Agentry Perspective.
- The Agentry application project should be in a state where it makes sense to share with other developers on the team.
- The Agentry application project cannot be connected to a share.

Task

This procedure describes the steps involved in creating a new share repository. When this procedure is complete, a share will be created with the initial revision. The contents of this revision will match the current state of the Agentry application project from which the share is created. This local project will be connected to the new share. The share will be available to

others for import to create local projects based on the initial revision. Going forward the share will support all share operations by developers with access to the share.

1. Open the source Agentry application project in the Agentry Perspective within Eclipse.

2. Right click the root project folder in the Project Explorer View. In the context menu select the item **Team | Share Project...**

This displays the Share Project Wizard:
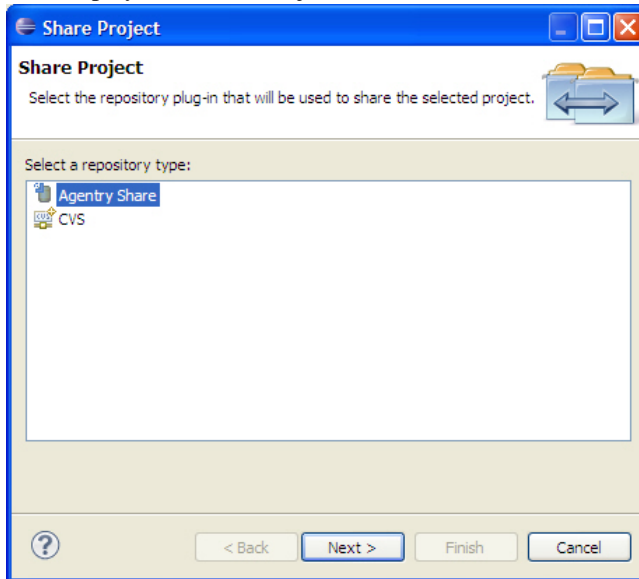
3. Select the item Agentry Share from the list displayed in this screen. Click the [Next >] button.

This displays the Share Location screen:
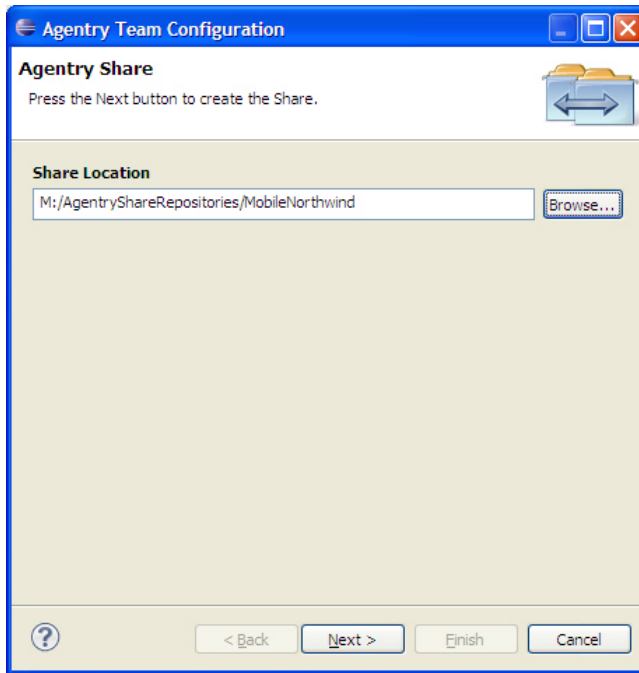
4. Enter the location of an existing directory accessible to the developers on the team and meeting the requirements of an Agentry Share Repository. Click the **[Next >]** button.

This creates the new share repository at the designated location. The status screen displays the definitions as they are added to the initial revision of the share:
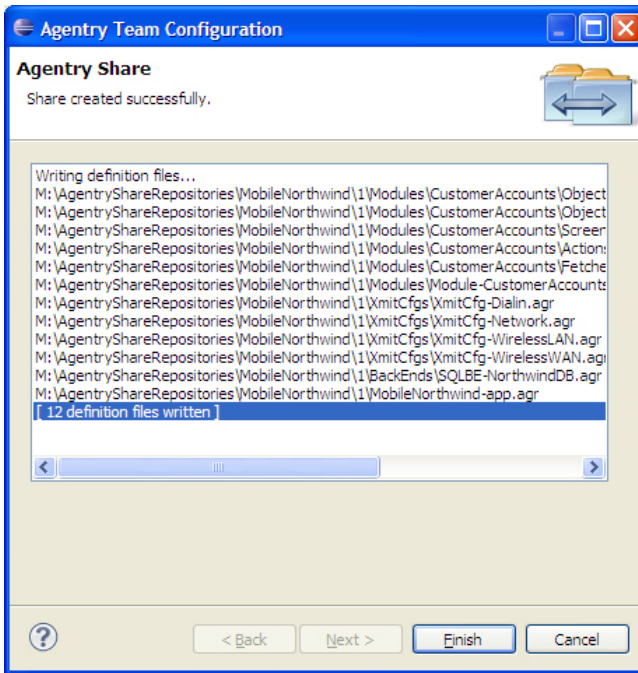
5. Click the [Finish] button to close the wizard and return to the Agentry Perspective.

When this procedure is completed, the new share repository is created. The initial revision of this share matches the current open project in the Agentry Perspective. This local project is connected to the newly created share repository. The share is now available to other developers with read-write privileges to the selected location from their Agentry Editors.

Next

Developers with access to the share can import the initial revision, or the current tip revision, creating local Agentry application projects. Work performed by all developers can then be committed to this share, making it available to all others on the team.

Share Operation: Checking Out (Importing) From a Share

Prerequisites

The following items must be addressed prior to performing this procedure:

- The Agentry Perspective must be open in Eclipse.
- The developer performing the check out must have read-write privileges to the share location.

Task

This procedure describes the steps to check out the tip revision of a share repository, creating a local Agentry application project in the current Eclipse workspace. When this procedure is completed, a new Agentry application project will exist in the workspace and will be connected to the share from which it was checked out. This procedure is similar to an import performed with a non-share source, such as an export file or published application on an Agentry Server. However, the import source for this procedure is an existing share repository.

1. Right click anywhere in the Project Explorer View within the Agentry Perspective. Select the menu item **Import...**

    This displays the first screen of the Import Wizard:
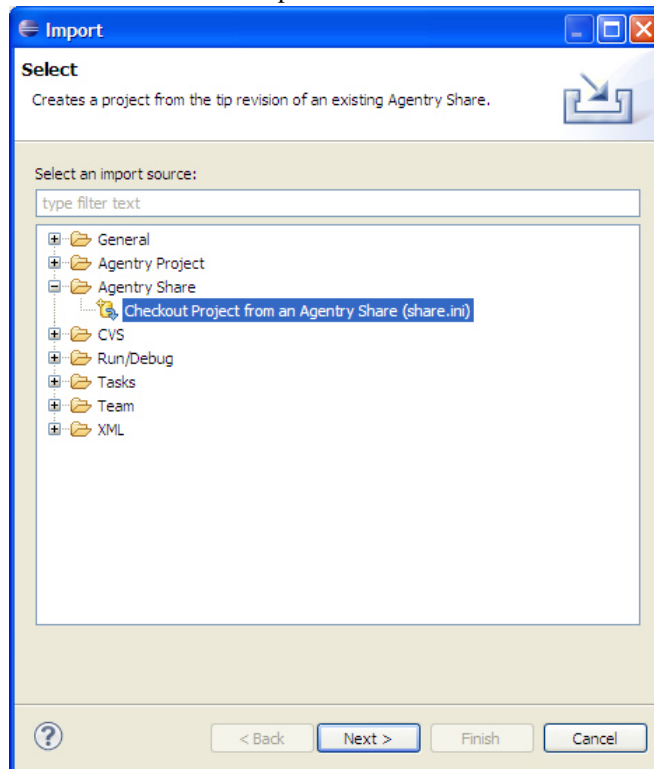
2. Within this wizard screen the import source is selected. To check out from a share, select the item **Agentry Share | Checkout Project from an Agentry Share (share.ini)**. Click the **[Next >]** button.

    This displays the second screen of the wizard:
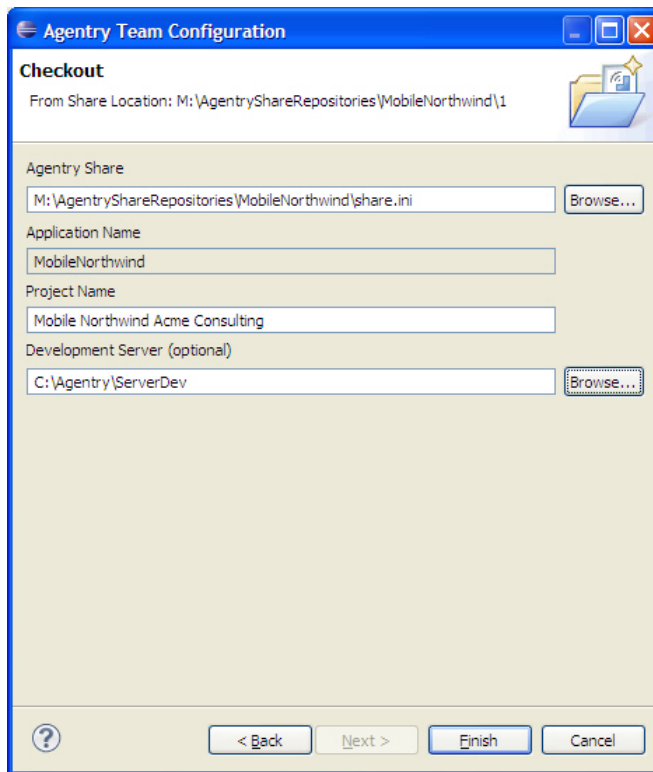
3. Within this screen set the fields according to the following instructions and then click the [Finish] button:

- **Agentry Share:** Enter the path to the share repository, including `share.ini` file. This can be entered manually or selected in a Windows File Dialog by clicking the **[Browse]** button and navigating to the share location and selecting the `share.ini` file.
- **Application Name:** When checking out of a share, this field is read-only. It is set to match the Application Name attribute in the share's tip revision.
- **Project Name:** This is the name given to the project in the Eclipse workspace. This name can be any unique value and is not committed back to the share.
- **Developer Server (optional):** Enter the path to the Agentry Development Server for the local project. This path can be entered manually or selected in a Windows File Dialog by clicking the **[Browse]** button and navigating to the Server's installation directory.

The new project is created by importing the definitions from the selected share's tip revision. The wizard is closed and the project is displayed in the Project Explorer View.

When this procedure is complete a new project is created in the current Eclipse workspace. This project contains the definitions matching the selected share's tip revision. The project is automatically connected to the share from which it was imported.

Next

The new project can now be modified by the developer. Changes made can be committed to the connected share and updates retrieved from it.

Share Operation: Committing Changes to the Share Repository

Prerequisites

The following items must be addressed prior to performing this procedure:

- The current Agentry application project must be connected to a share repository.
- The Agentry application project should be in a state believed to be stable and/or ready for integration into other developers projects.
- Any desired public tags should be applied prior to performing the commit.
- Any additional information needed for the revision about to be created should be noted in preparation for adding such information to the revision's comments during this process.

Task

This procedure describes the steps involved in committing changes made to a local Agentry application project to the share repository to which the project is connected. When complete, all definitions modified in the local project since the last commit will be added to the share repository as a new revision. This revision will be the tip revision of the repository received by other developers during subsequent updates up until another revision is committed.

1. Right click the open project in the Project Explorer View. In the context menu now displayed select **Team | Commit**.

This displays the Commit Wizard:
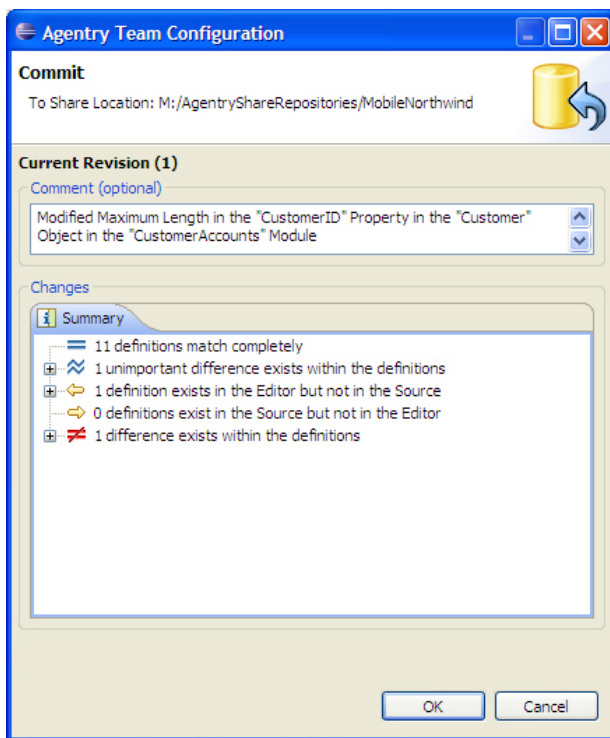
2. This screen displays, first, the comment field for the new revision and, second, the summary of changes made to the local project. Edit the comments as necessary. Click the **[OK]** button to proceed.

3. The definitions that have been modified in the local project are updated to the share as a new revision.

When this procedure is complete a new revision is added to the repository as the tip revision. This revision includes the local Agentry application project definitions as they existed at the time of the commit. Any private tags on definitions resulting from modifying those definitions have been removed from the local project.

Next

When the new revision has been created, other developers can retrieve the changes and integrate them with their local projects by performing update operations.

Share Operation: Updating From the Tip Share Repository Revision

Prerequisites

The following items must be addressed prior to performing this procedure:

- The tip revision within the share repository must be newer than the local Agentry application project's revision.
- The current Agentry application project must be connected to a share repository.

Task

This procedure describes the steps necessary to update the local Agentry application project to the latest, or "tip" revision in the share repository to which the local project is connected. When this procedure is complete, the local project will be updated to match the application definitions in the share's tip revision. As a caveat to this result, if there are conflicts between the tip revision and the local copy of the definitions, it is possible the resulting local copy will differ from the tip revision. These differences can result from how any conflicts are resolved during the update, which can include keeping the local version (which would differ from the share's tip revision) or merging the differences between the local copy and tip revision.

1. Right click on the local project in the Project Explorer View. Select the menu item **Team | Update** from the context menu.

   The update from the share begins immediately. If there are no conflicts with the share and the local project this procedure is complete. The local project is now updated to match the tip revision in the share.

2. If one or more definitions between the share and local projects are in conflict, a message is displayed indicating there is an issue. Also, the Comparison View is opened and the definitions in conflict are highlighted as differences within this view.

   The following message indicates there are conflicts:
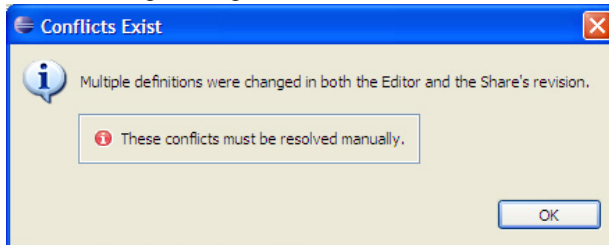
3. Resolve the conflicts using the Comparison View (opened automatically during the update) to manually import from the share or keep the local version of the definition. In the latter case, these local definitions will be a part of the next revision in the share. For further information on conflicts and resolving them during an update, see the information on "Update Conflicts and Conflict Resolution."

When this procedure is complete the local project is updated to match the tip revision in the share. If any conflicts occurred, and those conflicts have been resolved, the local project may contain differences from the share's current tip revision. Those differences will be a part of the definitions committed to the share the next time a commit operation is executed from the local project.

Share Operation: Reverting to a Previous Share Revision

Prerequisites

Prior to performing this procedure the following items must be addressed:

- The current Agentry application project must be connected to a share repository.
- It must be determined if the revision to be reverted to should replace the current project, or if a new project should be created with the previous share revision to be selected.

> **Note:** Selecting to replace the current project with a previous revision can lead to undesirable results, including the loss of all local, non-committed changes in the local project. Replacing the existing project should only be performed after a back-up of the local project has been made. This can be accomplished by either committing the project to the share, or exporting the project to an Agentry export file by executing he Export Operation.

Task

This procedure provides the steps necessary to revert the local project to a previous revision within the share repository; or, alternately, to create a new local project in the current Eclipse workspace based on a revision of the share prior to the tip revision. When this procedure is complete, and depending on the options selected, either the current local project will be replaced with the revision selected in the share repository, or a new local project will exist containing the definitions as they existed in the selected revision.

Note that this procedure can also be performed to revert the local project to the tip revision in the share repository. This differs from an update in that the current local project will be reverted to the tip revision and any local changes will be lost. None of the conflict detection or resolution functionality that is a part of an update operation is performed in a revert operation.

1. Right-click the open Agentry application project in the Project Explorer View. Select the menu item **Team | Revert |** *revision*, where *revision* is the share revision to which the project should be reverted.

   This displays the Revert Project wizard, with the selected revision's information displayed
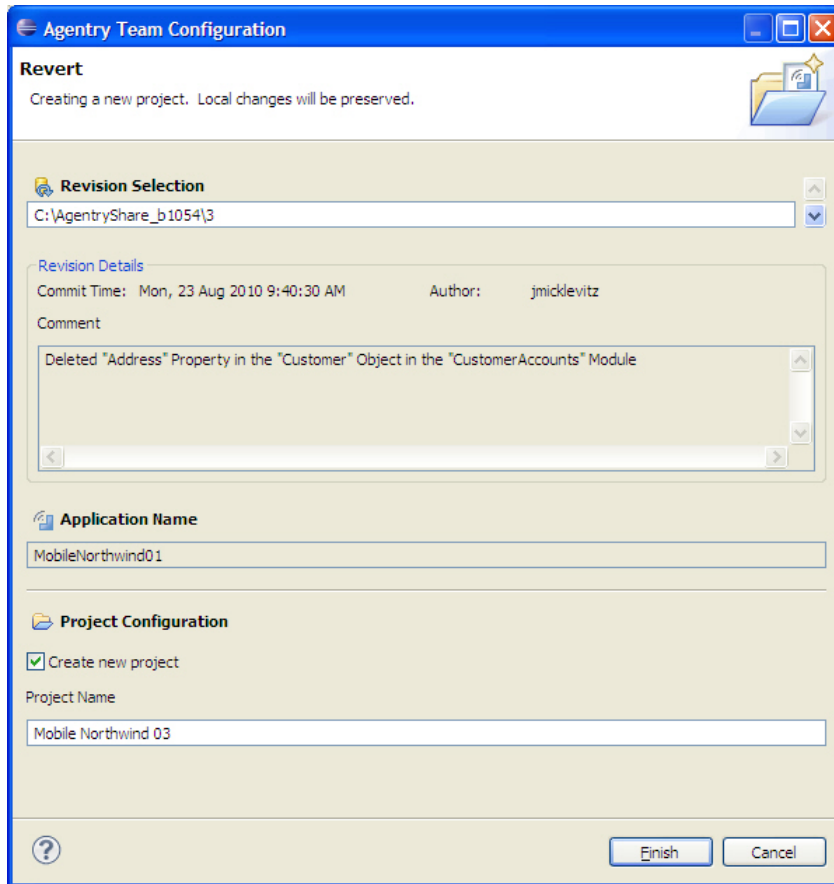
2. To create a new local project based on the selected share revision, check the **Create new project** check box. Then enter a name for this new project in the **Project Name** field. To not create a new project, but instead replace the currently open local Agentry application project, leave this box unchecked. Click the **[Finish]** button to proceed.

The revert operation now executes. Based on the selections made, either the local Agentry project is replaced with the selected revision, or a new project is created in the current Eclipse workspace matching the selected revision from the share.

When this procedure is complete, either the local Agentry project is reverted to the selected revision, or a new local project is created in the Eclipse workspace matching the selected revision. If a new project has been created it is connected to the share repository.

Next

The resulting project from this operation now matches the selected revision within the share. This project cannot be committed to the same share. Either the local project must be updated or

reverted to the tip revision, or it must be disconnected from the current share and new one created for the local project using the Share Project operation.

Overview of Mobile Northwind Sample Application

In the development guide for Agentry applications there is reference to the Mobile Northwind sample application. An overview of this application is provided here, including data structure, client behavior, and synchronization components.

The Mobile Northwind application is a basic order entry application with some light customer relations management and inventory-like functionality included. It is an extension of the Northwind sample database provided with MS SQL Server systems, a database for the fictitious company Northwind Trading.

Module Data Structure and Object Collections
The data definitions within the application include objects and related transactions, a complex table and a data table.

The object definitions include Customer, Order, and Order Item or Product. In the case of the Product and Order Item objects, these terms are interchangeable. The structure, purpose, and usage of the object is the same, regardless of which name is used. These objects are structured within the parent module Customer Accounts in a parent-child relationship:

```
MainObject > Customers > Orders > Products
```

The module main object contains a collection of Customer objects. The Customer object definition in turn contains a collection of Order objects. The Order object then contains a collection of Product objects.

The Customer object encapsulates customers of the Northwind Trading company. It contains property definitions for customer ID, company name, contact name, phone number, and address information.

The Order object encapsulates an order placed by the customer. It includes property definitions for the unique order ID, order date, delivery date, required date, and shipping information.

The Product/Order Detail object encapsulates an individual item ordered by a customer. It includes the unique product ID, product name, description, and quantity ordered within its property definitions.

Complex Tables and Data Tables
There is one complex table and one data table defined within the Mobile Northwind application. The data table contains a short list of shippers. The key field contains the Shipper ID value, and the value field contains the shipping firm name.

The Products complex table contains a list of the items which the Northwind company offers to its customers. There are fields defined for the Product ID, Product Name, Unit Price, and Quantity per Unit values. There is one index on the Product ID and one on the Product Name.

Transactions

Transactions exist for all three object types defined within the project. There is an add and edit transaction for the Customer object. The edit transaction allows the user to edit contact information for the selected customer, including contact name and phone number.

There is an add transaction defined for the Order object. Add is in support of order entry, allowing the user to record a new order for the selected customer.This transaction uses initial value rules on its transactions to set the initial value of all shipping address properties to the matching address properties of the selected customer object. the user can modify these values in the wizard for the transaction if necessary. The transaction also captures date and shipper information.

There is an add, edit, and delete transaction defined for the Product object. Products are added to the selected order object for the customer. Products are selected by the user from the Products complex table.

User Interface

The user interface for the Mobile Northwind application includes screen sets to display customers, orders, and products for orders. They are presented in a basic drill down navigation, with actions defined to go from one to the next.

Transactions are presented in standard wizards. The action to add new orders for a customer includes a looping SubAction step in addition to the transaction step for the add transaction for the order. The looping step executes the action for the Add Product transaction in a loop that continues until the user indicates they are finished. The screen flow from this then presents the screen set for the Add Order transaction once, followed by the screen set for the Add Product transaction being presented in a loop until the user has completed the entry of all desired products for the order.

The fields which capture data for the Add Product transaction are defined with update rules to calculate the total cost of the product order by multiplying the quantity being ordered by the unit price.

Target Paths and the Property Browser

Target paths are an important concept to understand when creating or working with an application project using the Agentry Editor. Target paths are selected using either a short list of likely options from a context menu for a given attribute field, or by using the Property Browser for more sophisticated paths.

To understand target paths it is first important to understand the concepts of target definition instances and referenced definition instances. A target is a definition instance that is affected by some other definition at run time on the client. The specific impact on a target is typically setting a value in that definition. Targeted definitions are almost always a property.

A referenced definition instance is one whose value, or in certain circumstances the definition instance itself, is returned to the definition referencing it. What is done with the value or definition instance returned depends entirely on the referring definition.

A target path is then used to specify the definition instance to be targeted or referenced. Target paths always deal with a specific definition instance. A target path is a path evaluated in the context of the definition in which it is contained at run time on the Agentry Client.

A basic example of a target path is the value of a property to be displayed by a detail screen field. As a part of a detail screen field's definition the property to be displayed by the field is specified. This property is specified using a target path. This path may take the basic form:

```
:>"CompanyName" Property
```

In this basic example, the field is defined to display the property named CompanyName found in the definition being displayed by the detail screen containing the field. At run time this path is evaluated by the Agentry Client and the value of that property at that time is displayed in the field.

Target paths can retrieve data values from numerous definitions, including properties, the current value of a screen field, complex table fields, data table fields, and other definitions. The path itself can be as basic as the previous example, or far more complex and include rule evaluation and currently selected items in a list to determine the desired value. The complexity of the target path depends on numerous factors, including the context of the definition containing the target path and how that context relates to the logical location of the value to be retrieved, the parameters by which the value should be selected (e.g. a currently selected item in a list, a record from a complex table, etc.), and the nature of the definition in which the target value is located. Consider the following more sophisticated target path:

```
:>Main Screen Set>"ShowCustomers_List_PPC" List Screen>Current
Object>"CustomerID" Property
```

This path returns the CustomerID property of the object that is currently selected in the ShowCustomers_List_PPC list screen, which is contained in the main screen of the current module. Such a path would be needed in a situation where the context of the definition needing this value is one where the module main screen set is not a descendent, for example a transaction definition. The above path may be one used to specify the initial value of a transaction property that must be initialized from the currently selected item in the list screen specified. Transactions are child definitions of the module, just as screen sets are. This means the transaction and screen set are siblings and the transaction is, therefore, not a descendent of the screen set.

Creating Target Paths: Attribute Field Context Menus

The first option when setting the target path for an attribute is to use the context menu displayed by the ellipses button for that attribute field. A basic example of this is a transaction property's target **Object Property**. This attribute is common among all transaction properties and specifies the object property whose value is to be set to the value of the transaction property when the transaction is applied.

When setting this attribute, the ellipses button is clicked in the Editor for this attribute to display the context menu. Included in this menu is a list of all properties found in the object definition targeted by the transaction. These items are displayed by the Editor as they are the most likely candidates for selection when defining a transaction property's target:
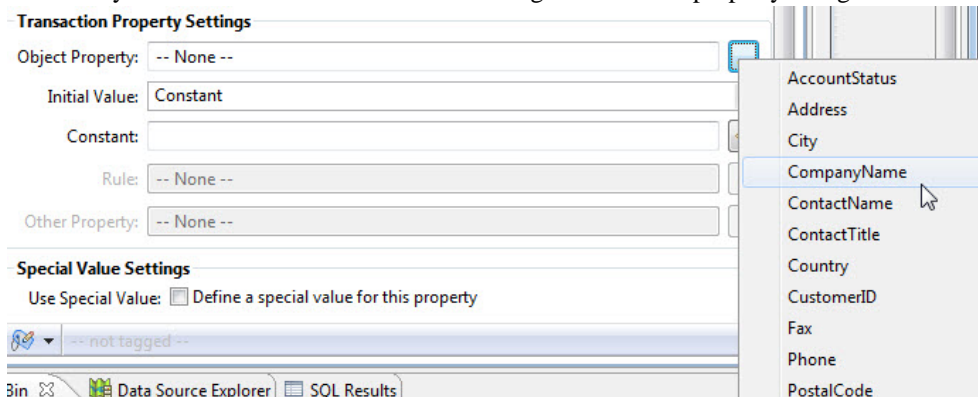
Selecting an item from this menu creates a target path like the first example shown previously for the CompanyName property. Though not created using the Property Browser, this is still a target path. The Editor provides this list of likely options for the attribute as a shortcut to defining the application.

Creating Target Paths: The Property Browser

The selection of more involved paths is made supported by the Property Browser. The Property Browser can be displayed to set any attribute within the application project where a target path is allowed or required. In most cases, the menu displayed for such an attribute will include a list of likely definitions to be targeted by the attribute (based on the current context) and an additional menu item of **Browse...** or **Browse Objects...** This menu item displays the Property Browser, which provides a tree control of the various definitions in the structure of the application project that are valid selections for the attribute:
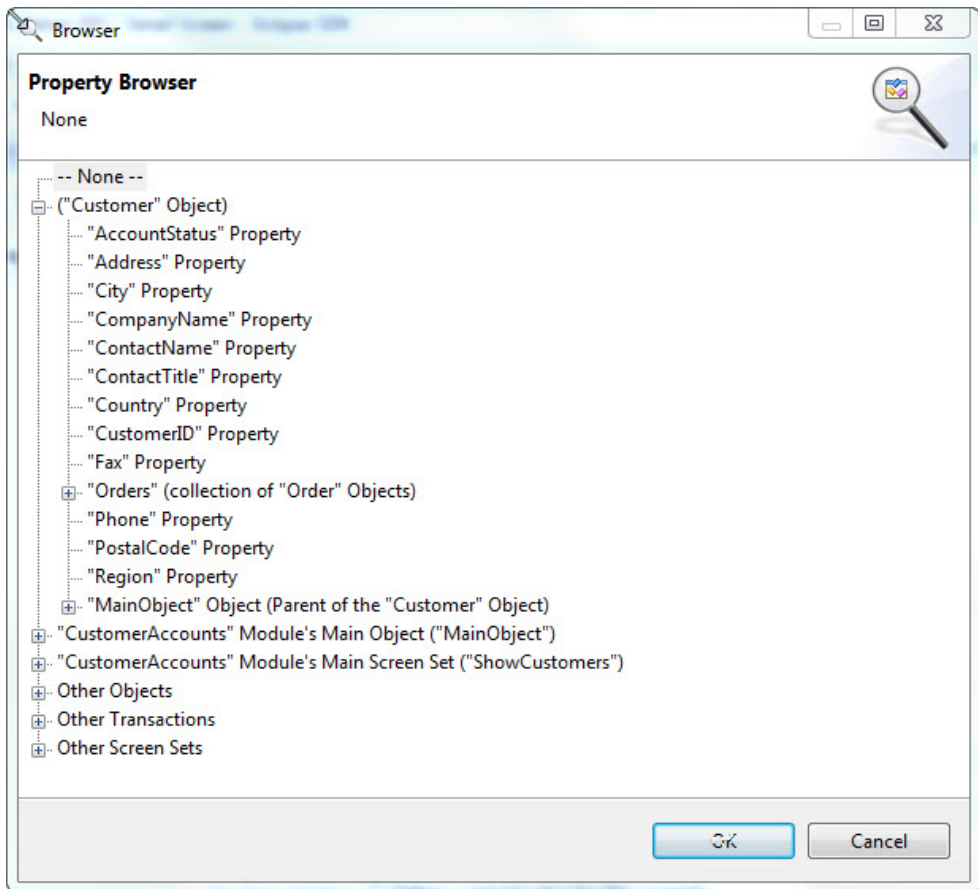
The available items in the property browser will vary from one definition to the next, depending on what the valid definition types are for the attribute and the definition containing that attribute. As with other aspects of the target paths behavior, this is driven by context.

An important concept to understand when using the Property Browser is that definitions are organized within it based on the data structure defined within the application project as it will exist on the Agentry Client at run time. This is a different structure than the one presented in the main Project Explorer View of the Agentry Editor.

Consider the example of two of the object definitions Customer and Order found in the Mobile Northwind application. When viewing this project in the main Project Explorer view, both of these definitions are listed under the module as its child definitions. Customer and Order are sibling definitions within the application project and are therefore presented at the same level in the hierarchy. By contrast, when viewing the Property Browser these definitions are presented differently. The Customer object may be presented as a root node in the tree (as in the previous example). Expanding it reveals all of that objects properties, including the Orders collection property which will contain the Order object instances for a given Customer at run

time. While the Project Explorer View also lists this collection property, the Property Browser goes further in that the Orders collection can be expanded to reveal several child nodes related to selecting a specific Order object instance based on some condition. Furthermore, under the selection criteria for an object instance in a collection, additional child nodes are displayed to allow for the selection of a specific property within that object.

The presentation of the definitions of the Property Browser is a reflection of the fact that the purpose of a target path is to select an instance of a definition at run time, based on some criteria. The criteria can be as simple as some property in the object instance targeted by a transaction; or it can be more sophisticated, such as a property in the parent object to the one targeted by the transaction, or even based on a rule evaluated at run time. Whereas the Project Explorer View deals with the definitions, the Property Browser details within specific instances of those definitions.

Beyond objects and their properties, the Property Browser can display several other definition types from which data can be retrieved on the Agentry Client at run time. These include items such as screen field values, the selected object in a list, a record in a data table or complex table, as well as specific fields within those records, and properties in transactions or fetches. The specific definition types available depends on the context in which the path will be evaluated at run time. For example, an object read step includes a **Read Into** attribute that specifies the target collection property into which data is read by that step during synchronization. When the Property Browser is displayed for this attribute there are far fewer options available than for some other attributes. in this situation, the options are limited to only object collection properties nested under the object for which the read step is being defined.

The Property Browser presents numerous options for selecting a specific definition instance, such as an object within a collection or a record within a complex table. Such options include the first or last item in a set, an object where the key property matches some specified value, or a record or object instance returned based on the evaluation of a rule. Understanding how to use the Property Browser to create such target paths is an important concept when developing many real-world applications or modifying existing applications to meet the specific needs of an implementation.

Basic Target Path Syntax
The syntax rules for a target path are important to understand so that when viewing a target path the developer can understand where the value being referenced is located within the application. There is no need, however, for the developer to understand the syntax at such a level as to be able to create such a path by hand. The Agentry Editor does not allow a target path to be entered manually for any attribute.

First, all target paths begin with the symbols `:>`. This simply denotes the beginning of the path for the target path parser built into the Agentry Client. This symbol is then followed by a definition name and type, or in some cases a generic definition type. A basic target path can end with just a single definition if that definition is found within the definition instance that is currently on context. Other paths contain multiple components, with each separated by the >

symbol. Each definition referenced in the path includes the name enclosed in quotes followed by the definition type.

Looking at the previous example, then:

```
:>Main Screen Set>"ShowCustomers_List_PPC" List Screen>Current
Object>"CustomerID" Property
```

The first component to this path is `Main Screen Set`. This is a generic component that refers to the main screen for the module. This value does not contain the name of this definition, which allows it to be used and evaluated properly even when the name of this screen set changes. The second component to the path is "`ShowCustomers_List_PPC`" `List Screen`. This component specifies the list screen of this name found within the main screen set of the module. `Current Object` is the next component and this refers to the currently selected object within the list screen. The final component is the "`CustomerID`" `Property`. This then specifies the CustomerID property within the currently selected object in the list. To create this path, the following was selected within the Property Browser of the Mobile Northwind application project in the Agentry Editor.
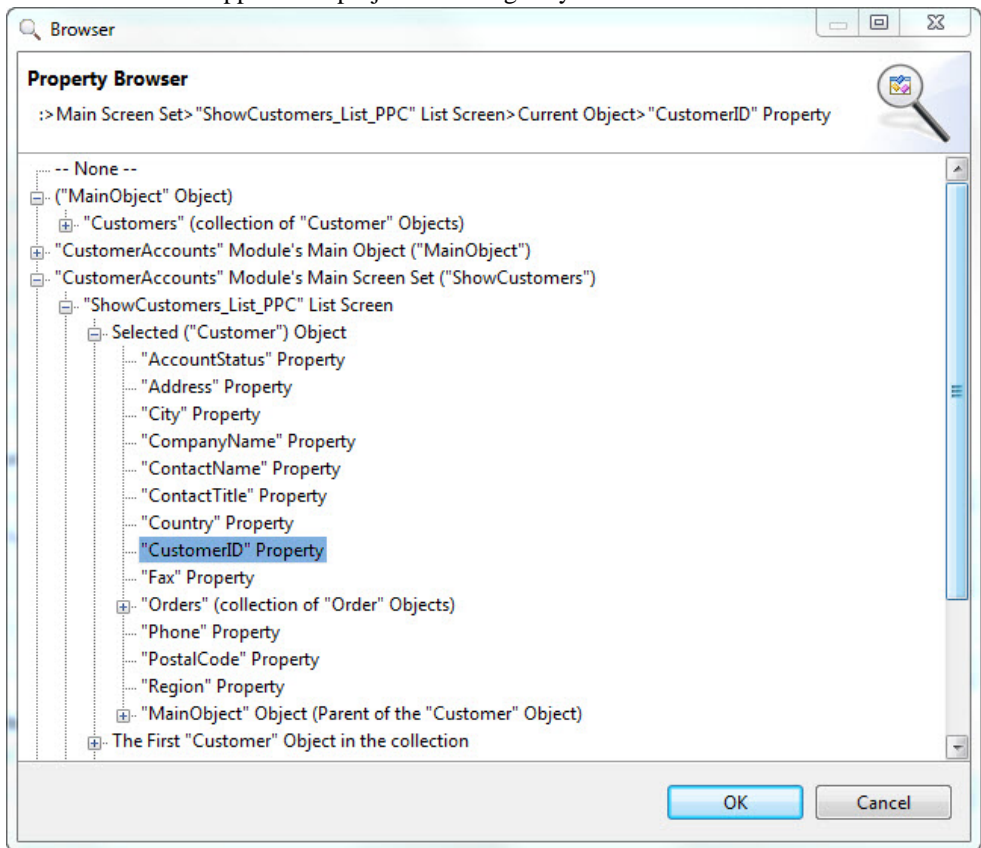
Note the selection, including the nodes above it in the tree control. The root node of the selected item displays "CustomerAccounts" Module's Main Screen Set ("ShowCustomers"). However, this full description is not the item returned for the target path. Note the line of text displayed above the tree control in the screen header of the Property Browser. This is the actual target path that will be returned to the attribute for which the selection is being made.

In many cases a definition will be replaced in the target path with a more generic value, such as "Current Screen Set", "Current Property", etc. At run time, when the target path is evaluated by the Agentry Client, these more generic values allow for the reuse of a target path in multiple contexts. In some cases such paths are required. As shown in an example to be provided shortly, the target paths evaluated in the context of a detail screen that is in turn displayed through one of the tile category of detail screen fields (List Tile View, Tile Edit, and Tile Display) require such generic values in the target path when referencing another value on the same detail screen.

In other situations the target path selected in, for example, a rule definition may be replaced with a more generic value after that rule is saved. A common example of this behavior is when specifying a collection property for the @COUNT function. In many cases when the collection is selected the target path displayed in the rule editor will be something similar to "Customers" Collection Property. However, when the rule is saved and subsequently viewed or edited that same path will be the value "Current Property". If the context of the rule evaluation includes the previously selected Customers collection, current property will resolve to that collection. Furthermore, the replacement of the name-specific value with the more generic one allows that rule term to be evaluated in the context of a different collection property, such as Orders.

Property Browser Details: Object-Related Options

The property Browser presents numerous options related to the selection of an object instance and a specific property value within that instance. This includes both the selection of a value from the object in the current context and also the selection of an object within a collection. Many of the more sophisticated options are related to the selection of an object instance within a collection. These options include:

- Selection of a property within the object instance in the current context
- Selection of an object instance within a collection based on it's position (first or last)
- Selection of an object instance within a collection based on a rule definition
- Selection of an object instance within a collection based on the key property value of that object

The specific layout of the module's data structure within the Property Browser depends on the context in which the target path being selected will be evaluated at run time. In some cases the object is displayed as the first root node. In other contexts a collection is presented. In addition to these options, others may be available for screen sets, complex tables, data tables, and other options. These are addressed elsewhere, with the focus here on the target paths selected solely based on the object instances.

Target Path for an Object Property - No Collection

When setting the target path of an attribute, and when the context of the path to be evaluated includes a single object instance, the Property Browser is presented similar to the following:
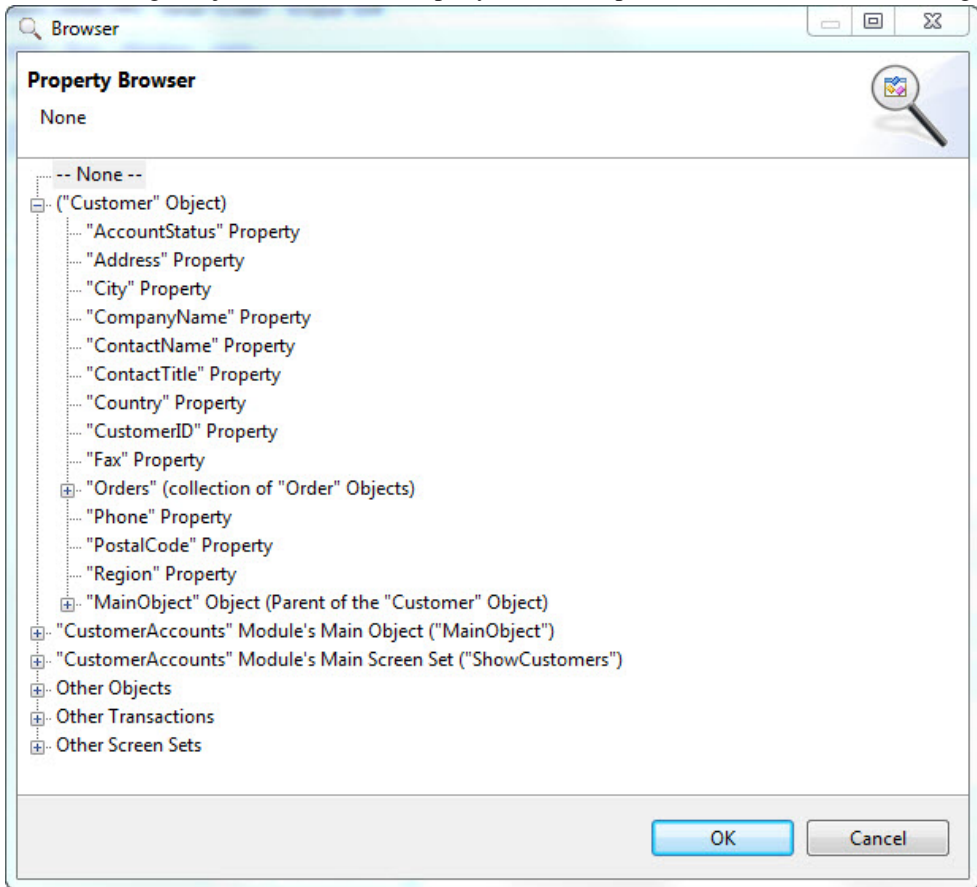
In this example the target object property of an edit transaction property is being set. The object in context is the object to be modified by the transaction. In the Property browser, the Customer object is displayed as the first root node and is expanded, as shown in the above example. If the proper option for selection is one of the properties under the Customer object, then it is likely that the Property Browser is not needed, as in almost all contexts the properties of the customer object would be displayed in the context menu of the attribute.

However, if the target paths should reference a collection property within the object, the Property Browser is needed. Collection properties are not displayed in the context menu of a transaction property's target object attribute. The collection is a valid selection if the edit transaction should replace the object collection in the object with values captured by the transaction.

It is also possible to reference the parent object of the one in the current context. Again in the previous example, the Customer object is expanded. Note the last child node under the Customer, which is MainObject. The description indicates this is the parent object of Customer, meaning the object instance is stored in a collection property of the MainObject. To select another property within the MainObject, that node can be expanded and the property selected from it. This same general procedure is available to all objects stored in collections to access the parent object containing that collection.

Target Path for an Object Instance - Selected From a Collection

In many contexts the object to be selected exists in a collection of objects of the same type. The specific object instance to be targeted must be specified. In almost all cases once the specific object instance is determined, a property of that instance is then the actual target being selected. When working with an object collection property, the Property Browser provides several options to select the specific object instance:
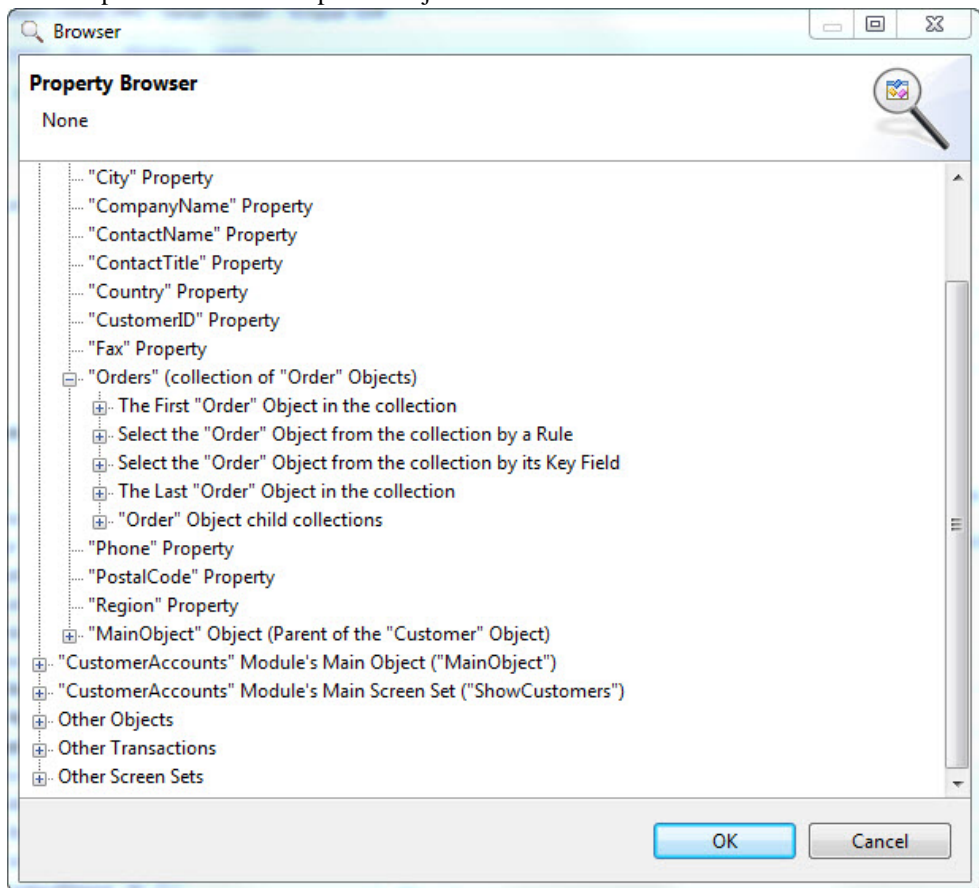
When the node for a collection property is expanded in the Property Browser, the following items are available:

- The first object in the collection
- Select the object from collection by rule
- Select the object from the collection by the key field
- The last object in the collection
- Object child collections

The First Object in the Collection: The first object in a collection is the first object instance added to that collection. This may be the first one retrieved from the back end system during synchronization; or if the collection is empty after synchronization, it is the first object instance added to the collection on the client.

In the case of data synchronization, the order in which the objects are stored in the collection is not guaranteed by the Agentry Client. The synchronization logic itself can include ordering of the data retrieved, which then provides an order to the objects stored in the collection. However, there are few use cases in real world applications where a target path is needed to select the first object instance in a collection.

The one real-world example of such a target path is when the collection is known to contain only a single object in all situations. One situation where this occurs is when an object exists in a collection of the MainObject to store user-related information. Typically the synchronization retrieves only a single object instance for this user information. When it is needed, a target path to that object must be created. Target paths for object collections always assume the possibility that multiple object instances can or will exist within a given collection. Therefore, selecting the first object in the collection in a situation where it is known that only one instance will ever exist is an acceptable manner in which to select this object.

Select the Object from the Collection By Rule: Selection this option requires the definition of a rule to evaluate the objects found in the collection. The default behavior at run time is for the rule to be evaluated once for each object instance in the collection until the rule returns true, at which point that object instance is returned, or until all objects have been evaluated. The rule is evaluated in the context of each object instance in turn, allowing access to the property values of each object.

As optional behaviors the object selected can be the first one for which the rule returns false; the last one for which the rule returns true; or the last one for which the rule returns false.

To select or define a new rule for this purpose, as well as to select options to change the default behavior, right click on the node in the Property Browser to display a context menu. Here a context menu is displayed with the following selections available:

- All rule definitions currently defined in the module.
- The option to define a new rule
- The option to specify if the object instance selected is the first or last to meet the rule's criteria (the first object found is the default)
- The option to specify whether to select the object instance based on a true or false return from the rule (true is the default)

If the option to select the last object meeting the criteria is defined, the rule will always be evaluated once for each object in the collection, as it must check all objects before determining

the last one to meet the criteria. This behavior should be considered if the selected collection has the potential to contain a large number of object instances, as it has the potential to increase processing time as the number of objects to evaluate increases.

Once the rule has been either selected or defined, and the optional other settings have been selected, the node can then be expanded to allow for the selection of the specific property within that object instance. The value of this property is then returned, or the property itself is targeted during run time.

Select the Object from the Collection by the Key Field: The object instance can be selected from a collection via the value of it's key property, or alternately by another property within that object. In most cases the key property should be used as it is the only value guaranteed to be unique for all object instances within the collection.

To set this option, right click on this node in the Property Browser to display a context menu containing the properties in the object definition, as well as the value to which it will be compared. The key property is selected by default, but can be changed to any other non-collection property within the object.

The last item in the context menu **is equal to (Browse for Property)...** is then selected to display a second Property Browser. In this screen the value to be compared to the key property of each object in the collection is selected. This select is an example of a target path within a target path. This "nested path" is evaluated once for each object in the collection being searched, with the value it returns being compared to the key property of each object instance in the collection, until a match is found or until all objects have been searched.

One alternate selection in this context menu is the **Browse**... menu item directly below the list of all properties within the object. This can be selected for more complex search criteria, and will display a second Property Browser. Specifically, selecting an object from a collection that contains a nested collection with an object instance that meets some criteria. For example, selecting a Order object from the Orders collection where that Order object contains an OrderItem object with a Product ID value of 70.

To make such a selection, the property to be compared would be the ProductID property of the OrderItem object within the collection property of the Order object. The **is equal to (Browse for Property)...** menu item is then displayed to select the value to compare against the ProductID property of the child OrderItem objects. Note that such a selection will iterate over each object in the nested collection property of each object instance in the parent collection. So in this example, each OrderItem object is checked in each Order object until a match is found. Therefore, the potential number of iterations performed by the Agentry Client for such a target path is a multiple of the number of objects in the parent collection and the number of objects in the nested collection. This type of searching is rare, but there are a handful of use cases for which it can be applicable.

The Last Object in the Collection: The last object in a collection is the last object instance added to that collection. This may be the last one retrieved from the back end system during synchronization; or it will be the last object instance added to the collection on the client.

In the case of data synchronization, the order in which the objects are stored in the collection is not guaranteed by the Agentry Client. The synchronization logic itself can include ordering of the data retrieved, which then provides an order to the objects stored in the collection. However, there are few use cases in real world applications where a target path is needed to select the last object instance in a collection after data synchronization, and issues with such logic arise in that this selection becomes invalid if users add another object to that collection on the Client via a transaction.

A typical use case for building a target path that selects the last object in a collection is in the area of multiple transactions being instantiated and applied on the Agentry Client, where a transaction creates a new object, and a subsequent transaction must modify that new object in some manner. An example from the Mobile Northwind application is the order entry functionality. The Order object represents an order placed by a customer and includes header information such as the order date and the shipping address. It contains a collection of OrderItem objects that represent the products for that order.

From a usability standpoint, it makes sense for the creation of an Order object and the addition of one or more OrderItems to that object to appear as a single operation to the user. To accomplish this an action can be defined that executes two sub-actions. The first instantiates and applies the AddOrder transaction. The second instantiates and applies the AddOrderItem transaction in a loop. Any add transaction must know to which collection property the object it creates will be added. In the order entry scenario, the AddOrder transaction creates an object and adds it to the Orders collection property of the current Customer object. The subsequent AddOrderItem transaction must then target the collection of OrderItems contained in the newly created Order object. To specify this Order object it is safe to select the last Order object in the collection as the parent to the new OrderItem object created by the AddOrderItem transaction.

Object Child Collections: The selection of the Object Child Collections node under a given collection property displays a list of all collection properties nested under the current collection. Selecting one of these nested collections results in the return of all object instances in that collection in all instances of the parent collection.

An example to explain this is the Orders collection property contained in the Customer object of the Mobile Northwind application. If it is desired to define a list (list screen, list tile view field, list selection field, etc.) that displays all of the orders for all customers downloaded to the Agentry Client, the object child collections type of target path can be used. At run time this target path is evaluated by the Client and retrieves all Order objects from every Customer object. It can then display all of these Order objects in a single list regardless of the parent Customer object in which any of them are contained.

Property Browser Details: Screen-Related Options

The Property Browser provides several options for creating a target path based on the selections made and the values entered or displayed on a screen. In general there are two categories into which these options can be organized: List-related options and Detail Screen

Field options. The options available for different list controls are typically the same and, in most cases, the item currently selected in that list is the target path created. For detail screen field options, this is typically related to the value displayed in the field, or in some cases the item represented by the selection made in a field.

Target Path for List Controls

The target path based off of a selection in a list control, which can include the list displayed on a list screen, or the list displayed by one of the detail screen field types that present a list of objects from a collection, will return a property from an object within that list. The collection being displayed can also be accessed directly, though typically this target path is selected via one of the object-related options also presented by the Property Browser.

The following is a typical list related set of options as presented by the Property Browser:
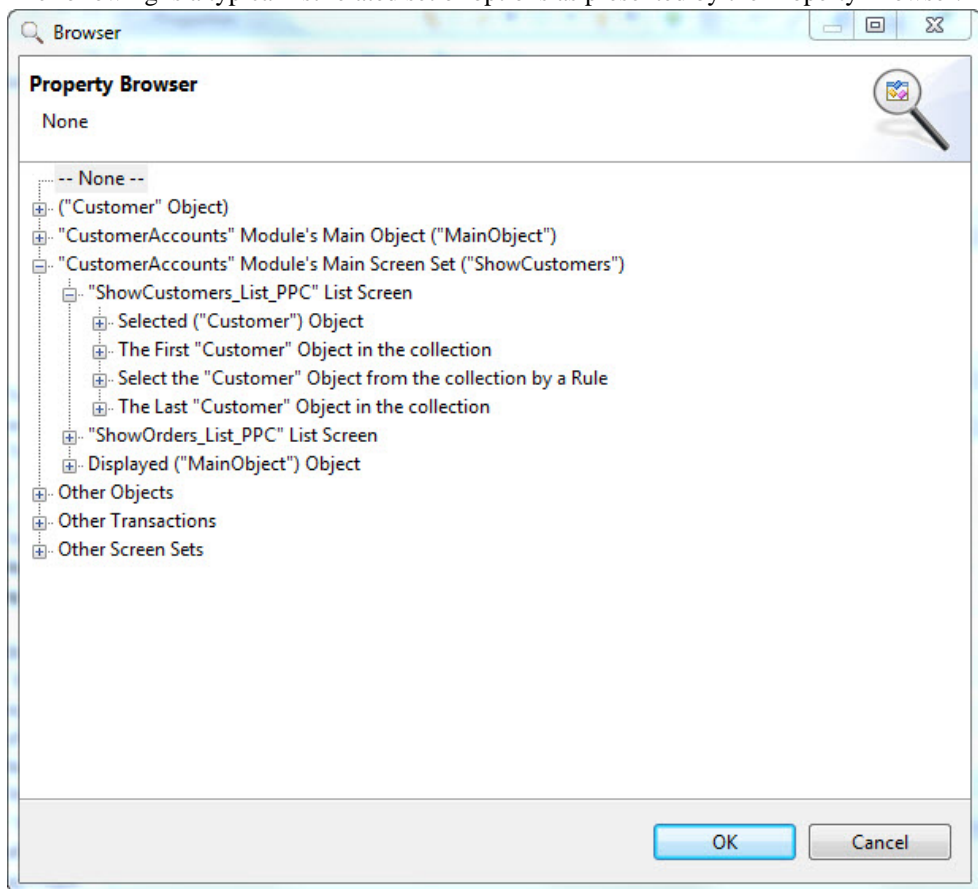
In this example the options for a list screen are shown. However, these same options are available for detail screen fields with an edit type of List Tile View and List Selection. All three

---

of these list types display a defined object collection property at run time on the Client, and all three allow the user to make a selection from that list.

The options displayed here include:

- The selected object
- The first object in the collection
- Select the object from collection by rule
- The last object in the collection

Of these options, only the selected object relates specifically to the different list controls. The other three provide the same target paths and behaviors as the options described in the Object-Related Options for target paths and Property Browser.

The Selected Object: Choosing the option of the selected object returns the object currently selected in the list. If the list is defined to allow for multiple selections, then this option provides the same options as when working with a collection property directly. The list of objects selected by the user are treated as a collection, with the instances in that collection being only those selected by the user. Like the options presented for a collection, there are numerous options that can be used to determine which specific object from those selected should be returned, including the first, last, by key field value, and by rule.

Property Browser Details: Complex Table-Related Options

The Property Browser provides several options for selecting a record from a complex table. Ultimately a specific field from the selected record is returned in almost all contexts. The specific field is a part of the target path created using the Property Browser. The selection criteria for the record can be based on one of the indexes defined within the complex table. When using an index, the first record, last record, or one selected by performing a search based on that index are all options. Searching on indexes in a complex table using a target path also includes support for parent-child indexes, if they are defined.

Target Paths for Complex Table Records

When a complex table record is targeted, the target path will ultimately include the specific field within that record whose value is to be returned. To select a record from a complex table, the selection criteria options are either the current record in the complex table, which is determined by context, or by using one of the indexes defined within the complex table. The following is an example of options provided in the Property Browser for a complex table definition:
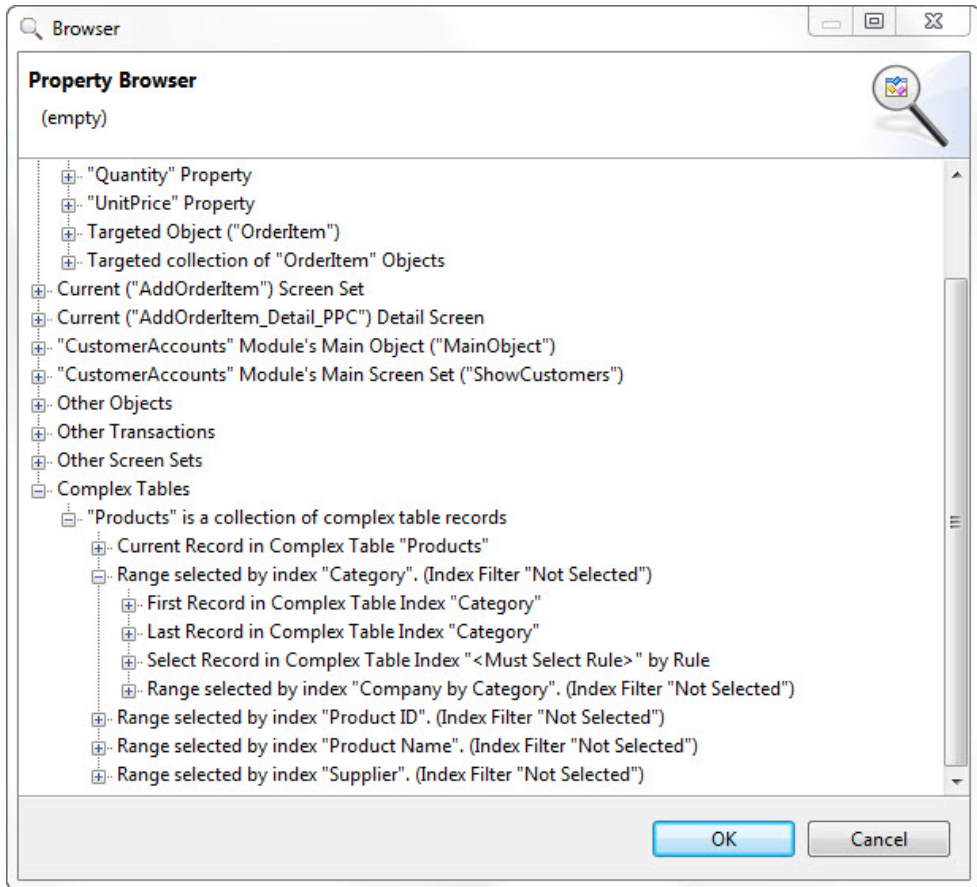
The options displayed here include the following:

- The current complex table record
- Range of records by index
- Select the first record in an index
- Select the last record in an index
- Select a record based on a rule
- Select the record using a child index within a parent index

The Current Complex Table Record: This option creates a target path that returns the current record in the complex table. The concept of the current record can be somewhat flexible and is based on the context of the target path evaluation. In general, the current record in a table is one that is currently selected in a list that displays complex table records, or some similar behavior.

Range of Records by Index: Within a complex table there will always be one or more index definitions. The index provides order to the records of the complex table based on the values of

a field within the records. indexes are defined to support both sorting and searching behaviors. When selecting records by using an index, right clicking on the node **Range selected by index "*IndexName*"** displays a context menu where the search value is specified. The source for this value can be either the value returned by a rule definition, or by creating a nested target path.

In the case of a rule, the rule itself is evaluated in the context of the parent definition for which the target path is being generated. The data type of the rule context matches the data type of the field for which the index in the complex table is defined. So if the index is defined for a field with one of the four string data types, the rule is evaluated in a string context and this is the data type of its return value.

In the case of a nested target path, a second Property Browser is opened and the source for the value to search on can be selected. Note that in this case, the source selected must be of the same general data type as the field to which it is compared. If the complex table field is a string, the search value must also be a string. The Agentry Client does not perform any data conversion for this comparison as it does in other situations.

Whether using a rule or some other data source for the search value, at run time the Agentry Client searches the complex table using the search value and the selected index. The resulting record may be either a single record or a range of records with the matching value. When selecting a range, the return should be for some list that supports displaying complex table records. This behavior allows for creating a temporary list of records that is a subset of all records in the complex table. This subset is dynamic in the sense that the source for the search value may return a different value under different conditions, especially when a rule definition is used. When a range of records is to be returned, the specific field from those records is not selected.

Select the First Record in an Index: When selecting the node **First Record in Complex Table Index "*IndexName*"**, the Agentry Client uses the order provided by the selected index to select the record, with the first record in the index being selected. A single field is selected for this option as the value returned for that record. As an optional behavior, right clicking on this node presents a context menu that allows for the specification of a record position within the index, counting up from the first record, which is at position 1.

Select the Last Record in an Index: When selecting the node **Last Record in Complex Table Index "*IndexName*"**, the Agentry Client uses the order provided by the selected index to select the record, with the last record in the index being selected. A single field is selected for this option as the value returned for that record. As an optional behavior, right clicking on this node presents a context menu that allows for the specification of a record position within the index, counting back from the last record, which is at position 1.

Select a Record Based on a Rule: When selecting the node Select Record in Complex Table Index by Rule, a rule definition is selected or defined by right clicking this node and selecting the appropriate option from the context menu. The rule used here is evaluated iteratively, once for each record in the complex table, until a match is found. The data type of the rule's context is Boolean.

The record selected is the first for which the rule returns true, by default. The order in which the records are processed is dictated by the order established by the index definition. As optional behaviors, the target path can be defined to select the first record for which the rule returns false. Also, the last record within the index for which the rule returns true (or false if selected) can be returned instead. Both of the optional behaviors are set by selecting the corresponding options in the context menu displayed for this node.

Select the Record Using a Child Index Within a Parent Index: When a child index is defined within the complex table, expanding the parent index reveals options for that child. These options are the same as the others for an index. However, the overall behavior includes child index searching. When one of these options is selected, selection criteria must also be specified for the parent index. Specifically, the selection of a range for the parent index must be specified, along with one of the selection options for the child index. This is required due to the nature of parent-child indexes in complex tables.

Property Browser Details: Data Table-Related Options

The Property Browser presents several options to select a record from a data table. The resulting target path created typically returns one of the two field values from the record, either the code or value. In some cases it may return the record, depending on the context in which the path is evaluated.

Target Paths for Data Table Records

When using the Property Browser for a target path to select a data table record, the options presented include selection via rule or position, as well as the current record. Typically the selection also includes specifying the field from the record to return. By default the value field is returned. Right clicking on any of the selection nodes provides a context menu where this default can be changed to the code field.

The following is an example of the options presented for a data table's target path in the Property Browser:
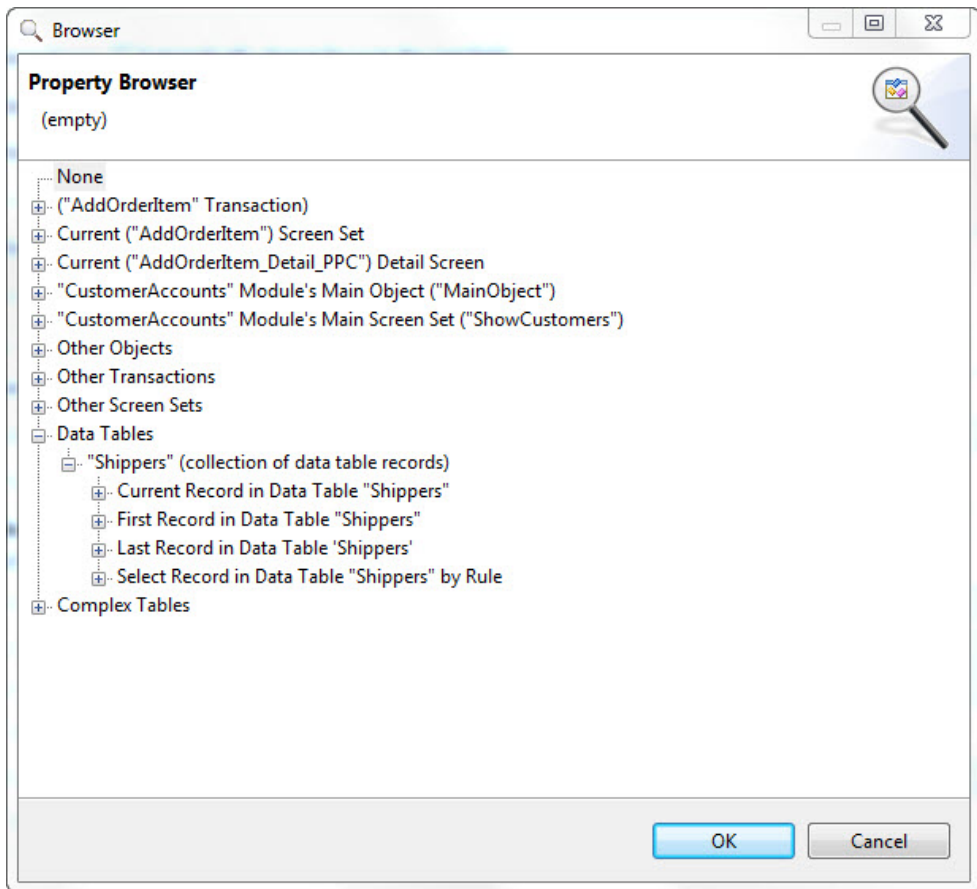
As shown in this example, the following options are available:

*   The current data table record
*   The first data table record
*   The last data table record
*   Select a record based on a rule

The Current Data Table Record: When the node **Current Record in Data Table** is selected, the record returned by the target path is the "current" record. The concept of a current record typically relates to the record in the data table currently selected in a list, or possibly in some other context where a current record exists, for example in a nested target path where the parent path includes a record's selection based on some other criteria.

The First Data Table Record: When the node **First Record in Data Table** is selected, the record returned is the first one stored in the table on the Agentry Client. It is important to note that the order of the records in a data table is not guaranteed, nor are the records sorted in any manner by the Agentry Client. However, the synchronization logic of the data table can order the records during retrieval to provide an order for their storage on the Client.

As an option to this selection, right clicking on the node presents a context menu with the option to specify a record at a position relative to the first. This position is set numerically, with the first record in the table at position 1.

The Last Data Table Record: When the node **Last Record in Data Table** is selected, the record returned is the last one stored in the table on the Agentry Client. It is important to note that the order of the records in a data table is not guaranteed, nor are the records sorted in any manner by the Agentry Client. However, the synchronization logic of the data table can order the records during retrieval to provide an order for their storage on the Client.

As an option to this selection, right clicking on the node presents a context menu with the option to specify a record at a position relative to the last. This position is set numerically, with the last record in the table at position 1. The second to last record is then at position 2, and so forth.

Select a Record Based on a Rule: When the node Select Record in Data Table by Rule is selected, a rule definition must be selected or defined. This rule is evaluated once for each record in the table, with the record in context. The rule's return data type is Boolean. The default behavior is to select the first record in the table for which the rule returns true, at which point the rule is no longer evaluated.

As optional behaviors, the return value of false can be used to specify the record to select. Also, the last record for which the rule returns true (or false, depending on the option selected) can be the one returned. If the last record is selected, the rule will always be evaluated for each record in the data table.

Target Path: Selecting an Object By Property Value

Prerequisites

Prior to performing this procedure the following items should be addressed:

- The object collection to be searched must be defined.
- The definition containing the value to be compared to the property value of the object must exist. This could be a property, a screen field, a global, or any other definition from which a value can be retrieved.
- It is strongly recommended that the value to be searched on is the key property of the object type contained in the collection.

The following items and definitions are a part of the example application project used in this procedure:

- The module's object data structure is Customer -> Orders Collection -> OrderItems Collection.
- OrderItems is a collection of OrderItem objects that represent items within a given order. OrderItems can be added to the Order object using the AddOrderItem transaction and related wizard screen set.

- When adding an OrderItem, the complex table Products is used to select the item to be added. The key property of the OrderItem object and the key field of the Products complex table are both the ProductID value.
- The AddOrderItem screen set displays a complex table search field for the Products complex table. When a record is selected in this field it displays the ProductID value of the selected record.
- A field of edit type Label is present on the detail screen in the AddOrderItem wizard. The field is defined to display the label text DUPLICATE. The RedText style has been applied to this field to display the label text in red and in a larger font size than other field on the screen.
- Other fields not directly related to this procedure displayed on this screen are the unit price, quantity being ordered, and discount percentage.

Task

This procedure provides instructions and an example on how to create a target path that returns an object instance from a collection property where a property of that object instance matches some value. The value to be compared against the collection must be accessible in the current context and therefore must be defined prior to performing this procedure. For this example, a detail screen field displaying a string will be used.

At run time the behavior of the Agentry Client will be to evaluate the target path and to compare the selected property in each instance the object in the collection property to the selected search value until a match is found. The first matching object instance is then returned. Note that this iterative processing can include up to as many iterations as their are object instances stored in the collection property. Typically this is not an issue for performance unless the collection contains an exceptionally large number of object instances (hundreds or thousands). However, if this target path is itself evaluated as a part of some outer loop of iterative processing, a performance issue could be encountered. For example, if the collection contains 100 object instances, and the outer loop iterates as little as 10 times, the total number of evaluations could be as many as 1,000.

Such a situation could also be encountered if the target path is evaluated in an update rule for a detail screen field. Update rules are evaluated numerous times during the initial presentation of the parent screen, and additionally whenever the user interacts with the detail screen. On a wizard screen, this would result in the target path being evaluated each time the user enters a character in a string field, or makes a selection in some list or drop down field on the same screen.

In this example the Property Browser is used to create a path that searches the OrderItems collection of an Order object for an OrderItem with a ProductID (key property) that matches the one currently selected in the AddOrderItem wizard. The goal is to display a clear indicator on the AddOrderItem wizard screen when a product is being selected that has already been added to the OrderItems collection. While the Agentry Client would prevent a second object with the same key property from being added to the collection, and an error message is displayed, this does not occur until the Agentry Client attempts to apply the transaction. A

cleaner user interface is possible that displays a message as soon as the duplicate product is selected.

The label field contain the text DUPLICATE is to be modified with it's Hidden Rule attribute set to a rule that checks the OrderItems collection for an object with a ProductID value equal to the one currently selected in the Product ID field of the AddOrderItem wizard. The rule will contain a target path that makes this check and returns the property ProductID from the object found. If no object is found, then a null value is returned by the target path. The return from this path is to be passed as a parameter to the rule function NOT, which treats any parameters as Boolean values and inverts them. The Hidden Rule evaluates the rule it contains in a Boolean context. When such a rule returns true, the field containing the attribute is hidden from the user. When it is false, the field is displayed.

The overall logic, then, will be that the user selects a product from the complex table. The Hidden Rule for the label field DuplicateFlag will be evaluated. This rule contains the target path that compares the selected ProductID value to the ProductID property of each object in the OrderItems collection. If a match is found, the ProductID value of that object is returned to the NOT function within the rule. This is treated by the NOT function as true, since the path is evaluated in a Boolean context and any non-null value is true. NOT inverts this value, thus returning false to the Hidden Rule attribute. This results in the DuplicateFlag field being displayed. On the wizard screen the text DUPLCIATE is displayed in large red text. If the user selects a product that is not currently found in the OrderItems collection, the target path in the Hidden Rule will not find a matching object. The value returned by the path will be NULL. The NOT function treats NULL as false. It will then invert this value, returning true to the Hidden Rule attribute. This will hide the DuplicateFlag field on the screen.

1. In our Mobile Northwind application we select the DuplicateFlag field definition in the AddOrderItem_Detail_PPC detail screen. In the Properties View we view the Rules / Hyperlink / Special Value tab:
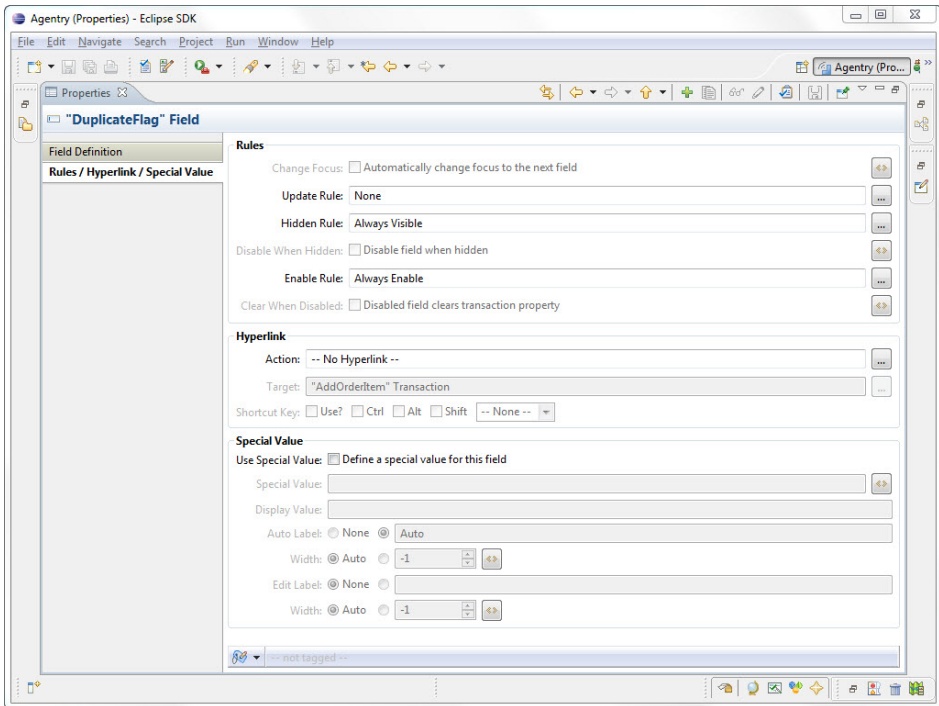
2. Clicking the ellipses field to the right of the Hidden Rule attribute displays a context menu. Selecting the menu item **Add Rule** displays the Rule Editor. Here the name can be left set to the default of DuplicateFlag_HiddenRule. Advancing the wizard displays the main Rule Editor screen. The first term to be added to this term is the NOT function:
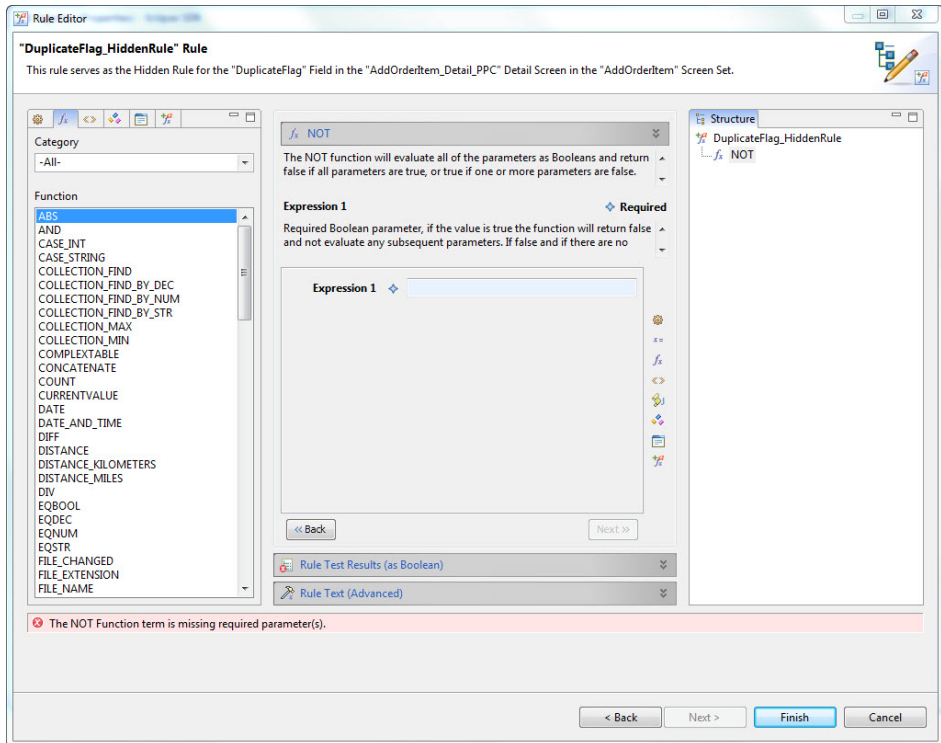
3. The first and only expression parameter for this function is to be the target path to search the collection. With the Expression 1 field selected in the Rule Editor, we click the Properties list and select **Browse Properties...** This displays the Property Browser. In this browser we will build the target path to search the OrderItems collection targeted by the transaction. We select the path **AddOrderItem Transaction | Targeted collection of OrderItems Objects | Select the OrderItem Object from the collection by its Key Field**. Right clicking the last item in this path displays a context menu:
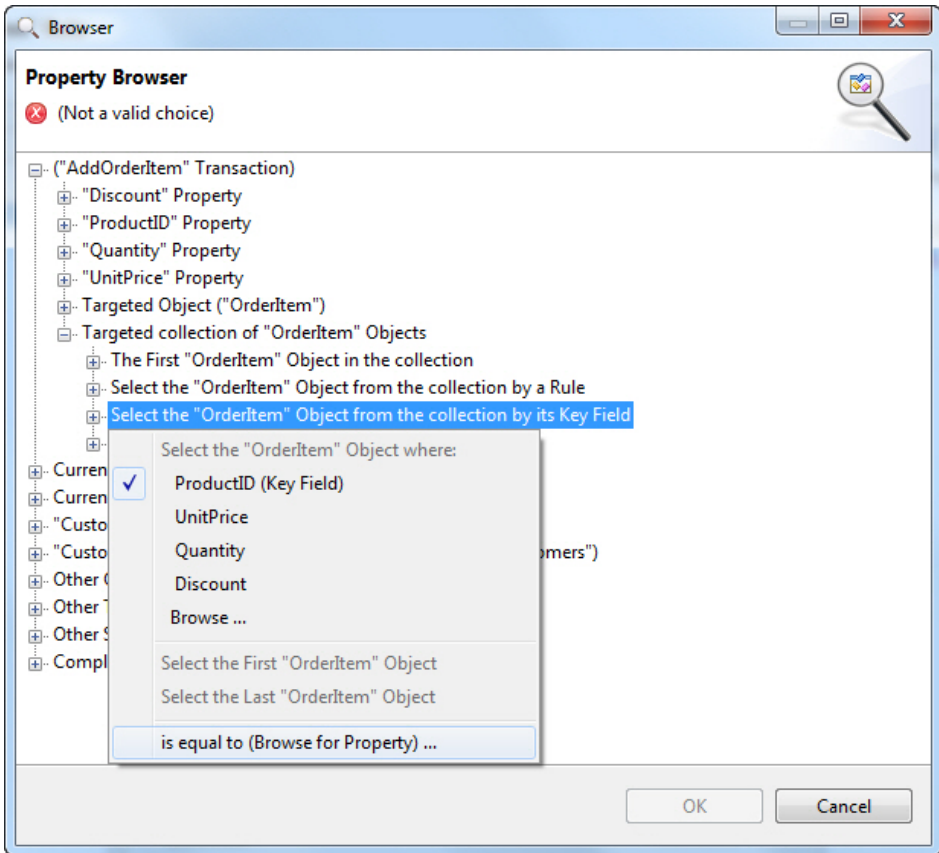
This selection creates the first part of the target path, which indicates the OrderItems collection targeted by the transaction. We are then specifying that we want a single object instance from this collection. There are multiple options for choosing the object instance. For this use case the proper selection is to look for the object by its key property.

4. In the context menu, first note that the key property ProductID is selected by default. The other properties are also listed, but typically the key property is used as it is the only value guaranteed to be unique within the collection. We now select the last item in the menu **is equal to (Browse for Property)...** This selection is where the value to be compared to the key properties of the object instances is selected. This selection displays a second Property Browser. The value we wish to search on is the current value in the Product ID field of the detail screen. So, the path we select is **Current ("AddOrderItem_Detail_PPC") Detail Screen | ProductID Field**:
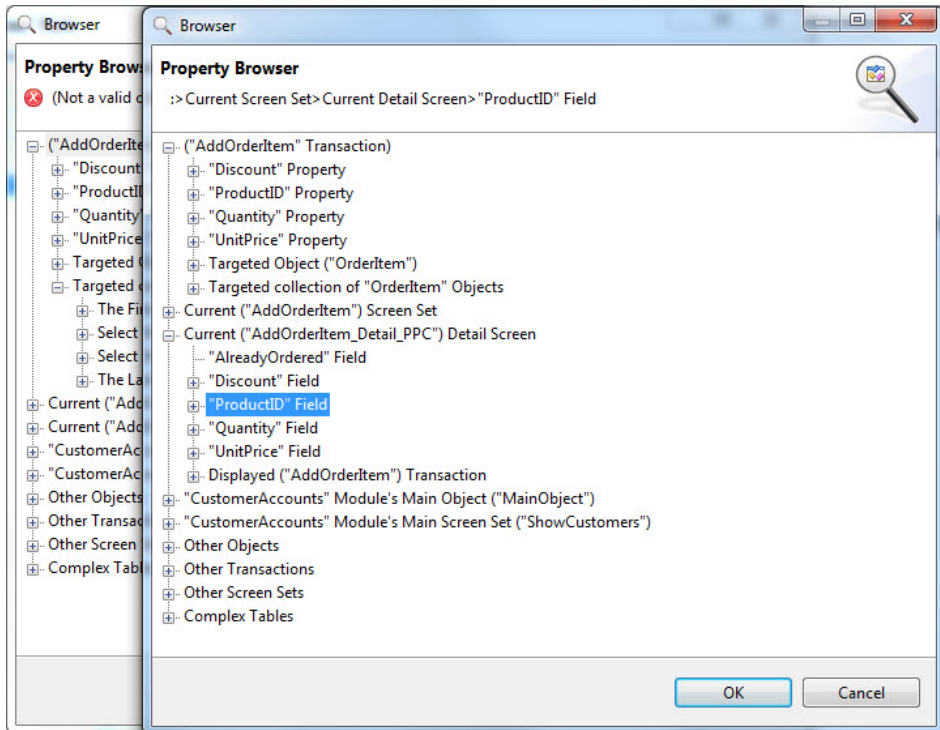
This selection creates a target path that returns the value currently displayed in the Product ID field of the detail screen. This path can be thought of as one that is contained within the path selected in the first Property Browser. This "inner path" is evaluated once for each object in the OrderItems collection until the value it returns matches the ProductID property of an object in that collection, or until all objects have been searched.

5. Click the OK button in this second Property Browser screen. We must now make one final selection in the first Property Browser, which is the property value to be returned from the OrderItem object found in the collection. This could not be selected previously as this selection cannot be made until the search criteria is specified. Now that it is selected, the node indicating search by key property can be expanded, revealing the properties within the OrderItem object. Here, the property to be returned is selected. The safe selection for our use case is the ProductID, as it will always contain a value. Other properties can be selected in other use cases, depending on what data is needed from the object found:
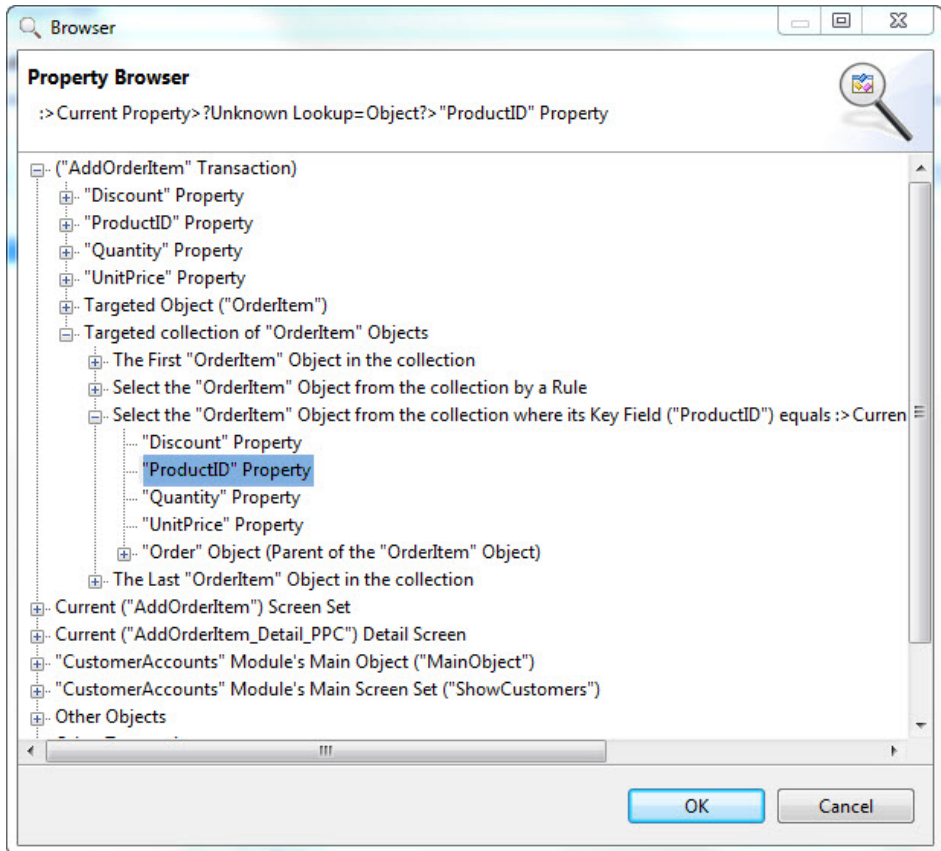
6. Close this Property Browser by clicking OK. We are returned to the Rule Editor, where the rule now appears as follows:
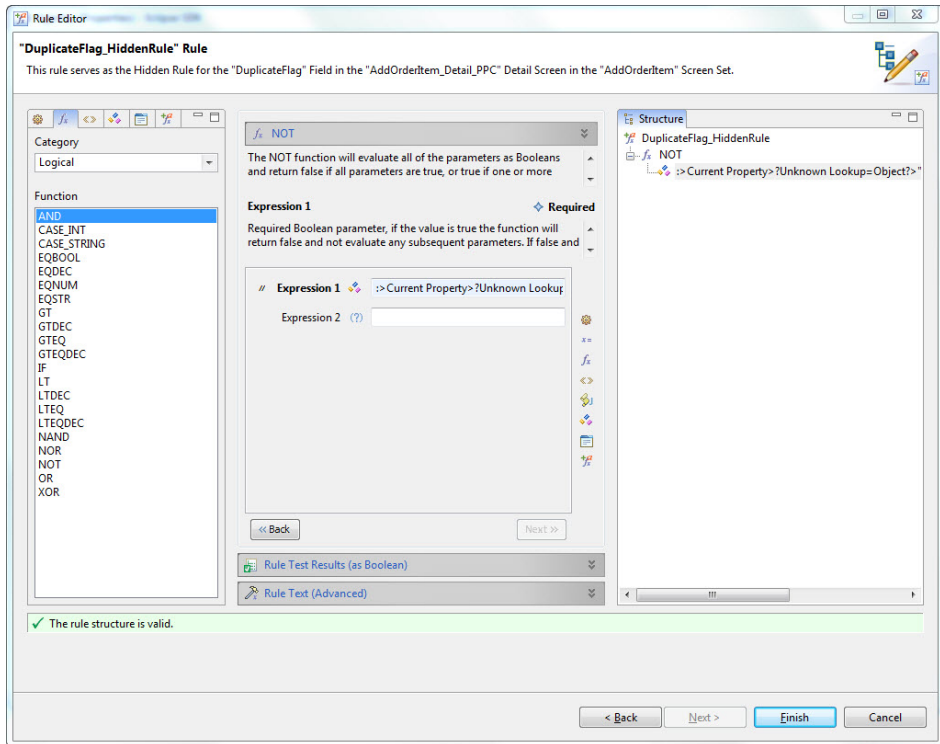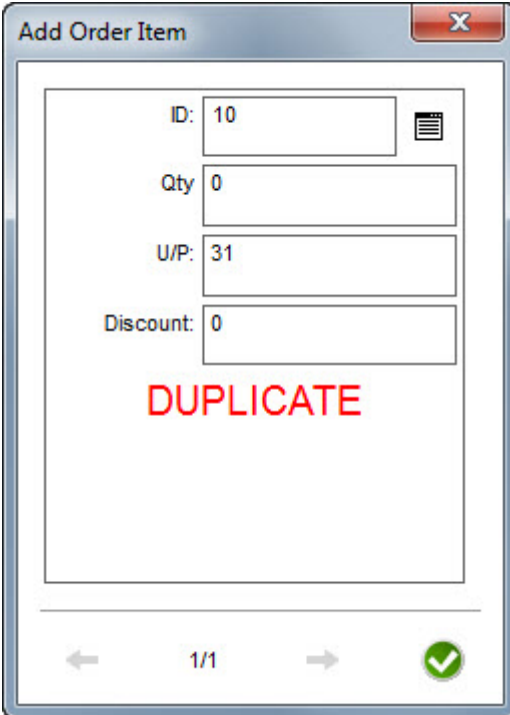
Clicking **[Finish]** returns us to the Properties View for the DuplicateFlag label field. The changes are saved in this view and the modification is complete in the Application Project.

At this point, the changes made are complete. We will now publish and test this modification. the following examples are from the Agentry Test Environment.

When the user adds an OrderItem for an Order, the AddOrderItem wizard is initially displayed:
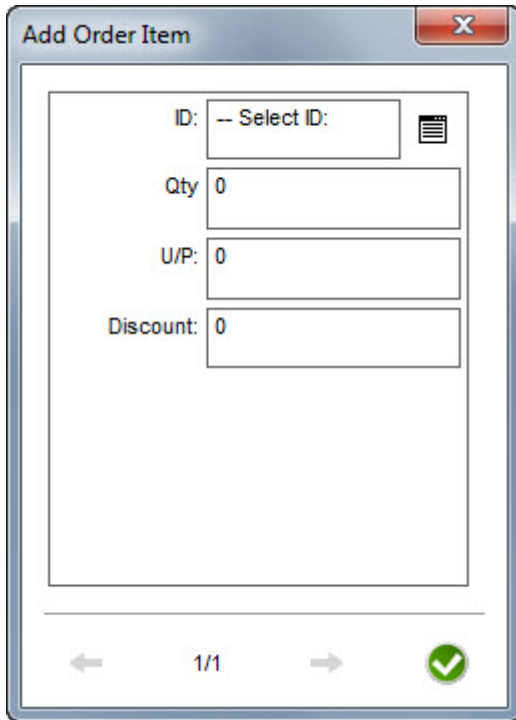
When the user makes a selection in the ID field, the Hidden Rule for the currently hidden DuplicateFlag field, the target path built in the previous procedure is evaluated within the rule definition. The value displayed in the ID field is compared to the ProductID property in each OrderItem object in the collection targeted by the AddOrderItem transaction. If a match is found, the path returns the ProductID of that object instance, which is treated as true. The NOT function inverts the value within the rule, returning false. A false Hidden Rule value indicates the field should not be hidden, and the field's label text is displayed:

The user has now been informed that the product currently selected is already a part of the Order. If the user then makes another selection, the Hidden Rule, including the target path it contains, is evaluated again. If no matching OrderItem is found, the DuplicateFlag field is hidden and the user knows the product can be ordered:
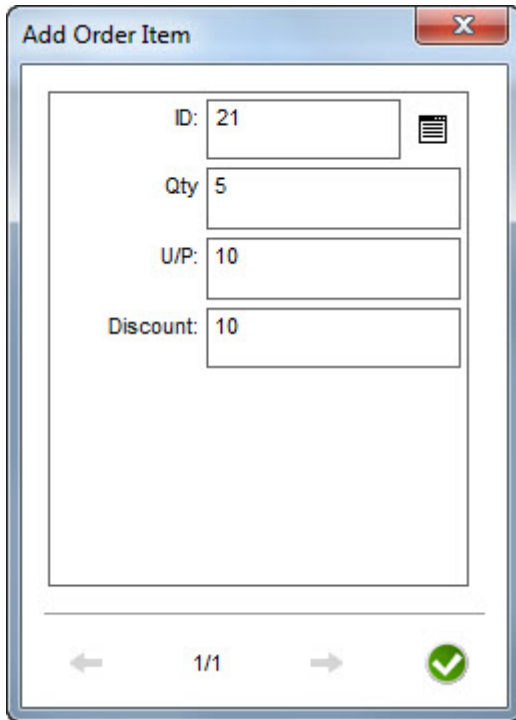
Target Path: Selecting All Nested Collections

Prerequisites

Prior to performing this procedure the following general items must be addressed:

- The object data structure must be defined, including the parent object and the collection property within that object.
- The functionality described here is only available on Agentry v. 6.0 and later.

The following items and definitions are a part of the example application project used in this procedure:

- The module object structure is defined as Customers -> Orders -> OrderItems. Each of these is a collection property of objects and are nested as listed. Customers is a top-level collection and thus is stored in the module main object.
- The main screen set named ShowCustomers is already defined. It includes a list screen displaying the top level collection Customers.

Task

In many applications the desired user interface layout includes displaying object top-level collections and also all instances of objects stored in nested collections regardless of the parent

object instance. As an example, it is desirable to display a list of all work orders in a work management application on the main screen set of the module. Additionally, a common change is to also display all of the equipment objects, which are stored in a nested collection of the work order object, in a single list regardless of the parent work order object.

Prior to version 6.0 of the Agentry Mobile Platform, in order to support this user interface layout, the equipment objects would need to be retrieved separately from the work orders and stored in a top-level collection of equipment objects. A list screen or a detail screen containing one of the list type fields was then defined to display this collection. The challenge with this approach was that in order to also display a list of equipment objects to the user for a single work order, one of two approaches was needed.

One option was to have a top-level collection of equipment objects, and a second, nested collection within the work order object. The issue with this approach was, of course, that the same data was retrieved twice, that being the objects for both collections. Both would then be stored on the Agentry Client. This was a waste of resources on the device, in essence doubling the amount of storage required to store the equipment information. The logic needed to retrieve all equipment for the work orders assigned to a given user was also sometimes challenging to write in an efficient manner.

The other option was to store all equipment objects in only the top level collection. To then display the equipment objects for a given work order, it was necessary to define an include rule for the list in which these equipment records were displayed that would only return the equipment objects from the collection for the currently selected work order. This could cause performance issues if there was a large number of objects stored in the top-level collection of equipment. Also, if this behavior was desired for what would otherwise be nested collections, numerous rule definitions were needed, making the maintenance of the application project more cumbersome.

Fortunately this issue has been addressed in the 6.0 release of Agentry. A new target path option is now available in the Property Browser that allows for the selection of all objects in all instances of a given collection property definition. So, building on the previous example of work orders and equipment, the work order object is defined to contain a collection property for equipment. There is no top-level equipment collection, only the nested collection in the work order object. A list can now be defined to display all of the equipment objects found in all of the collection properties of all work order objects by selecting the proper target path to return this data.

In the following procedure this behavior is defined for the Mobile Northwind application. A list screen is added to the module main screen set ShowCustomers, which initially contains a list screen for the Customers collection. The second list screen to be added will display all order objects for all customers. The data, however, will only be stored in the nested collections, meaning a given instance of the Customer object will contain an Orders collection property of just that customer's previous orders. The Agentry Client will merge all instances of the Orders collection from all Customer object instances and display them as a single list in the list screen.

Note that while a list screen is used in this example, the option to display all nested collection instances for a given parent object type is available in numerous cases and contexts. This includes other list controls, such as a list view or list tile view detail screen field, and also in non-user interface contexts for operations that affect a collection.

1. We begin by creating the list screen (or one of the list type fields for detail screens, where applicable) and setting the attributes as normal.

2. When setting the Collection attribute, click the ellipses button to the right and select **Browse Properties...** from the context menu. This displays the Property Browser. Here, select the path **MainObject Object | Customers (collection of "Customer" Objects) | "Customer" Object child collections | All "Orders" (combined collection of all "Order" Objects in all "Customer" Objects)**:
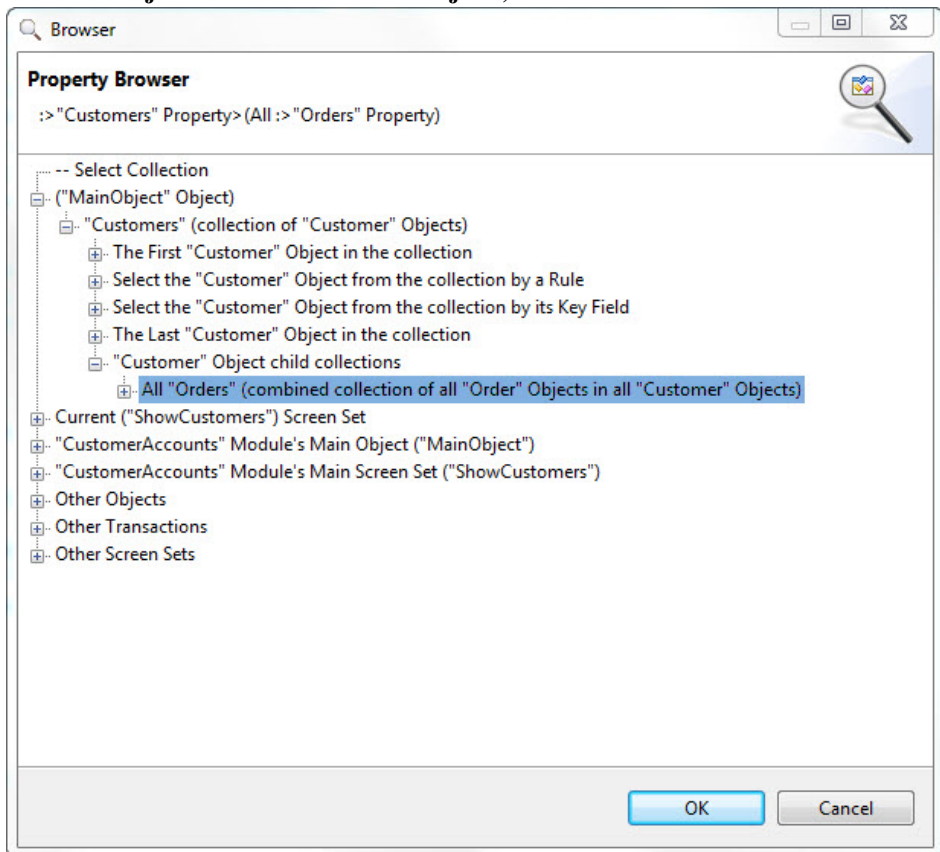
3. Close the Property Browser and return to the wizard (or Properties View if editing an existing definition). Complete the definition as normal.

When this procedure is complete a list screen (or other list control) is created that will display all object instances from all of the selection collection property instances in the parent object type. In the above example this results in a list screen listing all orders placed by all customers:

Rules: An Introduction

The Rule definition type is a module-level definition within the application project. A rule defines evaluation logic processed on the Agentry Client. A rule is evaluated by some other definition that calls or references it. The rule will return a single value to the caller. This value is then used by the referencing definition for its own purposes. The caller of the rule sets the rule's context. Rules are made up of data terms and function terms.

The rule definition is the most complex of the definitions within the application project. Its overall purpose is to perform involved logical evaluation and to then return a single value based on or resulting from that evaluation. Within the application project there are dozens of attributes that can reference a rule definition. The uses for rules are varied and range from dynamically setting display values, to enforcing business logic, to enabling or disabling functionality based on some condition.

There are several concepts related to rule definitions that are necessary to understand before defining rules. These include:

- Components of a Rule's Structure
- Context of Rule Evaluation
- Rule Function Terms
- Rule Evaluation at Run-Time

Components of a Rule's Structure

A rule is a definition within the application project and, as such, does contain a handful of attributes. Specifically, each rule has a name and a group. The name uniquely identifies the rule within the module. The group is an application project-specific value used to organize rules.

The real definition of a rule, however, is contained in its Structure. The structure of a rule is the encapsulation of the logic to be evaluated at run-time. This logic then determines what the rule ultimately returns to the definition that called the rule. The components of a rule's structure are referred to as Rule Terms. There are in general two types of terms that make up a rule. These are Function Terms and Data Terms. How these terms are then organized within the rule definition provides the overall structure of the rule; i.e., its evaluation logic.

Function terms provide specific processing or logic performed during the rule evaluation. Most functions take one or more arguments, or "parameters" that provide values to be processed by the function. The function itself will then return a value to its caller based on the value of its parameters. A function's parameters can include both the return value from other functions and rule data terms.

Data terms are any term that is not a function and that provide data values processed by functions within the rule. There are several different sources for a data term, including properties, globals, actions, screen sets, and constant values set within the rule structure.

Context of Rule Evaluation

The processing of a rule definition on the Agentry Client is referred to as "rule evaluation." This evaluation is always performed in some context based on the definition attribute for which the rule is being evaluated, and the data type expected to be returned by the rule. The context of a rule's evaluation will affect what values are in scope for that evaluation and the overall behavior of the rule and its functions.

This behavior is driven by the data type specified by the context of the rule evaluation. A given rule is expected to return a specific data type based on the context in which it is evaluated. The rule will always return a value in that data type. If a caller of a rule expects a string, the rule will return a value with a data type of string. Within the rule, function terms have a similar behavior. Functions will always return a value in the data type asked for by whoever called the function. This caller will either be another function, or the caller of the rule.

The impact of a rule's context on the data type of its return value, as well as the impact of a function's context on its return value's data type, is one that is important in understanding the overall evaluation processing of a rule. Most importantly, it is necessary to understand that not all functions support all return types. If a function is asked for a value in a data type it does not support, it will return a null value in the data type for which it has been asked. It is therefore important to have a clear understanding of both the context in which a function is called, and whether or not that function supports the return type dictated by that context.

Context will also impact what definitions can call a given rule. Rules are normally defined in the context in which they will be evaluated. However, as with most definitions, rules can be referenced by more than one caller. For rules, the data terms referenced by the rule, specifically properties must be available in all contexts in which the rule will be evaluated. A property will be referenced within the rule's structure via a target path. This path must be one that is valid in every context in which the rule is evaluated. If it is not, the rule will return a null value for that property. This is likely to produce unexpected return values from the rule.

Rule Function Terms

Rule function terms, or simply rule functions, are the terms that provide the overall processing of a rule. Each rule has an entry point, which is the term that will return the value to the rule that is then returned to the caller of the rule. The simplest rule definition is one for which this entry point is set to a data term. However, such rules are uncommon as there are few situations in which it is necessary to return a value such as this using a rule.

In the majority of rule definitions, the rule's entry point is a function call. Most functions take parameters. The parameters to a function provide the values the function will process or manipulate in order to produce a value to be returned by that function. This is similar to functions or methods in other languages a developer is familiar with.

Where rule functions are different is in their dealings with data types. The context in which a function is called sets the data type for that function call. A given function may support one or more data types, referred to as the function's "supported return types." If a function does not support the data type dictated by the context, the function will return the null equivalent for that data type.

The context of a function call can also impact the behavior of the function's processing. This impact can include the data type of the function's parameters, as well as how those parameters are processed by the function. Many functions will take parameters matching the data type of the context in which they are called. These same functions will support multiple return types. This means that the function can take parameters of one data type in one context and another data type in another context.

Rule Evaluation at Run-Time

A rule is evaluated at run time on the Agentry Client when the definition referencing the rule calls it. Rules are a purely client-side definition, meaning they only directly impact the behavior of the client application. Rules have no effect on the behavior of the Agentry Server or on communications between Agentry Client and Agentry Server.

When the rule is called is dependent on the type of definition referencing the rule. Rules evaluated to set the label of a button will be called when the parent screen for that button is displayed. Initial value rules to initialize transaction properties will be evaluated when the transaction is instantiated, or possibly when the transaction is applied, depending on the definition of the property's initialization attributes. Other rule uses will result in different events resulting in a rule being evaluated.

The structure of the rule dictates the behavior of the evaluation. This structure is represented in a tree control in the Rule Editor within the Agentry Editor. Following is an example of this structure for a rule that creates an ID value for a new object instance on the Agentry Client:
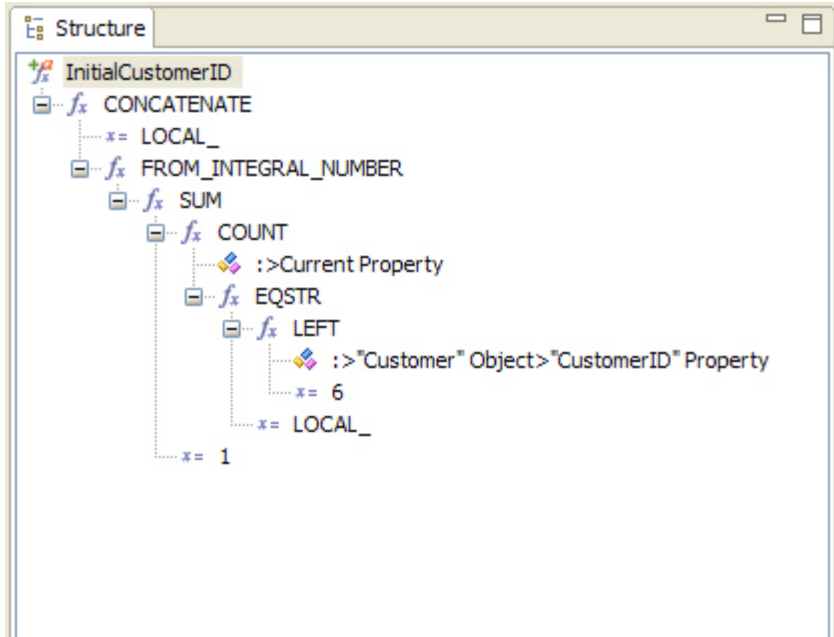
This example is taken from the rule editor and is one of the components of that tool provided to developers for the purpose of defining a rule. In this example, the rule itself is represented by the root node of the tree control, which is named `InitialCustomerID`. As the only child node to this root is a function call to the function CONCATENATE. This function concatenates two or more string values, returning the result. The strings to concatenate are provided as parameters to the function.

In the example provided here the CONCATENATE function takes two parameters. The first is a constant value containing the text `LOCAL_`. This second parameter is a call to the function FROM_INTEGRAL_NUMBER, which is a conversion function that converts integral values to other data types.

Working down through this tree structure there are other rule terms for functions and data terms. At run time, this rule will be evaluated beginning with the innermost term. This term will then return a value to its caller, which will be a function. Each parameter to a function is evaluated in the order provided. In the above example, then, the first term evaluated is the first parameter to COUNT, which is a property denoted as `:>Current Property`. For this rule this refers to a collection of objects to be counted by the COUNT function. The second parameter to the COUNT function is then evaluated, which is a call to the function EQSTR. This function takes another function call as its first parameter, which is the function LEFT. Due to the nature of the COUNT function, the second parameter that is the EQSTR function

call will be evaluated once for each object within the collection referenced by the target path `:>Current Property`. Once COUNT's evaluation is complete, it will return a value to SUM, which will add this value to its second parameter, the constant value 1.

This evaluation continues back up to the top level term, which is the CONCATENATE function. The value returned by the CONCATENATE function will then be the value returned by the rule to the definition that called the rule.

Rule Context

The context in which a rule is called, as well as the context in which the rule terms are evaluated within a rule, will have a significant impact on the resulting processing of the rule and rule terms. The context of a rule definition describes where the rule is being used within the application at run time. The context is set by the definition referencing the rule within the application project and that calls the rule at run time. Included in the context of a rule's usage are the definition referencing the rule, the data type expected to be returned by the caller of the rule, data definitions such as objects, transactions, properties, and others that are in scope when the rule is called and how other data definitions are related to those within the application's overall data structure.

In addition to the caller of the rule, each function within the rule definition will also have an impact on the context, specifically on those rule terms passed to the function as parameters. A given function can dictate the context of the terms used as its parameters including what data definitions are in scope and what data type is expected of the term being evaluated as a function parameter.

The context of a rule and its functions affects the following:

- **Return Type** - Rule function terms do not have a set return data type. Rather, functions support one or more data types for their return values. The caller of the function will dictate which data type is to be returned, and the function will provide a value in that data type. If the function does not support the data type being asked for, the null equivalent of that data type is returned by the function.
- **Target Paths** - Any target paths within a rule are affected by that rule's context. If the property of an object is included in a rule, how that property is found and its value returned is affected by the context of the rule. A rule that contains a reference to an object property will not likely be one that can be reused for a different object. The target path to the property will be invalid.
- **Rule Function Behavior** - Many rule functions will behave differently based on the context in which the function is called. These differences can include both the data type of the function's parameters as well as how the function processes those parameters. Many functions with numeric parameters will evaluate those parameters as the same data type for which the function is being asked. A given function, then, can evaluate its parameters as

integral numbers in one context and decimal numbers in another. Other functions may perform different processing in different contexts based on the data type of each context.

To define the term more precisely, context is the way in which a rule and its terms are called and that affects the behavior, data types, and target path resolution when that rule is evaluated at run time. Context plays a role in the evaluation of each term within a rule, including function terms, data terms , and sub-rule terms.

Rule Data Types

Each term within a rule is evaluated and then returns a value. The value returned by a given term will be in the data type asked for by the caller of the term. All terms will return a value in one of the following data types, which are the only data types available within the rule structure:

- Boolean
- Integral Number
- Decimal Number
- String
- Location (or "GPS Location")
- Property

This list does not restrict the types of values within the application that may be referenced in rules. Other data types, such as those found in property definitions, are converted to one of the above types when the term referencing the property is evaluated. The function terms available for use within the rule structure will return one or more of the above-listed data types.

Data Type Conversion in Rules

When a data term is evaluated within a rule, it will be asked for one of the data types available within rules. The native data type of the term's source may not be one of these six data types. This is most likely to be true when working with property or global definitions. There are far more data types for each of these definition types than those for the rule definition.

When a data term is evaluated within a rule and the data type it is asked for is not one of the six for a rule definition, the value of that term will be converted to one of the rule data types assuming that data term supports such a conversion.

The following table contains a cross reference of all property data types and rule data types. Each rule type and property type intersection in the table indicates whether or not the property type can be converted to the rule data type:

From Property Data Type To...	Boolean	Integral Number	Decimal Number	String	Loca-tion	Proper-ty
Boolean	Yes	No	No	Yes*	No	No

From Property Data Type To...	Boolean	Integral Number	Decimal Number	String	Location	Property
Collection	No	No	No	No	No	Yes
Complex Table Selection	No	No	No	Yes	No	No
Data Table Selection	No	No	No	Yes	No	No
Date	No	Yes*	Yes*	Yes	No	No
Date and Time	No	Yes*	Yes*	Yes	No	No
Decimal Number	Yes*	Yes*	Yes	Yes	No	No
Duration	No	Yes	Yes	Yes	No	No
External Data	No	No	No	Yes*	No	No
Identifier	No	Yes	Yes	Yes	No	No
Image	No	No	No	No	No	No
Integral Number	Yes*	Yes	Yes	Yes	No	No
Location	No	No	No	No	Yes	No
Object	No	No	No	No	No	Yes
Signature	No	No	No	No	No	No
String	Yes*	Yes	Yes	Yes	No	No
Time	No	Yes*	Yes*	Yes	No	No

* - This conversion may not be type safe or requires further explanation on the resulting value from such a conversion. See the description of the rule data type for more information on this conversion.

Boolean Rule Data Type

The Boolean data type within rules is similar to Booleans in all areas of software development, containing a value of true or false.

When converting from an integral or decimal number property type to a Boolean rule type a value of zero is treated as false and a value other than zero is treated as true.

When converting from a string property type to a Boolean rule type, the value of the Boolean will be set to true if the string value is "true." Any other string value is treated as false.

Integral Number Rule Data Type

The integral number data type within rules stores whole positive and negative values, or zero. This is a 32-bit integer value.

When converting from a date property type, or any other data source of type date, to an integral number rule type the value returned will be the number of days from the epoch date of January 1, 1901. Positive numbers represent the number of days after this date and negative numbers are dates before it.

When converting from a time property type, or any other time data source, to an integral number rule type the value returned will be the number of seconds after midnight. This will always be a positive integer.

When converting from a date and time property type, or any other date and time data source, to an integral number rule type the value returned will be the number of seconds from the Agentry epoch date and time of January 1, 1901 12:00:00 am. A positive number represents a date and time after the epoch date and time, and negative numbers represent a date and time before.

A decimal number property can be converted to an integral number rule type. However this conversion is not considered type safe. Any fractional portion within the source decimal number will be truncated from the resulting integral number.

Decimal Number Rule Data Type

The decimal number rule data type stores numeric values with a fractional portion. This is the equivalent to a 32-bit floating point decimal number.

When converting from a date property type, or any other data source of type date, to a decimal number rule type the value returned will be the number of days from the epoch date of January 1, 1901. Positive numbers represent the number of days after this date and negative numbers are dates before it.

When converting from a time property type, or any other time data source, to a decimal number rule type the value returned will be the number of seconds after midnight. This will always be a positive integer.

When converting from a date and time property type, or any other date and time data source, to a decimal number rule type the value returned will be the number of seconds from the Agentry epoch date and time of January 1, 1901 12:00:00 am. A positive number represents a date and time after the epoch date and time, and negative numbers represent a date and time before.

String Rule Data Type

A string rule data type stores one or more unicode characters as a string value.

When converting a Boolean property to a string rule type, the resulting value of the string will depend on the definition of the Boolean property. The attributes within the Boolean property **True Value** and **False Value** contain the text value returned by that property in a string

context. A data term for a Boolean property evaluated in a string context will then return the appropriate string depending whether or not the property is set to true or false.

When converting an external data property to a string rule type, the return value will be the full path and file name for the file referenced by the property. If the property does not reference a file, an empty string is returned.

Location Rule Data Type

The location rule type stores a value returned from a GPS unit that includes the latitude, longitude, number of satellites, and precision of the location value. This data type cannot be converted to other data types within the rule and other data types cannot be converted to a location.

There are specific rule functions within the System category of rules provided to work with the Location data type. These include functions for converting two decimal values assumed to be latitude and longitude coordinates to a location value, as well as those for calculating distances between two location values, and a function to retrieve a GPS location from the GPS unit for the client device.

Property Rule Data Type

The property rule data type is unique among the different data types within rules. A property type is the term applied to any point within the rule where a definition is expected. Certain functions are provided to allow for searching object collections, or to work with external data properties. These functions will take one or more parameters of type property and may also return a value of this type.

Many of the different functions for these types are intended for use with certain types of definitions within the application. The information for these functions indicate the expected definition type. The important concept for a property rule type is that what is returned from such a term is the definition itself. When a target path references a definition for a caller expecting a property rule type, that definition is returned, not just its value.

Rule Editor Introduction

Rule definitions within an Agentry application project are added to the module and defined using the Rule Editor. The Rule Editor is a tool used within the Agentry Editor to define the rule logic. It includes several tools and features to aid in the definition of this logic, including:

- Functions are presented with fields for each parameter to the function, with appropriate indicators for optional and required parameters, as well as the data type of the parameter.
- On-line help for each rule function term, including short and long descriptions of the function and descriptions of each function parameter displayed in the Rule Editor for each item as it is used.

- Test functionality providing the ability to test rules within the rule editor by providing test data for function parameters and viewing the return from the rule based on such values.
- Real-time rule structure validation and context information for the rule as a whole and each of its terms.
- Navigation items to make the different rule terms more readily accessible.

Overview of the Rule Editor

Following is an example of the Rule Editor. It displays a simple initial value rule that generates local ID's for new object instances created on the Agentry Client.
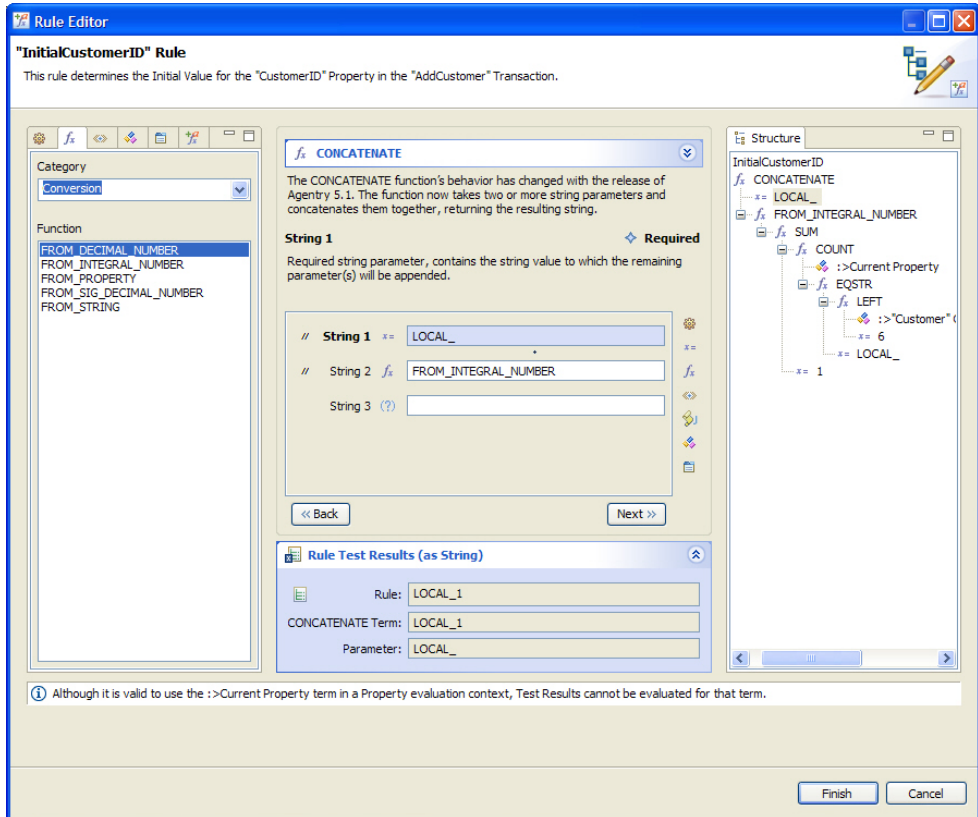
On the left of this screen there is a list of all items that may be added as rule terms. The various tabs allow for the selection of actions, functions, globals, properties, screen sets, and sub-rules. Selecting one of these tabs will then list the items of that type. When functions are selected a drop down list is displayed to allow the developer to select the function category, which will list only those functions within that category. There is also the option to list all functions.

The main center portion of the screen displays the function currently being added or modified. Included here is the short description of the function, as well as fields for each of the function's

parameters. Selecting a parameter field will display the description of that parameter above the parameters list.

To the right of the parameters list are shortcut buttons. These buttons allow for adding the same terms as the list on the left. When the shortcut button is selected, a menu is displayed with the items of that type. In addition is a button to include a JavaScript term. This should only be used with the JAVASCRIPT function and is provided to allow for the entry of JavaScript in a multi-line editable text box.

Below the list of field parameters is the Rule Test Results section, which may be expanded or collapsed (expanded in the example). This section can be used to test the results of the rule by specifying test values for various terms within the rule.

To the right of the screen is the Structure view for the rule. This displays the entire rule structure. For developers familiar with the rule editor in previous releases of Agentry, this view presents the rule and allows for the same functionality. Within this structure view, the rule terms may be added, edited, deleted, or dragged and dropped to different positions within the structure.

Creating Rule Definitions

To define a new rule definition within the Editor, the Rule Wizard and Rule Editor are used. The Rule Wizard is displayed to capture the attributes for the new rule definition, including the name and group. The Rule Editor is displayed next to allow for the definition of the rule's structure. This procedure uses an initial value rule for a transaction property as an example. The same process is followed to define a rule regardless of where it is to be used within the application.

1. Start the Add Rule Wizard by selecting the Add Rule menu item in the menu displayed for the attribute to reference the rule definition. This is normally an ellipses button to the right of the attribute field.

   The first screen of the wizard is displayed with a default name and group, based on the definition referencing the new rule definition:
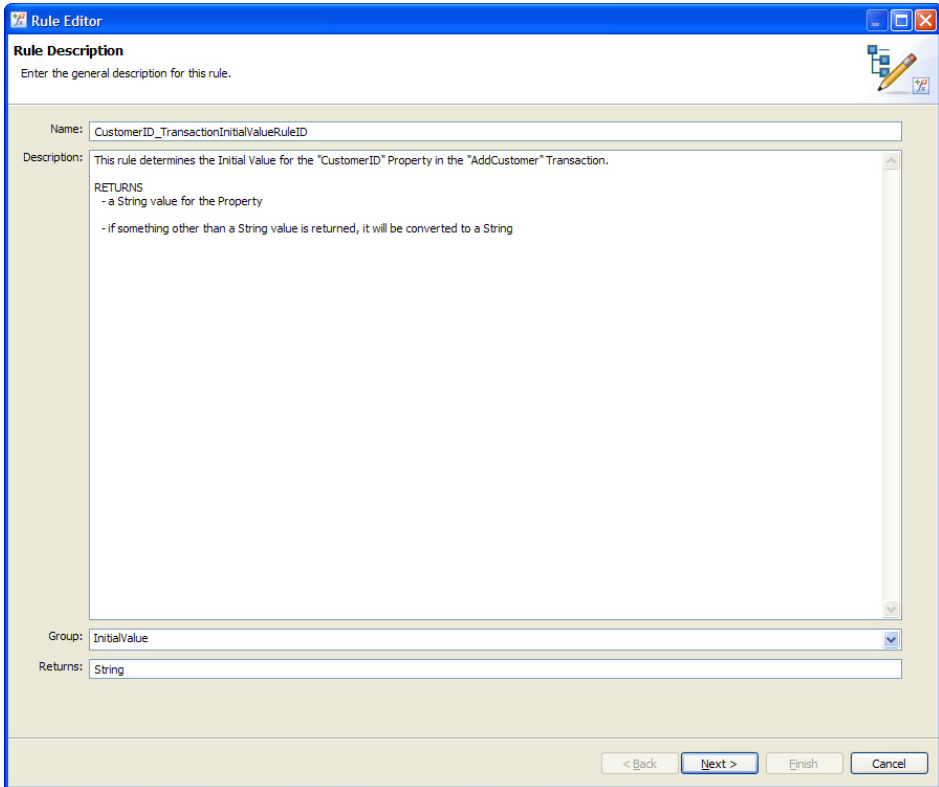
2. Set the Name and Group attributes as desired for the new rule definition. The Description field may also be edited. The default text display is based on how the rule is to be used. The Returns field is read-only and specifies the data type of the value to be returned by the rule when it is evaluated. Click the **[Next >]** button to advance to the Rule Editor.

The second screen of the Rule Editor is displayed with the Rule Entry Point selected:
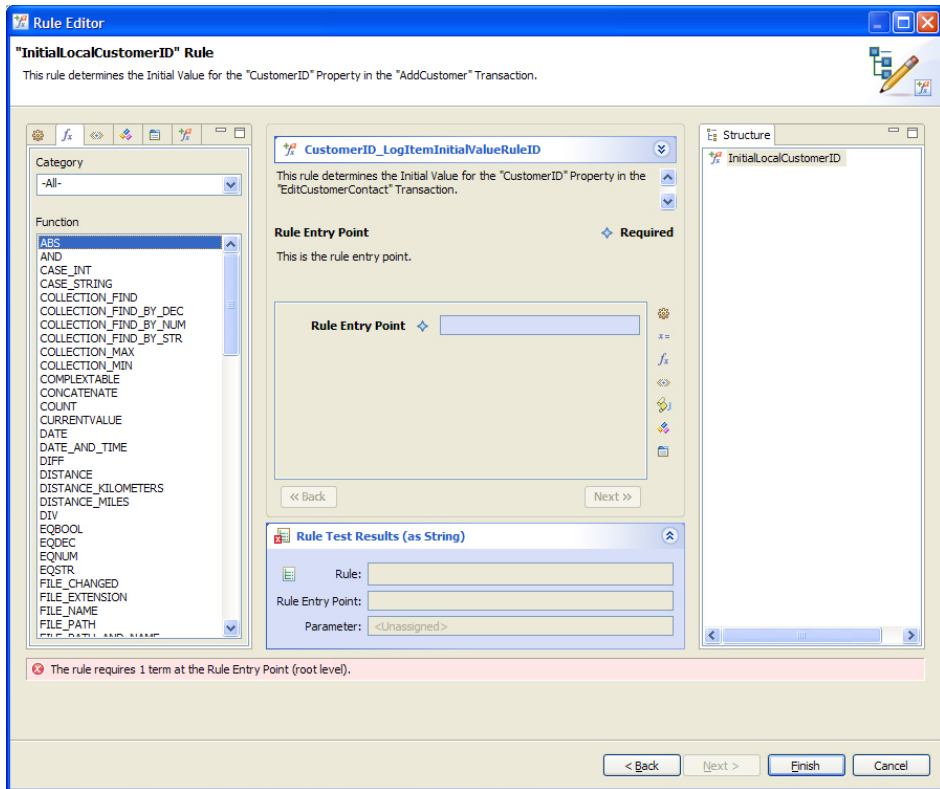
3. To begin defining the rule structure, begin by selecting the field Rule Entry Point. This is the first term for the new rule definition. Selecting this field will allow for the addition of a rule term, either a function or a data term. This term's return value will be the value returned by the rule at run time.

4. To add a term to the entry point in the rule, select an item from the list of terms on the left. By default the list displays the available rule functions. In most cases this is a rule function. Other options include an action, global, property, screen set, or sub rule. To change the list of terms to select from, select one of the available tabs above the list.

5. If any term is selected other than a function, the rule's definition is complete, as no other terms can be added below a data term within the rule's Structure. If a function is selected from the list, the editor will display that function in the center of the screen, with fields listing the function's parameters. The function name followed by its short description is shown at the top-center of the screen. Clicking the name of the function, or the arrows to its right will display the function's long description.
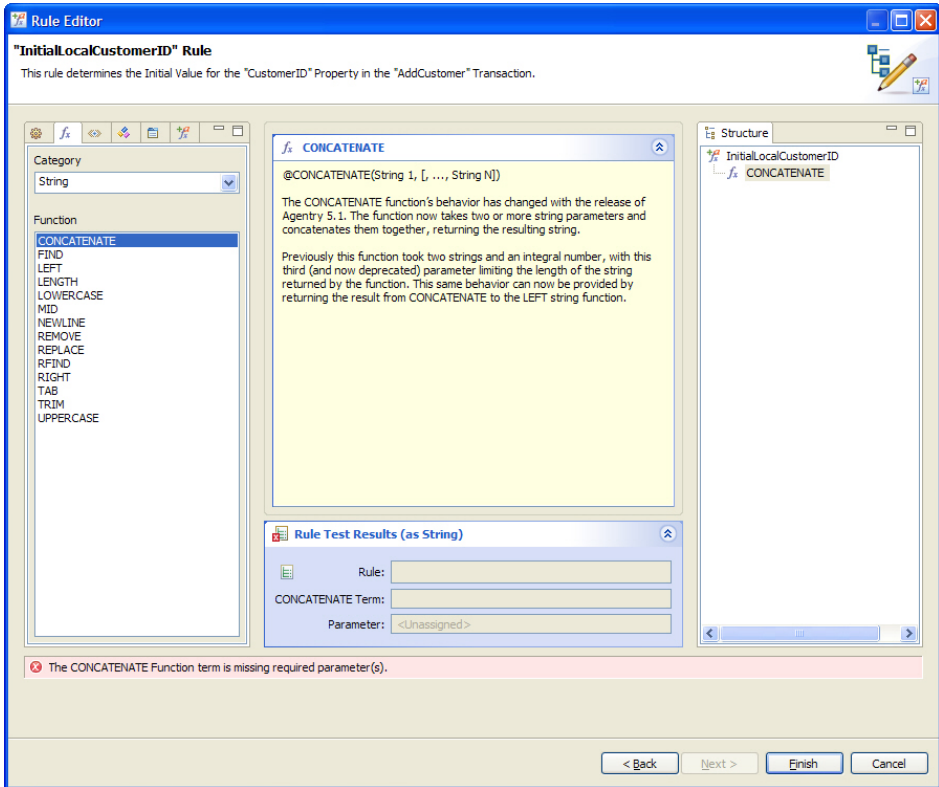
6. When a parameter field is selected in the Rule Editor the description of that parameter is displayed. Selecting a field will display the list of terms on the left. Any term that supports the return type for the parameter may be selected:
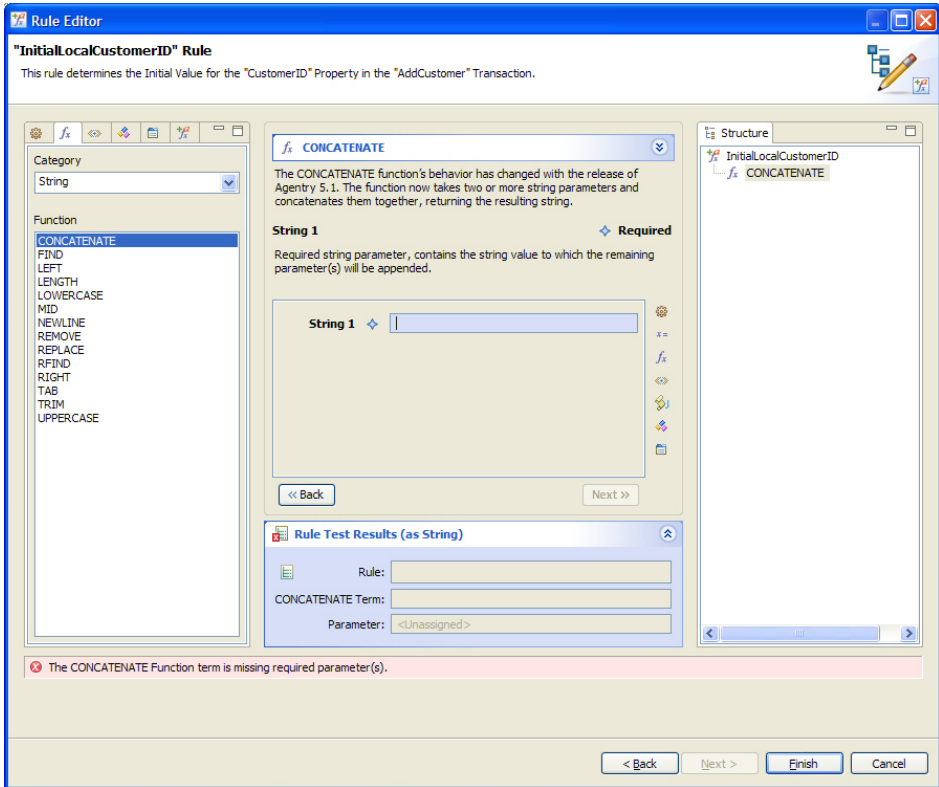
7. To add a constant value as a parameter to the function simply type that value in the parameter field.
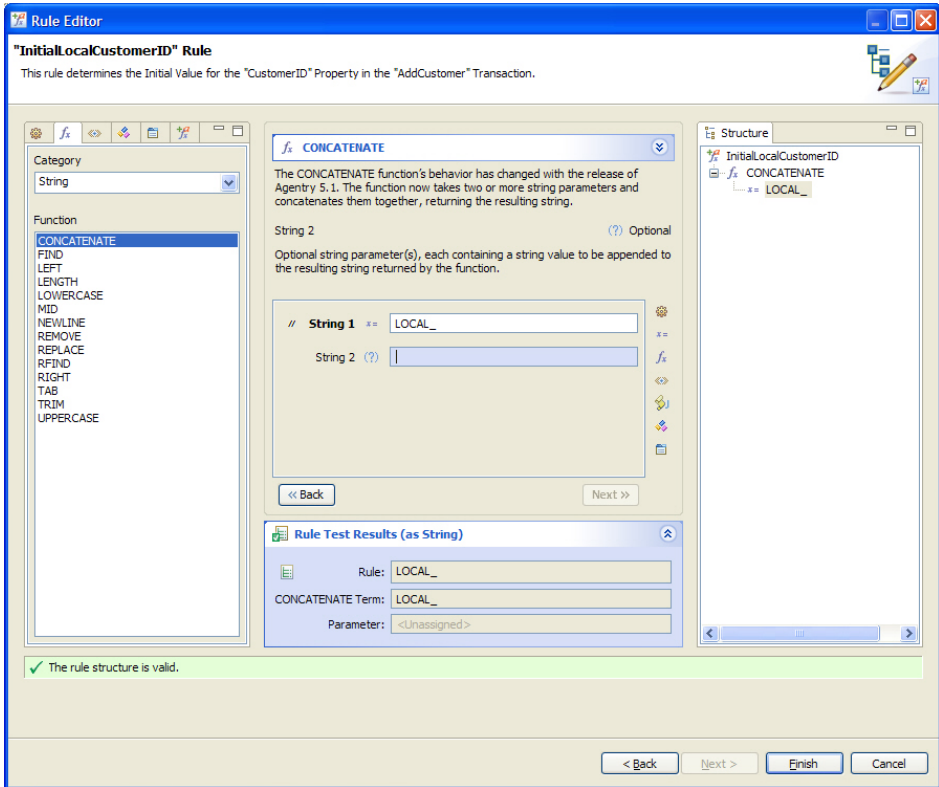
8. To add any other term type, select it in one of the lists on the left side by double-clicking it. If a function is added as a parameter to the current function, that function will then be displayed in the middle portion of the screen, along with its short description and list of parameter fields.
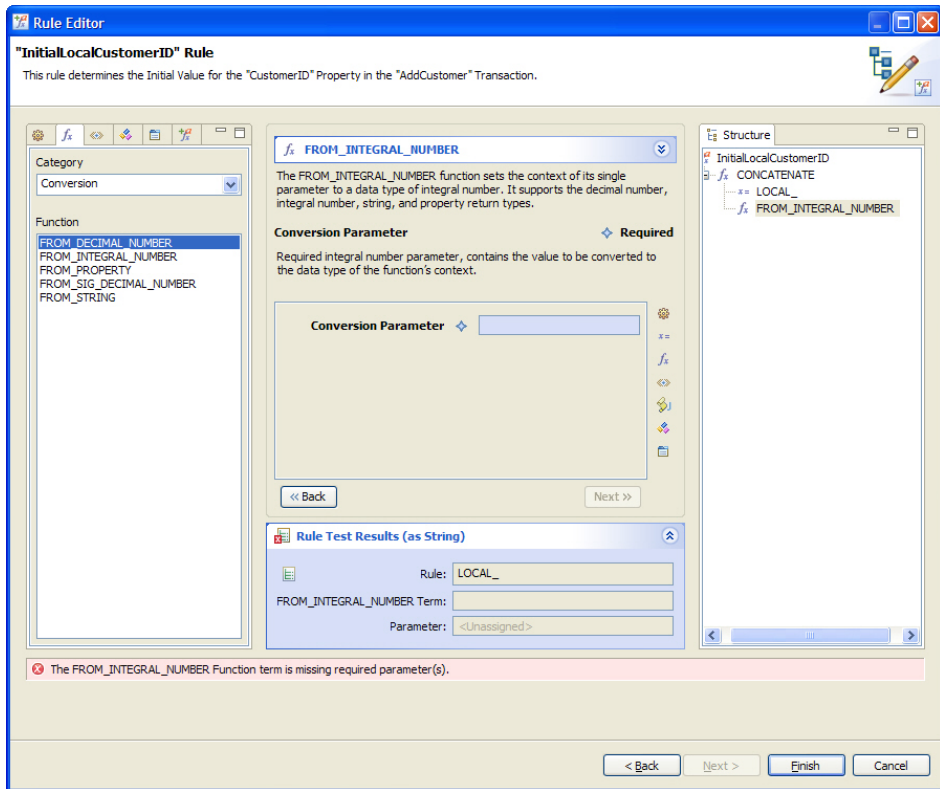
9. At this point the process is repeated until the structure of the rule has been defined.

Note the structure view to the right of the rule editor. As functions are added, their position within the overall rule structure is represented. This structure view can be used in the definition of the rule as well. Right-clicking on any function in the rule structure displays a popup menu allowing for the addition of parameters to this functions. The menu also provides options to replace terms and delete them. Additionally, terms may be dragged and dropped to different locations within the structure if it is desired to modify the rule in this manner.

Next

Once the rule has been defined it can be tested within the Rule Editor. This can be done before finishing the Rule Editor, or the developer can return to the definition later and perform any testing.

Testing Rules in the Rule Editor

The rule editor, in addition to providing the interface to define and edit rule definitions, also provides tools for testing rule definitions. Within the Rule Editor, below the list of function parameters, is an expandable frame labeled "Rule Test Results." Expanding this frame will display the return value for the rule as currently defined, with certain assumptions made about

return values within the rule. Using this test frame, the developer can select a parameter and then provide a test value for that parameter. The return value of the rule will then be determined and displayed, using the new test value.

1. Within the Rule Editor, expand the Rule Test Results frame. Within the Structure view select the rule function term for which a parameter test value is to be specified. The function will be displayed in the middle of the screen. Select the parameter for which a test value is to be entered.

   Within the test results pane, fields are displayed for the values returned by the rule, the currently selected function term, and the currently selected parameter to that function.
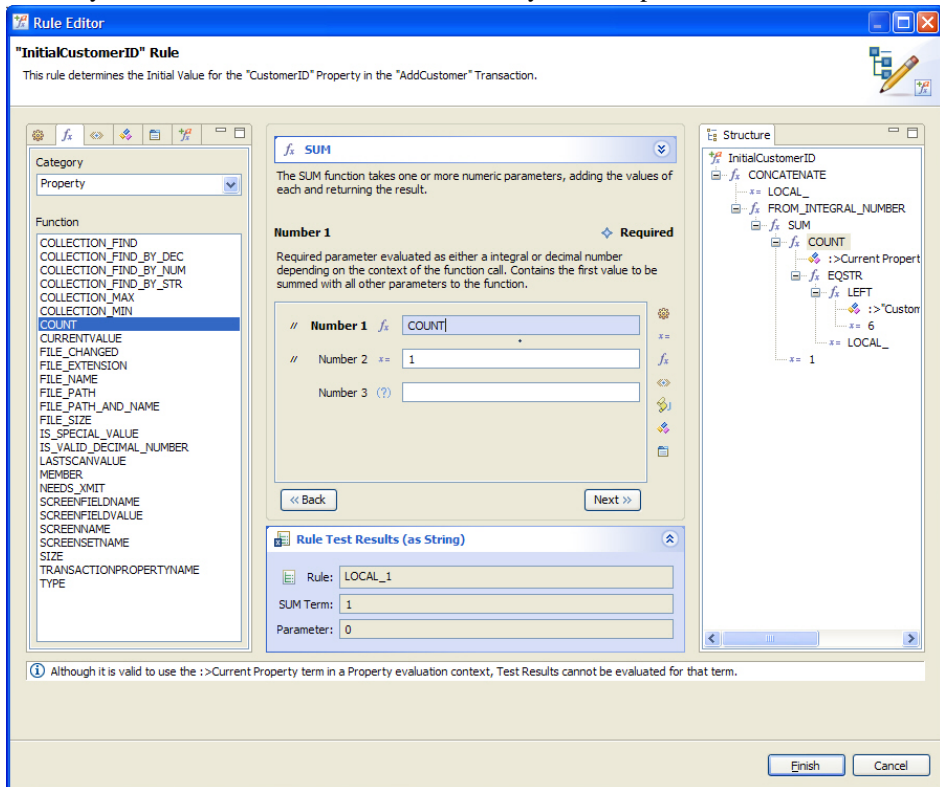
2. To change the test value for the selected parameter, click the button to the left of field Rule within the Rule Test Results frame.

   A screen is displayed allowing for the entry of a test value. When entered, the return values for the function term and the rule will be updated automatically:
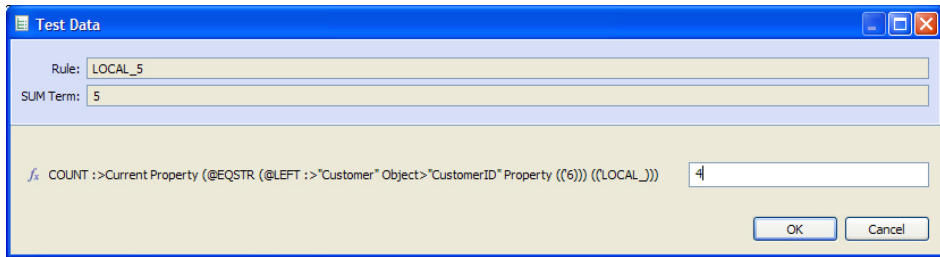
3. The Rule field displays the value the rule will return based on the test data entered. The line below it displays the value returned by the function, again based on the entered test data. Below these fields is the section where the test data can be entered. The fields displayed here include all values under the selected term which may be replaced with test data. Excluded from this list are constant values and object collection properties. Changing any of the values will automatically update the return value from the selected function term and the rule as a whole.

The results of entering test data will not affect the rule definition in any way. However, proper use of the test functionality can significantly improve the stability of the rule before it is published for development and unit testing. Additionally, areas of logic for which the proper structure is unclear can be made easier when using the test data to determine if the expected values are actually returned by the logic within the structure. Note that while this functionality is a powerful development tool, it is not intended to replace the proper run-time testing that should be a part of any software development or product implementation project.

Syclo Data Markup Language

When synchronizing data between the mobile application and the back end system, it is necessary to have access to the mobile application's data values. This access is provided in Agentry using the Syclo Data Markup Language, or SDML. The SDML is a markup language consisting of tags that provide access to the data values of the mobile application. Additionally, the SDML includes a full set of functions, or function tags, that can be used to perform logical operations in relation to this values or to drive the overall logic the Agentry Server will execute against the back end system.

The SDML tags used during synchronization are a part of the text within the scripts for step definitions defined for SQL Database, HTTP-XML, and File system connection types. Also, the synchronization components of data tables and complex tables for each of these system connection types can contain SDML. In addition to SQL Step definitions, other `.sql` script files run by the Server may also contain SDML tags. Steps defined for Java Virtual Machine system connections also include the ability to access SDML tags, but these tags may not be contained directly in the source code of the Java Steplet files used by these steps.

The Agentry Server will pre-process the script files of steps containing SDML markup. This processing is referred to as tag expansion. Each tag within the script is expanded, with the

value it represents replacing the tag at the exact position of that tag within the file. Function tags are expanded with the results of their expansion being placed in the exact position of the function call within the file. Once the tag expansion has completed, the resulting text is submitted to the back end system for processing.

The two categories of tags within the SDML are data tags and function tags. Data tags represent data values available to the script file based on when it is executed. This information must be known when writing the script in which the SDML will be contained. For step definitions the values in scope are dictated by the step usage definition running them. For .sql scripts run by the server, but not a part of the step definition, the values in scope will vary depending on how that script is used. Certain values are globally available, such as the user ID as entered by the user to log into the Agentry Client.

Function tags are globally available, with certain exceptions. Function tags provide the logical, mathematical, string manipulation, and other similar functionality to the SDML. Function tags can take values passed in as arguments, parameters, or expressions. These values are processed by the function during tag expansion, with the resulting value of the function call being placed within the script.

Following is a basic example of a simple SQL statement containing SDML data tags:

```
SELECT
    A.FIELD1,
    B.FIELD2,
    C.FIELD3,
FROM
    TABLEA A,
    TABLEB B,
    TABLEC C
WHERE
    A.NAME = '<<user.agentryID>>'      AND
    A.ACCTNUM = '<<object.acctnum>>'      AND
    B.ACCTNUM = A.ACCTNUM        AND
    C.ACCTNUM = B.ACCTNUM
```

In this example, the value <<user.agentryID>> is replaced with the user ID as entered when the user logged into the Agentry Client. The data tag<<object.acctnum>> will be replaced with the value of the acctnum property of the object currently being processed.

SDML Syntax and Data Tag Expansion

The SDML is a markup language containing data tags and function tags. The basic syntax for a tag is to enclose a given tag within the tag markers << >> to denote it as an SDML tag. Within these tag markers is the name of the tag, as well as any values that may be a part of the tags expansion processing. Following is the general form of data tag and function tag syntax:

Data Tag

```
<<parent.tagName.parameter namedParameter="value">>
```

Function Tag

```
<<functionName argument "expression" namedParameter="value">>
```

Named parameters to both functions and data tags have the specific requirement that the parameter name, equal sign, and the value can not be separated by any white space:

Correct Parameter Syntax

```
<<functionName namedPrameter="value">>
```

Incorrect Parameter Syntax

```
<<functionName namedParameter = "value">>
```

Depending on the nature of the SDML logic and processing needed, it is common for one tag to be nested within another. When this is the case the end markers for such tags may be adjacent. In this situation at least one white space character must be used to separate the two end markers. This may be a space, tab, or a newline:

Incorrect

```
>>>>
```

Single Space

```
>> >>
```

Tab

```
>>          >>
```

Newline

```
>>
>>
```

Data Tag Syntax
Values within a given tag will depend upon a number of different factors. Data tags generally take one of the following forms:

```
<<parent.tagName namedParameter=value>>
<<parent.tagName.parameter>>
```

The parameter and named parameter for a given data tag will depend on the data type of that tag. Some may have no parameters or named parameters, others may support one or both. A parameter generally provides access to different contents of the data tag's value, such as raw or

string. A named parameter generally provides formatting instructions for how the value should appear when the data tag is expanded.

The value for a named parameter may be a hard coded value or another tag within the SDML. When the value is plain text, it must be enclosed in double quotes. When the value to the parameter is another SDML tag it cannot be enclosed in double quotes.

Parameter from Plain Text

```
<<parent.tagName namedPrameter="value">>
```

Parameter From Data Tag

```
<<parent.tagName namedParameter=<<tagName>> >>
```

Function Tag Syntax
Function tags within the SDML take the general form:

```
<<functionTag argument "expression" namedParameter=value>>
```

A given function may take multiple arguments, expressions, and/or named parameters, or it may not take any of these depending on that function's prototype and purpose. Separating each of these values to the function is one or more white space characters.

Arguments should be enclosed in tag markers if a data tag is used, unless specified otherwise for a given function. Certain functions, notably `<<if...>>`, `<<foreach...>>`, and `<<case...>>` specify that if the first argument is a data tag it cannot be enclosed in tag markers, but rather should only be the name of the data tag. If the argument is a hard coded value it should be enclosed in double quotes.

Expressions are always enclosed in double quotes, regardless of whether or not they contain SDML tags. The contents of a given expression can span multiple lines, which is often the case as expressions tend to be longer text values.

The value for the named parameter can be a data tag, in which case the tag should be enclosed in tag markers. If the value is a hard coded value it must be enclosed in double quotes. The value for a named parameter can contain white space within the double quotes and can be a combination of SDML tags and plain text.

These values for a function tag can span multiple lines, with the opening marker preceding the function name and the closing marker somewhere after all specified values for the function, as in:

```
<<functionTag
      argument
      "expression"
      namedParameter=value
>>
```

When arguments or named parameters contain function or data tags, those tags will be expanded before being passed to the function for processing. Expressions containing tags will not be processed until the function returns that expression.

SDML Expansion

At run time, when the Agentry Server processes a script file the tags it contains are parsed and expanded. This process is called SDML expansion and occurs for all scripts not using a Java Virtual Machine system connection that are run by the Server.

Tags are expanded in a top-down, inside-out order. This means that each line of a script is processed starting with the first in the file and working in order to the last line. When a line is processed, the data tags are expanded from the innermost tag to the outermost one. Consider the following example:

```
3.....2....1        ..................
1a..................                                      4
```

```
<<if <<ne <<object.acctnum>> <<parent.acctnum>> >> "not equal" >>
```

Ignore the numerical notations for the moment, as they are for reference purposes only and not a part of the SDML text. This line says that if the values of the acctnum properties in the object and the parent of the object are not equal to return the string "not equal". This begins with the <<if...>> function. The single argument to this function is <<ne...>>, which is another function whose name is short for "not equal". The <<ne...>> function takes two arguments that are compared for equality. The two arguments in the example are both data tags for property values within objects.

In this case, the expansion goes as follows. First, the two arguments, noted as 1 and 1a, are expanded. These are data tags, so the tags are replaced with the values of the two acctnum properties in the object being processed and the parent of that object. If the object property acctnum has a value of 1234 and the parent has a value of 1122, the line would expand as follows:

```
<<if <<ne 1234 1122 >> "not equal" >>
```

Next, the <<ne...>> function is expanded. The two values of 1234 and 1122 are passed as arguments to this function. The function then compares the two values and determines they are not equal. The line would now then appear as:<<if true "not equal" >>

The <<if...>> function, which provides if-then-else logic, takes the return value from the <<ne...>> function value as an argument. The "not equal" text is the expression that is returned when the argument is true. In the example provided the resulting text placed at the point of the <<if...>> function call will be the text: not equal.

SDML Syntax Quick Reference

Following is a quick reference of the basic syntax rules for the Syclo Data Markup Language:

- Function and data tags are enclosed in the tag markers `<<tagname>>`. Named parameters to both function and data tags, as well as function arguments and expressions are enclosed within the same set of tag markers.
- Hard coded values passed to parameters or arguments are enclosed in double quotes. Data or function tags are enclosed in tag markers and should never be enclosed in quotes.
- Expressions are always enclosed in double quotes, whether or not they contain tags. Tags within an expression are also enclosed in tag markers.
- Named parameters for both function and data tags take the form of a key and value pair separated by an equal sign (=). No white space can exist between the named parameter, equal sign, and the beginning of the value for the parameter. The parameter itself can contain white space and, when it does it should be enclosed in double quotes.
- Adjacent end tag markers must be separated by at least one white space character. Excluding this character will result in an error during tag expansion.
- Function tags can span multiple lines in a script file. This is commonly the case with expressions.

Agentry Data Definitions Overview

In any application the data structures and the processes to synchronize the production data are the foundation of the functionality. Within an Agentry application project there are three definition types intended to define data stored on the Client: objects, complex tables and data tables. All three define the data for the application and also include components for synchronizing data with the back end system.

Objects exist at the module level and complex and data tables are defined at the application level. Complex tables and data tables are defined to store lists of records on the Client and normally contain values displayed to the users in lists or other controls from which they can make selections. Objects store the production data for modules and normally encapsulate some business entity.

While objects, complex tables and data tables are the main data definitions, there are others related to the storage and synchronization of data on the Client. The first of these are object properties. An object property defines a single piece of data for the parent object. Note that there are also transaction properties, which are similar to object properties but are defined within a transaction. The discussion of properties here will focus on object properties.

Other definitions related to production data are those defined to synchronize the data. The primary definition for data synchronization is the step definition. A step defines a piece of processing to be performed by the Agentry Server with a specific back end system. Steps are used by other definitions that are processed during data synchronization. This allows for reusability as well as the multi-system support provided by Agentry. Steps are defined to synchronize data stored in objects and transactions.

The synchronization process for complex and data tables is defined within the definitions themselves. Both complex tables and data tables contain components responsible for the downstream, or back end-to-client data synchronization.

Data Synchronization Overview: The Exchange Data Model

When developing mobile software solutions, one of the primary considerations is the most efficient way in which to synchronize production data. While all client-server systems must account for this, mobile software development presents its own set of challenges, which stem from the almost universal truth that, at some point, mobile users will need to work in a disconnected environment.

Users will not always be connected, and in many cases will spend most of their day without network connectivity. Therefore, when users do synchronize their clients, information must be resolved concerning what data a user needs. It can be extremely inefficient to attempt to retrieve all production data during synchronization. Most production applications contain large amounts of data stored on the client devices and attempting to retrieve everything during synchronization can result in long delays during the synchronization process. This is unnecessary in most environments, as much of the production data stored on the client is likely to still be current and accurate as compared to the data in the back end system.

An Agentry application project accounts for this within its structure. The architecture of all synchronization components allows the developer to be far more selective about what data needs to be retrieved during the synchronization process. The method recommended by Syclo is called the Exchange Data Model.

Exchange data is the term used to refer to information about what production data the client has and when it was last retrieved, as well as the data contained in the back end system and when it was last modified. With this information available the developer can implement synchronization processes that only retrieve information that has been modified since the last time a client synchronized with the Server. Any unchanged data is not retrieved. This model will result in quicker and more efficient synchronization for users, as well as reducing the amount of resources needed by the system as a whole during the synchronization process. All data definition types and their related synchronization components allow for and are intended to be used in an exchange data model.

The use of the exchange data model requires certain information be available during synchronization. This information can include:

- The date and time when an object, complex table record, or data table was last downloaded to the Client.
- The date and time when the data in the back end system was added or last modified.

For the date and time of data retrieval on the clients, the synchronization processing within Agentry provides the ability to retrieve, store and access the client-side information about

when data was last retrieved. Objects, complex tables and data tables all have the ability to store what is called the "last update" value that represents the date and time data was retrieved.

For date and time values related to changes made on the back end system, mechanisms must exist or be added for the mobile application to track changes to data that occur in between users' synchronizations.

During synchronization the exchange data about which data has been retrieved, changed, and added, and when those events occurred is put to use according to the following general process. Note that this applies only to downstream synchronization:

1. The Agentry Client sends the information to the Agentry Server about what data it currently contains and when it was retrieved. This includes unique identifiers and the last update values for each data instance.
2. The Agentry Server processes the client-side exchange information according to the synchronization definitions. This can involve adding the data to back end objects created specifically for the mobile data synchronization, or by using existing back end objects that suit these purposes.
3. The Agentry Server next processes the synchronization definitions that determine what has changed in the back end system since the date and time for the client-side data. Comparisons are made between the client's date and time values and the date and time values in the back end system that reflect when the back end data was last affected. Items with date and time values more recent than matching items on the client, or items added to the back end not currently residing on the client, are flagged for retrieval. Alternately, and depending on the synchronization methods specific to the type of back end system, the comparison and retrieval may be accomplished at the same time. This data is retrieved using synchronization definitions and returned to the Agentry Server.
4. Data that should be removed from the client is determined separately from data that should replace or be added to the client. Definition types within the Agentry architecture exist to specifically look for and return items that should be removed from the client.
5. The Agentry Server builds object instances, or complex table and data table records based on data returned to it. These instances are then sent to the client to be stored in their respective structures.
6. Data to be removed from the client is denoted via it's unique identifiers. The ID's are sent by the Agentry Server to the client. The messaging sent includes instruction to the client to remove the denoted item from its respective data structure. This includes deleting object instances or removing complex table records. Note that individual records cannot be deleted from data tables, for reasons explained in the discussions specific to this definition type.

The specific methods and mechanisms for accomplishing the above tasks will differ as a result of a combination of different factors that include the type of production data being synchronized (objects, complex tables, or data tables), the type of back end system in use, the capabilities of a specific back end system, the specific needs of a given application, and the specific needs of an implementation of a given application.

Regardless of the technical details of how the above steps are accomplished, the following summary of the exchange data model holds true. Begin by determining what the client has and

when it received it. Next use this information to determine what is different in the back end system and when it was changed. Finally, retrieve only the data which is different on the back end. Any other data in the back end system can be ignored as it has not been modified and therefore is accurate and still current on the client.

Data Synchronization: Data Filtering Overview

When developing a mobile application the concept of data filtering should always be at the forefront of the developer's mind during all phases of the development life-cycle. Data filtering is the term used to refer to filtering the data provided to the mobile user so that unnecessary and unneeded data is not retrieved. When the proper data is retrieved for the user and unneeded data is excluded, the application ultimately provided will be far easier for the end users, and will operate more efficiently during synchronization and client-side operations.

The need for data filtering in mobile software development is driven by two main factors. First, while mobile devices continue to become more powerful and more sophisticated, they do not have the same capabilities as a traditional personal computer or work station. Attempts to store large amounts of data on such devices can result in, at the least, poor performance of the application, and at worst the client device can become overwhelmed and not function at all.

The second factor in the need for data filtering is the end user. Many users of mobile software need only certain information concerning a particular business entity. Additional information can result in a cluttered user interface as well as confusion on the part of the end users.

For these reasons as well as others the concept of data filtering can and should be applied to all areas of the application design and development process as it relates to the data structures.

Overall there are many areas in which data filtering should be applied and there are often many options available on how to do this. Which is used will depend on the type of data and what information is available about the data in the back end system. It is important to remember that the client device is not a permanent data store for the enterprise system in use. Rather, it is both a snapshot and a subset of data from that enterprise system.

Object Data Filtering: Property Definitions
Objects contain the child definition property. A property stores a single value for the object. A given object will contain multiple properties. When designing an object and the properties it will contain, it is important to consider what data the user actually needs.

Using a database back end system as an example, where an object is defined to contain data from a given table, the developer should always consider what data the end user will need from that table. Many database tables in a back end system can contain dozens or more columns. This data is necessary for the records within the table and is likely related to multiple processes. Examples of these processes and needs can include performance reporting, accounting requirements, change tracking, and auditing. However, much of this data is not

needed by the end user. It will not be displayed to them nor captured from them on the Client. Furthermore it is not needed during downstream or upstream synchronization.

Because of this, there is no need to retrieve this data from the back end system. Though a given value may be small in size for a particular object, remember that it is likely that there will be dozens or hundreds of instances of a given object stored on the Client at a given time. As a result, a single unneeded value for an object can result in significant wasted resources on the Client for storage, as well as unnecessary bandwidth and processing being consumed during synchronization. These same statements can be made about any object for any type of back end system.

Complex Table Data Filtering: Field Definitions

Furthermore, complex tables should also be designed and developed with data filtering in mind. Complex tables contain field definitions, with each record in the table containing the fields defined for the table. Like properties, a single unneeded complex table field can result in significant wasted resources on the client device. With complex tables, however, the resources wasted can be even more detrimental than with objects. While there may be hundreds of instances of a given object stored on the Client, there can be thousands of complex table records.

User-Specific Data and Data Filtering

Taking this a step further, entire object instances or complex table records may be excluded from the Client if the proper design and development considerations are applied as they relate to data filtering. Whenever possible the developer should consider what data can be user specific. For objects this tends to be the case most of the time. Objects are usually defined to encapsulate business entities in the back end system that are user specific. A work order is assigned to a specific technician. A customer is assigned to a single account executive. These are two examples of what would normally be object definitions in a mobile application.

Complex tables tend to store data that is applicable to multiple users. Complex tables may contain records of inventory items available to be ordered by any customer, or assets for the company that one of many technicians may work with. However, there are still ways to filter this data. First, in some cases the data of a complex table may be user specific. In this case the data can easily be filtered for a given user during synchronization.

In other more common situations, the data is not user specific. In these cases the developer should look for other ways to partition data. Some suggestions can include the location a user may work in can mean certain records will never be needed. If a technician works in location A, then the complex table containing assets need only contain the assets that reside in location A. Similarly inventory items customers may order can also be filtered. If an account manager services customers in a specific industry or of a certain type, there may be items within the inventory that those customers will not order. Assuming the inventory information can be cross referenced with an industry or customer type, records can be excluded from the complex table containing that data on the Client.

Object Development Concepts and Considerations

An object definition encapsulates a business entity and its related data. An object's child property definitions give that object its characteristics. An object can also define how its data is retrieved from the back end system. The object definition is the primary data definition for modules. At run time objects are instantiated during synchronization by the Agentry Server, which then transmits those instances to the Client. Object instances can also be created at run time on the Agentry Client via add transactions.

The object definition contains only a few attributes related to its identifying value, or "key property," and the value displayed for the object during synchronization. The heart of an object definition lies in its properties. An object property defines a single piece of data for the parent object. The definition of an object property should always match the aspects and behaviors of the back end value it is created to store. This includes data type, data sizes, and the name.

The name of the object property should match the name of the back end value whenever possible. The Server matches the values returned by any back end steps to the properties within the object by matching the names. Any value returned from the back end whose identity does not match the name of a property in the object is discarded. Any object property that does not have a corresponding value in the return set from the back end is initialized to null upon object instantiation.

If it is not possible to match the property name with the back end value's identifier, the value should be aliased in some manner within the return set. As an example, in SQL select statements a field can be aliased using the AS keyword, as in `Field1 AS Name1`. In Java the data structure containing return values can be named to match the properties.

The definition of an object should also include properties that may be needed on the Client side only. These values may be used for client-side processing or behaviors. In this case the values of these properties will be initialized to null when objects are instantiated during synchronization. They can be set via transactions on the client at run time.

Object instances are stored on the Client in one of two ways. First, a single instance of an object can exist as a property of another object. Second, and far more common, is to store objects in a collection property.

Objects are synchronized with the back end system via the module level definitions fetch and push. A fetch defines how the Agentry Server synchronizes data for a target object collection by referencing the step definitions to perform this task. A fetch is processed during synchronization between the Client and Server, with the results being the retrieval of new object instances for the target collection, replacement of existing objects within the collection, and the removal of objects from the collection. All of these determinations are based on the definition of the fetch and its child step usage definitions.

A push defines when it is necessary to push an object in real time from the back end system to the Agentry Client and how that object's data is retrieved. Pushes are used only when a constant network connection can be maintained between the Client and Server. Like a fetch, a push targets an object collection property and will synchronize object instances within that collection. Objects can be added, replaced, or removed from the collection based on push processing. Differing from a fetch, pushes are run asynchronisly by the Server and pushing objects to the Client when changes to the back end system are made.

Another definition type involved in synchronizing objects are object read steps. An object read step references a step definition run to retrieve data from a back end system to populate an object's properties. When the synchronization process is defined using the exchange data model, it is often the case that the fetch is defined to determine what objects do and do not need to be retrieved, and the read steps are then run to perform the actual data retrieval. This is a common practice but not a requirement of the development. The fetch can be defined to accomplish both tasks without involvement of the object read steps. Likewise, push processing can involve running the read steps of an object to retrieve the object data from the back end system.

Regardless of whether a push or fetch is used, and also whether or not read steps are involved in the process, object data synchronization includes both the object collection targeted by the fetch or push, as well as any collection properties that are descendents of the objects within the targeted collection. Any collection property that is not defined within the main object, but rather as a descendent of the main object, is termed a "nested collection." Nested collections are objects whose data is considered a part of the parent and ancestor objects within the module data structure. Therefore the synchronization process for a collection includes any nested collections within it.

Object Properties Concepts and Considerations

An object property definition defines a single piece of data and its type. A property can also define minimum and maximum values, a default, or "special value" and other data-related behaviors. The specific data-related behaviors will vary depending on the data type of the property. The properties of an object give that object its characteristics. A property is the equivalent to a variable in other development platforms or languages.

When designing and developing an object's properties, the properties should represent all of the values to be retrieved from the back end system plus those that may be necessary for client-side processing. Examples of this latter group include state-related values or other data that will not be retrieved from or updated to the back end, but that may be needed on the Client.

Property Data Typing
When data typing your properties, the primary driving factor in the decision should be the data type of the back end value the property is to store. However, it is not a requirement that the data type of the object property match the data type of the back end value. The Agentry Server will always attempt to convert data retrieved from the back end system to the data type of the

property definition. Therefore, if a value stored as one data type in the back end system, such as an integer, needs to be stored and used as a different data type on the Client, such as a string, it is completely valid to create a string property. Be aware, however, that when data types differ in this manner that the rules of safe data conversion still apply. For example the conversion of a string to an integer is not considered type safe and can result in undesirable behavior.

Object Key Property

When designing an object's properties you must always include a "key property" that uniquely identifies each instance of the object. Any property definition of almost any data type within the object can be designated as the key property. In practice the key property should be the value that uniquely identifies the business entity in the back end system. Examples include the work order number or customer ID, which would be values that would be defined to be the key property. The object definition contains an attribute that specifies which of its properties is the key property. Therefore the property must first be defined, and the object then edited to specify which property is the key property.

The key property is then how instances of the object will be uniquely identified by the Agentry Client and the Agentry Server. Only one object instance with a specific key property value can exist within a given collection property. Furthermore, the key property is also how the Server identifies the object instances within the collection for synchronization purposes. If a collection property is created for an object definition, that object must have a selected key property. The Agentry Editor will not allow the collection definition to be created for an object that does not have a key property.

Property Names and Data Syn chronization

The name attribute of the property should be set to the same name as it is identified by in the back end system. This plays an important part in synchronization, as the Server will look to how data values are identified in the back end system and match those values with properties of the same name. In addition to this requirement, following the back end names also makes future maintenance of the application easier.

If it is not possible to name a property to match the back end value, the identity of that value should be aliased in some manner during retrieval. For example the `AS (SELECT Field1 AS Name1)` keyword in SQL allows for this. In a Java back end the data returned can be renamed using the return data structure that stores the data for objects, as the members of that structure can be named to match the property definitions, with their values assigned to variables with different names from the back end system.

Object Data Structure Concepts

The object definition is the primary data definition for a module. All production data for the module is stored in instances of the object definitions created in the application project. The definition of the objects includes not only the data to be stored within each object type, but also the relationship between the objects within the module. These relationships are hierarchical in nature, with one object type the parent to another.

The Module Main Object

The beginning point of a module's data structure is the module's MainObject definition. This object is a part of all modules, added automatically by the Agentry Editor whenever a new module definition is created. The intended purpose of the MainObject is to contain the top-level object collection property or properties, as well as other module-level data.

The Module's "Primary" Object

The primary object of the module is the one around which most or all of the functionality within the module revolves. This includes both client-side behavior and data synchronization. It is important to note that there is no setting or attribute within the application structure that indicates an object is the primary object. Rather, this is a logical term reflective of the design of the module, its objects, and its functionality and behaviors. Examples can include a work order object for a work management module, or a customer object for a customer relations module. When a new module is defined, the Editor will prompt you to create an object definition as well as a collection property within the MainObject. The object created at this point should be the module's primary object.

Object Collections - Parent-Child Objects

Other object definitions can and likely will exist within the module. These definitions are then associated with the primary object as child objects. This is accomplished by defining collection properties. An object is comprised of properties that define the data the object stores. These properties are of various data types, one of which is collection. A collection can store multiple instances of an object definition. When one object contains a collection of other objects, those objects are said to be child objects of the first.

The reason for defining a collection within a parent object is to indicate that those child objects are data that is a part of the parent, but that are also themselves business entities in need of encapsulation. Examples of this can include the two object definitions work order and job plan step. For a work management module, the work order object is likely to be the primary object. Instances of the work order object would then be stored in a collection property of the module main object. Instances of the job plan step object would then be stored in a collection property of the work order object. The job plan step objects within a given work order object would be those representing the steps for that work order's job plan. Each work order instance then has its own collection of job plan step objects specific to that work order.

This structure can continue to several levels deep within the application structure. As an example a job plan step may require certain parts are used. The job plan step object, then, could contain a collection property of a third object type created to encapsulate a part. In practice it is usually not necessary to create an object hierarchy within a module that is more than three or four levels deep. The term **nested collection** is commonly used to refer to any collection property that is not an immediate child of the module's main object. In the preceding examples, the work orders collection would be a top-level collection, and the job plan steps collection would be a nested collection.

When working with collections it is important to keep in mind that the collection is a property of the object. This means the object instances stored within a collection are data that make up that parent object, just as any other property data type. Also, a collection property itself is not an object. When working with other definition types that affect objects and/or collections, be sure to note this distinction. An attribute, argument, or definition that is expecting an object will not accept an object collection property. Likewise an object collection property cannot be used where a single object instance is expected.

Object Data Synchronization: Fetches

The synchronization of object data is handled by fetch and push definitions, with the fetch being the primary definition for this purpose. A fetch defines how the Agentry Server synchronizes data for a target object collection. This object collection must be a top-level collection within the module. A fetch is made up of steps that retrieve the data for the collection from the back end system. These steps are grouped into three categories within the Fetch definition: Client Exchange Steps, Server Exchange Steps, and Removal Steps. A fetch may also include properties to store data captured from the user and validation rules for those property values. A fetch may also contain property and validation rule definitions, though this is a less common implementation option for fetches.

General Fetch Processing
During a transmit the Server processes all main fetches within the application. This is the primary distinction between main and non-main fetches. If a fetch is not a main fetch it will only be processed by the Server if the transmit step within the action on the client lists it as one to be processed. Regardless of whether or not a fetch is a main fetch, the behavior of a given fetch is the same when it is processed by the Agentry Server.

A part of fetch processing also includes a separate definition type, object read steps. The fetch targets an object collection within the module. The object definition can include object read steps. If the object type within the collection targeted by the fetch contains read steps, those steps are processed as the last part of the fetch processing. Note that read steps in any nested collection of the target collection of the fetch are not processed.

The order of processing the step usage definitions involved in fetch processing is:

1. Client Exchange Steps
2. Server Exchange Steps
3. Removal Steps
4. Object Read Steps

Each of these definition types references a back end step definition within the same module as the fetch. A fetch can contain multiple definitions of each of these types, with the order of execution defined within a given type defined by the developer.

The fetches of an application are the last synchronization definitions processed during transmit. For applications with multiple fetch definitions, the order in which each fetch is

processed is undefined. Therefore each fetch definition should be defined to operate independently of any others within a given application.

Object Retrieval and Replacement Processing

When a fetch is processed during transmit, any step executed by a client exchange, server exchange, or object read step can return data to create new objects or replace existing objects. This process involves a step returning data from the back end system to the Agentry Server. The values of this data are named or identified according to the logic within the step. For example, a SQL step containing a select statement will return a data set of records. Each column in the return set is identified according to either the name of the database table from which it was retrieved, or according to any column alias for the values contained in the SQL statement. The Agentry Server then processes this data according to the following procedure:

1. During the transmit the Client provides the Server with the key property and last update value of any object instances currently contained in the object collection targeted by the fetch. These values are stored in an object collection in the Server's memory that mirrors the collection on the Client. This occurs before the steps of the fetch are processed.
2. When a step within the fetch returns data, a value with the same name as the key property of the object is searched for in the return set. If such a value is not found then the data is discarded, as the Server cannot determine to which object the data belongs.
3. When the key property value is found, the Server compares this value with the key properties of any object instances it currently contains in the object collection.
    a. If a match is found, any other data within the record is processed. Each value identified with the same name as property in the object is assigned to that matching property. If the object property contained a value prior to this processing, it is overwritten. This is object replacement processing.
    b. If a match between the return set's key property and an existing object instance, a new object is instantiated by the Server and stored in the collection. The remaining values in within the record is processed, with each value identified with the same name as a property in the object assigned to that matching property. Any properties within the object that do not have a value in the return set are initialized to null. Any values in the return set that do not have a match in the object properties are discarded.
4. This process repeats for each step executed as a part of fetch processing that returns data. When the fetch step usage definitions and object read steps have all been executed, the Server sends down any new or replaced objects to be stored in the object collection on the Client.

Object Removal Processing

In addition to retrieving object data, the fetch processing can also result in the removal objects from the Client. When a removal step returns a value identified as the key property for the object type being synchronized, the Server will send this value to the Client with the indication that it should be removed from the collection property. The removal step should normally only ever return the key property value, as any other data returned is not used by the Server and therefore unnecessary.

When an object is deleted from the Client it is important to keep in mind that the objects stored in any nested collections are also removed. Remember that these child objects are data for the parent object and, just as any other property would, they are removed with the parent object.

The order of processing during a transmit is such that, if an object is to be removed as the result of fetch processing, any data captured on the client in transaction definitions targeting the fetch will have already been processed. No data captured on the client is lost as a result of a fetch removing the object, as the transaction will have already been processed and, presumably, updated any captured values to the back end system.

Object Read Step Concepts

The object read step definition is a child to the object definition type. An object read step references a step definition within the same module. Its purpose is to retrieve data for instances of the object from the back end system. The steps are processed by the Agentry Server during a transmit. The step being referenced can be executed once per transmit or iteratively.

Multiple object read steps can be run to retrieve the data for an object. During synchronization the Agentry Server will create instances of the object after the first set of return data that contains values for the object key property. Subsequent steps that return data for the object must also include the key property to indicate which object the data belongs with. These values will then be used to set the property values of the object. Any object read step can return any property value for the parent object type. A single step can return all of the data or multiple steps can be run to retrieve all of the data. This processing is defined by the developer and the nature of back end system and how data can be retrieved will dictate the proper way to retrieve the data.

Object Read Steps and Back End Steps

The two key items to keep in mind when defining an object read step are that, first, the step definition being executed is separate from the object read step. The step being executed is a module level definition containing the processing logic desired. The step definition must be defined first, and then the object read step can be defined to run it. Depending on the step type this can be a SQL statement, Java logic, or HTTP-XML calls. The object read step references the step definition and specifies when and why it should be run. It does not define the actual processing or the back end system to use. This separation of the logic and the context is an intentional part of the overall architecture that allows for multiple steps defined for different back end systems to be used an executed to synchronize data for a single object type.

The step executed by an object read step has access to certain data about the object. The specific values that are in scope depends on how the object read step has been defined to be executed. To access the in-scope values the Syclo Data Markup Language (SDML) is used.

Any step executed as an object read step must be defined to return not only the data for the object properties in need of values, but must also return the value of the object key property. The key property is used by the Agentry Server to determine which object instance should be assigned to values in the return set. If a read step returns a key property that does not match an

existing object a new object is instantiated and the other values in the return set are assigned to the new instance's properties.

The purpose of an object read step is to return data for properties of the object. This includes object collection properties. While a collection contains object instances, from the context of the parent object the collection is simply another property containing data that is a part of the overall object instance. Therefore an object read step can return data to create instances of the object type stored in the child collection property. An object read step must be defined to read the data into that child collection. The data returned by the step must include the key property of the parent object and the key property of the object instances to be added or updated within the collection property.

As an example of reading data into a child collection property, consider a work order object that contains a collection of job plan steps. An object read step can be defined in the work order object that retrieves data for object instances within the job plan steps collection. The step executed to retrieve this data must return the work order objects key property and the job plan object's key property. This data is needed by the Server to determine, first, which work order object contains the job plan step object and, second, which job plan object instance the data should be assigned to. Just as with any object read step, if a job plan object key property value is returned that does not match an existing object instance, a new job plan step object is instantiated and added to the collection property of the work order object.

Object Read Steps and Fetch Processing

Object read steps are run as a part of the synchronization process for objects and object collection properties. During Client-Server transmission, fetches are processed as the first part of object data synchronization. If the object type that makes up the collection targeted by the fetch contains read steps, those steps are processed after the fetch. In a common application architecture, the fetch will synchronize the exchange data and the object read steps will use that data to determine which object instances to retrieve from the back end system.

When object read steps are run as a part of fetch processing the steps executed are, in most cases, defined to retrieve data for all instances of the object in a single execution. The object read step is defined to run one time in this case, rather than to iterate over the object collection being synchronized. Note that this is the most common way object synchronization is defined, but is not a requirement. There are situations in which a portion of the object synchronization must be performed one object instance at a time, or iteratively. A ready example of such a situation is when file transfer, or "attached documents" functionality is being implemented.

Steps run as object read steps during fetch processing that are also defined to iterate over the object collection have access to the key property of each object instance in the collection and the last update value for each object. These values are sent by the Client to the Server at the beginning of the fetch processing during transmission.

Object Read Steps and Transaction Processing

Read steps may also be run after a transaction has been processed that targets an instance of the object. This processing only occurs when a server data state step or server update step within

the transaction has been defined to replace the client object after transaction processing. In this situation the object read step is run for the object instance targeted by the transaction. Therefore the step executed should be defined to retrieve data for a single object. Read steps intended to retrieve data for object collection properties can still be defined to retrieve all data for objects within the collection.

When an object read step is run as a part of transaction processing, with the intent of replacing the object on the Client, the in-scope values for the step include the object's key property, the object's last update value.

Object Read Step Execution

The execution of a read step is controlled by the attribute Run. The Run attribute specifies how often to run the step in relation to the object instances currently being processed and in scope. This execution can be either once or iteratively. Running the step once means the step being processed is expected to return all of the needed data for all of the objects in a single execution. The Server is capable of processing such return sets to create or update multiple object instances.

For iterative processing there are two options. First, the step can run one time for each instance of the parent object currently in scope. For example, when processing a collection of work order objects, a read step within the work order object can be defined to run once for each work order object instance the Server currently contains.

The second option for iterative processing is to execute the step once for each object instance in a collection property of the parent object. This iteration then includes the parent object as well as the objects in the collection property. So if the work order object contains a collection property of job plan step objects, an object read step can be run once for each object within the job plan steps collection of each work order object.

In the case of iterative processing of the object read steps it is assumed a previous read step or one of the fetch steps has returned the data needed to create the objects. Read steps defined to be executed iteratively then run once for each of these object instances to continue the data synchronization process. If an object instance does not exist when the iterative read step is to be processed it will not be executed by the Server as there are no objects to iterate over.

Object Read Step Development Considerations

When designing and developing object read steps, the following items will factor into how the steps being executed should be defined as well as the object read steps themselves.

- The overall context of the read step execution, i.e. is it being executed for fetch, push, or transaction processing? Will the object read steps as a whole be executed for more than one of these?
- The data to be retrieved by the step and where it is intended to be used. Within the parent object of the read step, in a collection property of the object, in a descendent collection property?

- The overall requirements of the object data retrieval process, as dictated by the back end system. The order in which object read steps are executed is always an important consideration.
- The type of data or objects being retrieved by the step. File transfer functionality, for example, will have different effects on the design and development of the step than the retrieval of some other data types.
- The overall requirements of the back end system and system with which data is being synchronized.

One of the main aspects of the object read step definition to keep in mind during the design and development of an application is that the object read step is always run as a part of the processing of other synchronization definition types. There is no point in the synchronization process in which the object read steps are run by themselves. They are, rather, executed after the processing of a fetch, push, or transaction. Because of this overriding aspect of the read step definition, detailed discussions of the development of a read step are deferred to the discussions of the overall processes for these other definition types.

Here information is limited to the data available to the steps being executed as object read steps under various circumstances.

Read Step In-Scope Values: Fetch and Push Processing
The data values available to a read step run as a part of fetch or push processing is dependent in large part on how the step is being executed, that is, the setting of its Run attribute. Therefore the following table lists the data available to the read step organized according to the different settings of this attribute:

Run Attribute Setting	Available Data Values
Run One Time	Any SDML local data tags created by the fetch or previously executed read steps.
Run Once Per Object	Any SDML local data tags created by the fetch or previously executed read steps.
	The key property of the current object instance.
	The last update value of the current object instance.
Run Once Per Collection Object	Any SDML local data tags created by the fetch or previously executed read steps.
	The key property of the current object instance.
	The key property of the current collection object instance.
	The last update value of the current collection object instance.

Read Step In-Scope Values: Transaction Processing
The data values available to a read step run as a part of transaction object replacement is dependent in large part on how the step is being executed, that is, the setting of its Run

attribute. Therefore the following table lists the data available to the read step organized according to the different settings of this attribute:

Run Attribute Setting	Available Data Values
Run One Time	Any SDML local data tags created by previously executed read steps.
	The key property of the object instance being replaced.
	The last update value of the object being replaced.
Run Once Per Object	Any SDML local data tags created by the fetch or previously executed read steps.
	The key property of the object instance being replaced.
	The last update value of the object instance being replaced.
Run Once Per Collection Object	Any SDML local data tags created by the fetch or previously executed read steps.
	The key property of the object instance being replaced.
	The key property of the current collection object instance.
	The last update value of the current collection object instance.

Fetch Development Using the Exchange Data Model

The design and development of fetch to synchronize an object collection should always be based on the exchange data model. The child definitions of a fetch, as well as object read steps and the object itself, are organized and architected with the intent of using this model. The main tasks to accomplish when developing a fetch to use the exchange data model are:

1. Create or configure the exchange components in the back end system to be used both during the synchronization process, as well as to track changes to the back end system between client transmits. These components should track the unique identifier values for each business entity, the date and time of the change, and the nature of the change. This last includes tracing the addition of new instances, modification of existing data, or the removal or other modification to be treated as a removal by the Agentry application.

2. Define the fetch to determine and record in the exchange components what objects the Client contains at the beginning of the synchronization process. This includes the date and time when each object was last retrieved from the back end system.

3. Define the fetch to use the exchange and tracking components in the back end system to determine what object-related data has been added or modified since the last time the Client synchronized.

4. Define the fetch to use the exchange data generated in the previous steps to determine the differences between the Client objects and the back end data. Retrieve only those objects that need to be added or replaced, and define the fetch removal steps to retrieve the key property values of those objects to be deleted from the Client.

Back End Exchange Data and Tracking Components

The design and creation or configuration of the back end components used in the exchange data model have certain general requirements. These requirements hold true regardless of the contents of the object data being synchronized.

Beginning with the tracking components, the purpose of these items is to track changes of interest to the object data in the back end system that occur between transmits from the Client. These components should track changes to the object data in the back end system. Depending on the nature of the back end system, changes to track can include adding new objects, modifying the data of an existing object to be reflected on the Client, or the deletion or modification of an object that should result in that object being removed from the Client.

Note: Regarding the removal of objects, this may occur within the Agentry application as the result of various types of changes to the data beyond the removal of the object from the back end system. Other changes to the object can dictate the end user should no longer have the object on the Client. Examples include the reassignment of a work order, a change in the objects status, such as inactive or deprecated, or similar modifications. This should be kept in mind when implementing both the back end components as well as when defining the removal steps of a fetch.

When one of these changes occurs, the information recorded must include:

- The value or values that uniquely identify the modified object in both the back end system and the Agentry application.
- The date and time, as provided by the back end system, when the change occurred.
- The nature of the change, that is, is it a new object, a modification, or a removal.

The Object Last Update Value

The object definition type is capable of storing a date and time value called **lastUpdate**. This data value is separate and in addition to the defined properties of the object. The date and time value it stores must be returned with the object data during synchronization and aliased as "lastUpdate." The proper source for this value is the current date and time of the back end system when the object data is retrieved.

The purpose of the lastUpdate value is to store the date and time of the back end system when the object was retrieved. Each object instance has its own lastUpdate value. This value is stored with the object instance on the Client. It is accessible during subsequent synchronizations via the SDML tag <<lastUpdate>> and is intended to be used to determine whether changes have occurred on the back end system for the object since it was retrieved for the Client. It should be compared with the date and time value recorded by the back end tracking components for the same object.

This value is only accessible during synchronization and, while stored with the object instance on the Client, it is not exposed on the Client. It is not a property value and cannot be displayed to the user nor modified as a result of any client-side processing.

Client Exchange Steps in the Exchange Data Model

A fetch client exchange step defines how information about the target collection is processed by the Agentry Server. This definition references a step definition within the same module. This step has access to information about the target collection, as well as to any data captured in fetch properties. A client exchange step can be defined to execute once or iteratively, and can return data for an object collection. A fetch can contain multiple client exchange step definitions, which are processed by the Server in a defined order.

When a client exchange step is executed iteratively, its iterations are based on the object instances sent to the Server by the Client. The client exchange step then has access to the object's key property value and the object's lastUpdate value. If fetch properties have been defined, the client exchange steps of that fetch will also have access to all of these property values, regardless of how the client exchange step has been defined to execute.

The intended purpose of client exchange steps is to update the back end exchange data components with information about what object instances currently exist on the Client in the target collection of the fetch. This information should include the key property of each object and the lastUpdate value. Along with these values the client exchange step should also provide information about the user to whom the object belongs (SDML tag: <<user.agentryID>>). Finally it is a recommended practice that the Agentry Server instance also be uniquely identified (SDML tag: <<server.serialNumber>>).

Another task commonly handled by the client exchange steps is to clear out the exchange data for the current user from the previous synchronization. The step defined to accomplish this task should delete this data based on the user ID and, in most cases, also the Server's serial number. This client exchange step is run as the first step for the fetch. It is then followed by the client exchange step defined to provide the exchange data about the current objects on the Client.

When the client exchange steps of a fetch have completed processing during synchronization, the back end exchange data components should contain the information about what object instances currently exist on the Client and the date and time each was last retrieved from the back end system.

Server Exchange Steps in the Exchange Data Model

A fetch server exchange step defines how information about the back end system's data is processed. This definition references a step definition within the same module. This step has access to information about the target collection, as well as to any data captured in fetch properties. A server exchange step can be defined to execute once or iteratively, and can return data for an object collection.

Server exchange steps are always processed after client exchange steps. The intended purpose of a server exchange step is to determine what changes have occurred in the back end system to the object data for the user and to then either mark the objects in the exchange data components as those in need of retrieval, or to perform the actual retrieval of the object data. Which

behavior is defined is dependent on the type of back end system and its capabilities and behaviors, as well as the overall needs of the mobile application being developed.

Changes or differences between the Client objects and the back end data are found by comparing the information about the current Client-side object instances provided by the back end exchange data component with the information captured in the back end tracking components. The tracking components will contain the date and time when any object data has changed on the back end system, along with the unique identifier for that object. The exchange component will contain the date and time when the object instances were retrieved. The server exchange steps then should contain the logic to compare the information in the tracking components with the information in the exchange components. When an object is found to have been changed in the back end system more recently then it was downloaded to the Client, that object is one in need of replacement on the Client. Any new objects created in the back end system will not have a corresponding item in the exchange component. Such objects should also be treated as those in need of retrieval for the Client.

When differences are found, the functionality can take one of two directions. First, any changes for existing objects result in the object record in the exchange component being flagged or marked in some manner indicating the object should be replaced. Any new objects in the back end system not found in the exchange data components are added to the exchange component. This information includes the unique identifier, or key property of the object, the user ID, the server ID, the current date and time, and finally the same flag or indicator that the object should be retrieved. In this scenario, the actual object data is not yet retrieved. This logic is most common with SQL systems. In this scenario the data retrieval is then left to either additional server exchange steps, or, more commonly is handled by object read steps.

The second option is define the steps used as server exchange steps to make the determination about which objects need to be retrieved and to retrieve that data at the same time. In this situation, the server exchange steps do not update the exchange component with the information about the new changes in the back end. Instead, the data in the back end exchange component as provided by the client exchange steps is used only for comparison purposes with the tracking components. This logic is most common with Java and Web Service (HTTP-XML) systems.

Fetch Removal Steps in the Exchange Data Model
A fetch removal step is defined to determine which objects should be removed from the collection targeted by the parent fetch. A removal step references a step definition within the same module. This step has access to information about the target collection, as well as to any data captured in fetch properties. The step referenced by a removal step definition is expected to return the key property of any object(s) that should be deleted from the target collection on the Agentry Client.

The removal steps defined within the exchange data model should use the exchange data components to determine what objects the client has. It should interrogate the corresponding data in the back end system to determine if any of those objects should be removed from the Client. The removal of a Client object can occur in many situations, only one of which is the

actual deletion of the dat in the back end system. In point of fact, this is usually the least likely situation, as most enterprise systems do not remove data once it has been added.

The removal steps can use the data in the back end exchange data component to look for object data for a user in need of removal. As an example, if a user currently has work order 123 on his Client and that same work order has been reassigned in the back end system to a different user, the removal step can return the key property of that work order object so that it is removed.

Object Read Steps in the Exchange Data Model

An object read step references a step definition within the same module. Its purpose is to retrieve data for instances of the object from the back end system. The steps are processed by the Agentry Server during a transmit. The step being referenced can be executed once per transmit or iteratively.

Object read steps may be run at different times during a transmit. One of these times is as a part of fetch processing. Object read steps are not required to retrieve data during fetch processing, as both client exchange and server exchange steps can accomplish this task. However, it is common to use object read steps for this purpose. In the exchange data model, the fetch is processed to determine what changes have been made to the back end data and the object read steps are then run to perform the actual retrieval. One of the primary reasons for this division is simple organization of the project.

When used in the exchange data model the steps executed as object read steps should contain logic to use the exchange data generated by the fetch processing. The object read steps should retrieve only that object data related to objects in need of retrieval or replacement on the Client. The processing of the object read steps and the logic executed should be the final culmination of all exchange data model processing performed to this point. The read steps take advantage of the information generated and gathered by the child definitions for the fetch, retrieving only the objects needed by the Client and excluding the retrieval of any unchanged objects.

Agentry User Interface Definitions Overview

The primary purpose of the architecture of the client-side user interface definitions within Agentry is to support multiple client device platforms from a single application. This architecture, then, includes the separation of the application's business logic from the user interface. This separation then requires the specific structure of the user interface definitions within the Agentry application project.

The primary interface definition is the Screen Set. The screen set is a module-level definition. A screen set definition defines the Agentry Client's user interface. The screen set defines the definition type to be displayed, which can be an object, transaction, or fetch within the same module. The properties of this definition type can then be displayed by the screen definitions within the screen set. Screen sets contain the child definitions screen and platform. The screen set is the definition referenced by other definitions for display and are universal to all supported platforms within the application project.

The platform definition is one of two child definitions to the screen set. A platform definition defines how a screen set's screens will appear on a specific device type. A platform is defined to use one or more screens within the same parent screen set. There are different platform types, each corresponding to a different type of client device. The platform affects the placement of buttons and the form factor of the screens it uses.

The screen definition is the second child definition to the screen set. A screen definition defines how the property values in the definition being displayed are presented to the user on the Agentry Client. This includes which values are displayed. A screen also defines, via its child control definitions, how a user can interact with the Client. There are two types of screen definitions: list screens, to display object collections; and detail screens, to display a single instance of an object, transaction, or fetch definition.

Screen Sets, Platforms, and Screens at Run Time
The overall structure of the screen set definition is intended to support multiple client device platforms from a single application project. Within the screen set there exists one or more platform definitions. Each platform definition matches a client device type within the implementation environment. A given platform is defined to use one or more screens within the same screen set. Depending on the types of client devices, there may or may not be overlap in the list of used screens among the platforms; i.e., it is possible for the same screen to be used by more than one platform. Alternately, a given screen or screens may be used by only one platform within the screen set.

When new or edited screen sets are published to the server, they are then downloaded to the clients at run time. When this occurs, the client provides the server with information about the client device upon which it is running. The server then interrogates the screen set definition, looking for the platform definition within it for that client device. When found, the screens used by that platform are then those downloaded by the client for that screen set. No other screens within the screen set are received by that client. A second client running on a different device type will receive the same screen set, but with different screens. These screens would be the ones used by the platform definition for this second device type.

The screen set, then, is the definition referenced by other definitions for display. Typically this is an action step, such as a navigation or transaction step. Actions and their steps are platform independent, meaning all clients receive the same action definitions regardless of the client device type. The action steps then dictate the screen set to be displayed. When a screen set is displayed on the client, the screens it contains are always the ones matching the device type, as the screen set on a given client will contain only those screens for that client device.

How the screens of a screen set are displayed depends on the definition type the screen set is defined to display. When a screen set displays an object, its screens are displayed in a tab control. Each screen definition is represented by a tab. Selecting a tab displays the corresponding screen.

When the screen set is displaying a transaction or fetch, the screens are displayed in a wizard format. This results in each screen being displayed one at a time, with wizard buttons (back,

next, cancel, finish, etc.) displayed at the bottom. The user navigates through the wizard using these buttons, entering data in each screens fields.

Screen Types: List Screen
There are two types of screen definitions that can be added to a screen set: list screens and detail screens. A list screen definition displays an object collection property on the Agentry Client. Object instances from the collection are displayed as rows in the list. A list screen contains the child definitions column and button. A column is defined to display the property value for each object instance in the collection. Buttons are defined to execute actions related to the object instances. List screens include definable behaviors related to filtering, scanning, and sorting, as well as other screen enhancements for displaying data stored in the object instances of the target collection property.

The list screen definition is typically used for a basic presentation of an object collection property. Each object instance is displayed in a list control, which is the main feature of the list screen. This screen type can only be used to display object collection properties. The columns of a list screen are defined for the properties of the object type in the collection being displayed in the list control.

The list screen can be defined to allow or prevent users from resorting the list of objects by clicking a column header within the list control. The default is to allow resorting. Disabling this functionality will prevent the user from resorting the list of objects. This is often used when the objects represent some prescribed order, such as safety plan procedures. In this case it is generally considered good form to select a fixed sort property from the object collection being displayed. This is also defined in the list screen.

The columns themselves are defined to either be included or excluded from those values upon which the list can be filtered. The default for a column is to be included. If it is desired to prevent the user from filtering the list screen on certain object values, the columns for those values can be defined to be excluded from the filter values. As a separate attribute in the column is wether or not the column values should be included in scan filtering. Scan filtering is the behavior where a user can scan a barcode value and the currently displayed list is then automatically filtered to only those items that match the scanned value. The column definition includes the Scanner Filter attribute that specifies whether or not the column value should be used to filter the list based on a scanned value. This behavior only applies when the parent list screen is used by a platform for scanning, and only when the client device is equipped with a barcode scanner.

List columns can also display the values for each object as a hyperlink. When this feature is enabled, each cell in the column is displayed as a hyperlink. When the hyperlink is selected an action is executed. Part of the hyperlink functionality is to define the action to execute and the object instance to be targeted.

In addition, each list column definition can be enabled or disabled based on a rule, with disabled columns hidden from the user. Columns can also be formatted using the Format attribute. Finally, the default width of the column can be specified. Related to this is a behavior

defined in the parent list screen. It is possible to prevent users from resizing the columns by defining the list screen to disable this featured.

Screen Types: Detail Screens
A detail screen definition displays a single instance of an object, transaction, or fetch on the Agentry Client. The properties of the definition instance are displayed in fields, a child definition to the detail screen. Definable behaviors of a detail screen are predominantly controlled by the screen's child field and button definitions, which can include read-only or read-write values within the fields, as well as numerous field type behaviors. Detail screens for transactions and fetches do not have the child definition button.

The overall behavior of a detail screen is dependent in large part on the fields it contains. A detail screen field defines a field control for display on the parent screen. The field displays data to the user and, when displaying a transaction or a fetch, can capture data from the user. A field can be defined to have one of several edit types that will affect both the appearance and behavior of the field on the screen, especially when capturing data.

The field edit types vary from basic string fields to more robust fields including several different list types, a calendar control, date and time pickers, and several others. The proper field edit type depends on, first, the data type of the value the field is displaying, and, second, the desired method in which users should enter data. When fields are displayed on detail screens displaying object instances, data entry is typically not a part of the design as the fields are read-only when displaying object properties. In this case, the field's edit can be changed, but typically it is left set to default. When the field's edit type is set to default, the field's edit type at run time matches the data type of the property being displayed.

For detail screens displaying transactions or fetches, also known as wizard screens, the field edit type should always be considered carefully as these fields will be used to capture data from the users. The method of data entry provided to the user can have a significant impact on the applications usability, as well as the accuracy and validity of the data captured.

Fields for a detail screen can be hidden or displayed based on conditions checked by rules. Likewise fields can be enabled or disabled conditionally. the label for fields can be a simply text value, a hyperlink that executes a defined action, or can be committed entirely, leaving just the field itself with no label. Fields can be defined as read-only. This attribute affects fields on wizard screens (for transactions or fetches) or on fields for object screens when those fields do not target any object property.

The field includes several attributes related to its positioning on the screen, the viewable size of the field, and the amount of space within this size dedicated to the field's label. Fields can also have a shortcut key associated with them, which will set the focus to the field when selected.

Detail screens are defined with a certain number of columns and rows. These are for the purpose of layout. A given fields position on the screen is defined by specifying the column and row in which the field's upper-left corner should reside. Likewise, the fields width and height are also defined in terms of the number of columns wide and number of rows high. A

detail screen is created with a default number of rows and columns which can be edited by the developer. Note that changing the number of columns or rows for a detail screen does not change the size of the screen. Rather, it results in a larger or smaller number of "pieces" to that detail screen for the purposes of field layout.

When changing the number of columns and rows for a detail screen, it is recommended this be done before fields are added to the screen. If these values are changed after fields have been added, the layout of those fields will be affected. If fields are positioned on rows 1-10, and the detail screen is then edited to contain only 8 rows, the fields on rows 9 and 10 will no longer be displayed and must be repositioned some where in the first eight rows.

Button Definition for All Screens

The button definition is common to both list and detail screens. A screen button defines a button control to be displayed on a Client screen. The button may be displayed as a standard button control, a tool bar button, a menu or menu item, or as a separator. A button is defined to execute an action when clicked or tapped, unless defined as a menu or separator. When executing an action the button also defines the target object instance provided to the action for processing.

When developing mobile applications screen space is at a premium for many of the client device types in common use. For this reason, the button definition has multiple types. The type action button creates a traditional button control that when clicked executes a defined action. The exception to this is when the selected "action" is Popup Menu, which is one of the available options in the Action attribute of a button. When this is selected, the button will not execute an action. Rather, it is displayed with the defined label, plus an arrow pointing up. When selected on the client, a popup menu is displayed. Additional action buttons on the same screen can be defined to be displayed on this popup menu.

Application Menu is a button type that adds a menu item to the client's menu bar, in the menu Actions. This menu is hidden unless at least one Application Menu button has been defined for the current screen. This type of button creates a menu item within this menu that, when selected, executes the defined action. This button type does not support image icons or the style attributes.

Toolbar Button is a button type normally used on Pocket PC devices, or devices with this form factor. It creates a button with no label and only an icon. The button resides below the screen in the toolbar of the client. When clicked it executes the defined action.

Separator is a button type that does not create any button or menu control. Rather, it is used to help organize buttons on the screen. When a separator is defined and the button is not defined to be displayed in a popup menu, additional space is placed at the position of the Separator button definition. When the separator button is placed in a popup menu, a menu separator is drawn at the position of the Separator button.

Buttons for List Screens

The button definition for list screens is defined to target, by default, the currently selected object or objects in the screen's list control. Depending on the action being executed this may

or may not be the proper selection. The type of object targeted by the object must match the object type for which the action has been defined. The exception to this is when the action is not defined for any object type (attribute **For Object:** -- None --).

Typically either the selected object or the parent to the collection being displayed by the list screen are the two items selected for the button target. Buttons that execute actions to navigate to another screen set, and that execute actions to instantiate an Edit or delete transaction are normally defined to target the currently selected object. Buttons that execute actions to instantiate Add transactions, or actions such as Transmit or CloseThisScreenSet should be defined to target the parent object of the collection being display in the list screen.

Buttons for Detail Screens

The button definition for detail screens is defined to target, by default, the object currently being displayed in the detail screen. In traditional development work within Agentry this was not often changed. However, in more contemporary applications developed using later versions of Agentry, the selection of a different button target has increased in frequency. this is due to many of the newer field edit types added to Agentry. Many of these fields display lists of objects, or a selection from a list. These fields then support the selection of a target for a button from that field's current selection. This can be selected using the target browser within the Agentry Editor.

Client User Interface Considerations and Guidance

When developing the user interface for a mobile application in Agentry, the first consideration should be given to the screen flow, i.e. how the user should navigate through the information presented in the screens. In general it is a good starting point to look to the data within the module. In any real-world application there is likely to be multiple collection properties with a structure or hierarchy of their own. For example, Customers may contain Orders, which in turn contain Products. Likewise, Work Orders may contain Job Plan Steps.

When designing the screen flow, then, it can be useful to start with a basic drill-down approach. First, present the user with a list of the top level collection in the module. Then, allow them to select a object in this list to view details about that object. These details can include the property values of the selected object displayed in fields, as well as the collection properties it may contain, displayed in their own lists. If further nesting of object collections exist, this can be repeated for level of data within the module's data structure.

Note that this is a beginning point within the design. This structure need not be a part of the final implementation, and in fact may never be implemented exactly in this manner at all during development. However, it can be used as the foundation for the final UI design and implementation.

Once the basic drill down structure has been designed it should be further refined to match the needs of the application, and to reduce the amount of interaction required by the user, that is, to reduce the number of clicks required to get to the information or functionality needed.

Next, the portion of the user interface for transactions should be considered. If users can add instances of an object, consider the best point or points within the UI flow to expose this ability. Similarly, edits to the objects, and also deletes, should be exposed at points where it makes sense to the users. Again, when making these determinations, begin with the basics. If allowing users to add a given object type, expose this action on the screen where this collection is listed. Also, deleting objects is functionality typically exposed in the list for the collection.

Edits can be expose in lists as well, but may also be exposed in detail screens for the object. In many applications there are numerous edit transactions for the same object, with the different edits affecting different property values within the object definition. If these values are displayed in detail screens for the object grouped together in a manner similar to how they are organized in the different transactions, it makes sense to expose those transactions in those detail screens where the values the transaction affects are displayed.

Security Related Development Overview

When developing a mobile application security is always an important aspect to the process. Using the Agentry archetype many of the security features are implemented for the application as a part of the development of the Agentry application project. Information is provided here on the security features and development options available and how they are implemented in the application project using the Agentry Editor.

Client-Side Data Encryption
Any Agentry Client can support the encryption of all data stored locally on the client device. When implemented, production data retrieved from the back end system, as well as the application data (or business logic) of the application is stored encrypted. Subsequent information is provided on how to implement this functionality for your mobile application. This may be defined within the application project while it is being initially developed, or it may be a change made to an existing application.

Securing File Attachments From iTunes on Agentry Client for iOS
Depending on where file attachments are stored on an iOS client device, they may be accessible through iTunes when the client device is connected to that application. Information is provided on how to modify or define the External Data properties of the Agentry application project so that files stored on the client device are not accessible to iTunes.

User Lockout After Failed Login
A standard part of any IT department's security policies is a specification on the maximum number of failed login attempts can be made by a user before restricting their access to the system in some way. This behavior is supported in Agentry via the use of security settings within the Application definition of the Agentry application project. Included in this functionality is the ability to define the maximum number of login attempts allowed by the user, and the corresponding lockout action to take when this maximum is met. As a part of the

definable behaviors it is possible to require the user to perform a full transmit before being allowed to access the Agentry Client, as well as optionally removing some or all of the data stored on the client device by the Agentry Client.

Transaction Authentication

As a part of the workflow of the client application it is possible to require the user to re-enter their user credentials before a transaction is applied. When implemented the user will be required to enter their user ID and password, which is validated against the locally stored credentials for the user, before the transaction is applied and saved on the client. Additional information may be captured from the user as a part of this process. This data is both stored locally and is also available for update to the back end system as a part of the transaction processing during transmit.

Defining Client-Side Data Encryption

This procedure describes the process of defining the Application definition within the Agentry application project such that Agentry Clients will encrypt all data stored on the client device.

1. Open the Agentry application project for you mobile application in the Agentry Editor.
2. View the Application definition and select the Application Security tab in the Properties view.
3. Set the attribute **Client Database will be encrypted** to true. Save the change.
4. Publish or deploy the project to the Agentry Server.

Data stored on the Agentry Clients will be encrypted.

Securing Attachments on iOS Client Devices

Prerequisites

It is assumed that attached documents functionality has been defined for the mobile application. This procedure does not describe how to define or implement this functionality, but only how to define the mobile application for iOS client devices to prevent attached documents from being accessible via iTunes.

Task

This procedure describes how to define the mobile application to secure attached documents from iTunes access on iOS client devices. This process involves setting the iOS Base Path attribute of external data properties within the application to a value other than Documents, typically the Application Support option.

This procedure needs to be repeated for each external data property within the mobile application project.

1. Using the Agentry Editor navigate to the external data property definition within the mobile application project. Select the File Locations tab within the Properties view of the definition.

2. Within the iOS section of attributes on this tab, set the Base Path attribute to the option "Application Support."

3. Save the changes made and repeat this process for any other external data properties within the application project.

4. After all external data properties have been modified, publish the application to the Agentry Server and test the behavior. Be sure to connect the iOS device to an iTunes application and verify the attachments for the mobile application are no longer accessible through iTunes.

5. When ready to make this behavior available to the mobile users, publish or deploy the application to the Agentry Server in the production environment. This new behavior will take affect when mobile users perform their next synchronization.

After this procedure is complete the files stored on the iOS device by the Agentry Client will no longer be accessible to the iTunes application.

Configuring User Lockout for Failed Login Attempts

Prerequisites

The following items must be addressed prior to performing this procedure:

- Determine the desired number of maximum login attempts before locking out a user.
- Determine the proper response by the client when locking out a user. Review the information provided in the *Agentry Language Reference*, specifically the section "Application Definition," in the subsection "Application Security Attributes."

Task

This procedure describes how to configure the user lockout behavior on the mobile application, which occurs after the defined number of failed login attempts by the mobile user. A part of this configuration is the resulting behavior, as set by the "lockout level", when a user is to be locked out.

These settings should be configured to match the security requirements of the implementation environment. Possible lockout behaviors range from simply requiring the user to perform a successful login and full transmit with the Agentry Server before being allowed to proceed; removing all module-level production data (including object instances and pending transactions) and requiring a full login and transmit; or completely resetting the Agentry Client executable, removing all data stored by the application, and requiring a full transmit and synchronization before being allowed access to the application.

Defining this behavior requires the modification of the Application Security attributes found in the Application definition, followed by publishing the changes to the Agentry Server, with a

subsequent transmit by each Agentry Client to update the mobile application with the new settings.

1. Open the Agentry application project in the Agentry Editor. View the Application definition and select the Application Security tab in the Properties view.

2. Begin by setting the maximum number of login attempts to allow by setting the attribute Login Attempts to the desired value.

3. Select the desired lockout behavior by selecting the appropriate option for the Lockout Level attribute.

   For details on the Lockout Level options, see the "Application Definition" section in the "Agentry Language Reference." Review the information for the Lockout Level attribute found in the "Application Security Attributes" subsection.

4. Save the changes made to the Application definition. Publish the application to the Agentry Server used for testing and verify the desired behavior. Publish or deploy the application the Agentry Server in the production environment when you are ready for the mobile users to receive these changes.

The desired lockout behavior for mobile users reaching the maximum number of failed login attempts has been defined. The behavior will be exhibited on the Agentry Client for mobile users in the production environment once published or deployed to that environment.

Transaction Authentication/Electronic Signature Support

The purpose of transaction authentication is to validate that an authorized user is the one that entered the information captured by the transaction being authenticated. This functionality is also implemented to support electronic signatures in environments where audit trails are a requirement.

Transaction authentication is defined within the transaction itself and can be set as always required or conditionally required based on the Boolean return value of a rule definition. During transaction authentication on the Client the user is required to enter the user ID and password with which they logged into the device. Additional information may also be captured as a part of the authentication process where needed.

Any transaction defined within the application project can also be defined to include transaction authentication. Data captured during the authentication process is accessible during the synchronization of the transaction during transmit. There are different definitions involved in the transaction authentication processing, including:

- Object definition to store authentication data on the Client
- Screen set definition to display and capture data during the authentication processing on the Client
- Step definition to process the authentication information during transmit and data synchronization
- Rule definition (optional) to determine when the user should be required to authenticate

Authentication Object
The object definition displayed in the screen set during transaction authentication, termed the "authentication object," should contain properties for each of the pieces of information to be captured from users during the authentication processing on the Client. This typically includes both the user ID and password values. It can also include additional information from the users as may be required for the specific environment. This data is accessible tot he step definitions of the transaction during transmit.

Authentication Screen Set
The screen set definition displaying the authentication object during transaction authentication, termed the "authentication screen set," should be defined to display the object definition. Unlike other object screen sets, however, when displaying the authentication object the screen set is displayed as a wizard screen set. It should contain only detail screens and the fields of those screens are defined to captured the desired authentication information from the user.

Step
A step definition can be defined to specifically process the authentication data, or this processing can be included in a step definition that processes the data of the transaction. Either format is acceptable and depends on the overall nature of the synchronization processing performed for the transaction. The step can access the values of the authentication object using data tags within the SDML. The following syntax is the manner in which these values are accessed:

```
<<transaction.authenticationObject.propertyName>>
```

The value `authenticationObject` must be replaced with the name of the object definition being used. `propertyName` is replaced with the name of the property definition to be accessed. In a JVM system connection Java steplet, the values are accessed using the "getter" methods provided in the `TransactionSession` class. The property names are passed in as:

```
authenticationObject.propertyName
```

As with the SDML tags, the object definition name and property definition name are substituted in the above syntax.

Rules (Conditional Authentication)
As a part of the definition of the transaction authentication processing it is possible to define a rule definition to be evaluated prior to presenting the authentication screen set. This rule is evaluated in a Boolean context. A true return will result in the user being required to authenticate; a false return will not require authentication.

Transaction Authentication Behavior

The overall behavior of the transaction authentication begins on the Client. When a transaction is instantiated for which authentication has been defined, the transaction is processed as normal on the client up to the point just before it is to be applied. At this point, if the transaction is to require authentication, the authentication screen set is displayed. The user then enters the user ID, password, and any other information required. The password is validated against the password for that user to lo into the Client. If this validation fails, the user is presented with an error message and the authentication screen set is then displayed again. Once the authentication is successful, the transaction is applied on the Client.

During the next transmit, the pending transaction is sent to the Server and includes the information captured in the authentication object. The transaction's server data state and server update steps have access to all properties within the authentication object. The specifics of how these values are processed depends entirely on the requirements of the back end system. The step definitions that process the authentication information can be defined to perform whatever processing is required and supported by the back end.

Defining Transaction Authentication

Prerequisites

The following items should be addressed prior to performing this procedure:

- Determine if the transaction should require authentication at all times or conditionally. If conditionally, determine the specific conditions and the values on the client involved in the determination to support the creation of the rule definition that will be needed.
- Determine the requirements for the audit trail and/or electronic signature information dictated by the back end system, including what information is needed and how it should be recorded. Note this information to support the logic needed in the back end processing for the transaction as well as in the definition of the authentication object and its properties.

Task

This procedure provides guidance and information on implementing transaction authentication. The steps here include the main process to be followed as well as guidance on variations to these standards, which may be implemented depending on need. The main process presented here is the recommended best practice for implementing this functionality.

1. Begin by defining an object, preferably named "Authentication". Add to this object the properties needed to capture the values required for the transaction(s) for which it is to be used. If capturing the password for the user, define a string property for this purpose and set that property's **Password** attribute to true. This will prevent the value of this field from being displayed on the client, and protect it from being displayed in log files and other potentially non-secure locations.

As an alternative, it is possible to capture the authentication information in the properties of the transaction. However, the use of a separate object is the recommended method as it is easier to define and better supports conditional authentication.

2. Next define a screen to set to display the Authentication object definition. Add to it the platform(s) needed for the environment. Finally, define the detail screens and fields to display the properties from the Authentication object. Do not add button definitions to this screen set, as it will be displayed as a wizard screen set and the client will display the buttons needed automatically.

3. The transaction for which the authentication behavior can now be modified. Navigate to the transaction and view the main Transaction Definition tab in the Properties View. In the Authentication section of this tab set the authentication attributes for Screen Set, Authenticate When, and Information In, as shown in the following example:
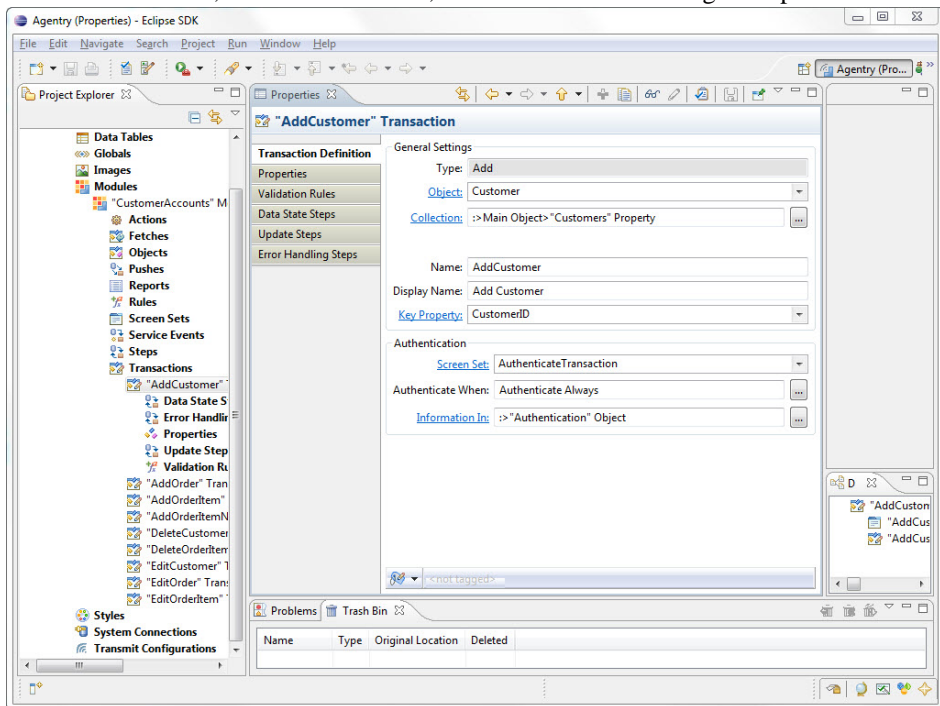
If storing authentication values in the transaction properties rather than a separate object, be sure to leave the default Information In attribute setting of "Properties of this transaction." Also, the Authenticate When attribute is where a rule can be defined and referenced. Remember the rule is evaluated in a Boolean context with a true result requiring authentication.

At this point it is possible to publish this application and test the client-side behavior of the transaction authentication. Note that authentication-related data captured at this point will not be processed to the back end system.

4. **OPTIONAL:** If condition transaction authentication is defined, a property should be added tot he transaction itself to be used as a flag indicating whether or not the authentication was performed for the transaction instance. The recommended practice is to define a Boolean property and to set the property to be initialized by a rule. In most cases the rule used to determine if the authentication should occur can also be used to initialize the Boolean property.

5. The final part of the transaction authentication is to define the logic to process the authentication data to the back end system. This is specific to the back end system's requirements for such information. When defining the step, authentication-related data is accessed using the SDML data tags `<<transaction.authenticationObjectName.propertyName>>`. For Java system connections, the values can be accessed in the `TransactionSession` class using the get methods and passing in the property names in the form `transaction.authenticationObjectName.propertyName`. In either syntax, the name of the object and the name of the property are used. The other standard transaction values, including properties and the transaction's time stamp are also accessible. Once the step is defined it can be run as a server update step within the transaction. If performing conditional authentication on the client, be sure the logic to process the authentication data checks the flag property value in the transaction to determine if the transaction was authenticated or not.

With the completion of this procedure, the transaction has been defined to require the user to authenticate themselves on the client before the transaction is applied and saved. Depending the specifics of the configuration, the authentication may be conditional.

Next

Once this procedure is complete, the behavior should be thoroughly tested in an appropriate environment. Testing should include verification of the client-side behavior, especially any and all scenarios related to conditional authentication. Back end processing should also be verified as accurate and again scenarios should factor in any conditional authentication.

Attached Documents and File Transfer: Key Concepts

Within Agentry it is possible to retrieve files during the download portion of synchronization, store those files on the client device with reference to them from the mobile application, and to send files up from the client device during the upload portion of synchronization. Files can also be attached to objects locally on the mobile application if those files are stored on the client device.

The implementation of this functionality involves several different definition types within the application project:

• A property with a data type of External Data to reference and track the file.

- Object definition to represent the document within the mobile application.
- A list to display the documents attached to a given parent object.
- One or more step definitions of a type compatible with the back end system to retrieve information about the files associated with an object and the location of those files.
- One or more File - Document Management step definitions to download and upload attached documents.
- An action step of type Windows Command to display a selected file on the client device.
- A transaction to allow the user to attach documents locally on the client and upload them to the back end system.
- A transaction to process documents that have been changed on the local client device and need to be updated to the back end system.
- One or more rule definitions to check the state of a file, including its location, size, and whether or not it has been modified since being downloaded.

For a given implementation some or all of these definitions may need to be defined. As an example, if files are to only be downloaded to the client, but never attached locally or uploaded to the back end system, then transactions and steps to process such operations are not needed.

External Data Properties and File References

When implementing the attached documents or file transfer functionality it is important to understand how the file is stored and referenced by the mobile application. An external data property is defined to reference a file stored on the client device. Files downloaded to the client device, and files attached locally on the device for later upload are not stored with the production data of the mobile application. Rather, they are stored external to the production data, with their location on the client device referenced by the external data property.

Encapsulate the File - Object Definition

As a recommended practice, the files to be downloaded, attached, and/or uploaded should be encapsulated within the application project as an object definition specifically for this purpose. While an external data property can be added to any object definition, it is a cleaner and more manageable architecture and design to create an object definition specifically for the files and to then store instances of that object definition in a collection property at run time.

All file-related operations and behaviors are far easier to implement and maintain when following this model, including downloading and uploading files, listing the files associated with a given parent object, and other operations.

Typically an object definition to represent a file is defined to include the external data property, a property to contain the name of the file (normally a string property) and a property containing the location of the file on the back end system when that location is somewhere on the file system. Normally the property containing the file name is defined to the be the key property of the object, as having more than one file with the same file name would cause issues during synchronization and storage, and designating this property as the key property will prevent such a circumstance.

Client-Side File Operations

As external files to the mobile application, files referenced by external data properties are not directly displayed in the mobile application. Rather, a list of the external data properties displayed on the client will include the name and location of that file. Similarly, detail screen fields displaying these properties include similar information about the file.

The mobile application can be defined to display an attached file in an application on the client device associated with the type of file being referenced. This application must exist on the client device and is not a part of the mobile application built in Agentry. To display a file in its native application, a Windows Command action step must be defined. It then contains as it's command the full path and file name, which is provided by the external data property. This path is passed to the operating system, which in turn "launches" the file in its associated application the device. From this point the user can perform what ever operations the application for the file allows. The only exception to this is when the file is set to be read-only, an option within the external data property. Such files are then not editable on the client device.

When attaching files locally on the client device, a file dialog is displayed to allow the user to navigate the file system and select the file to be attached. This operation is a part of a wizard screen set displaying a transaction to support this behavior. The transaction is no different than any other, containing properties to capture data from the user. An external data property is displayed in a field type specific to that property type that supports the selection of a file from the client device's file system.

Rule Functions for External Data Properties

There is a set of rule functions available for rule definitions to work specifically with external data properties:

- FILE_CHANGED: This function returns a Boolean value indicating whether or not the referenced file has been modified since it was downloaded to the client device. this can be useful when determining whether or not files downloaded to the client device are in need of update to the back end system based on changes the user may have made to those files.
- FILE_EXTENSION: This function returns a string containing just the file extension of the file referenced by the external data property. This can be useful in filtering a list of files, e.g. show just image files (`.jpg`), and similar behaviors.
- FILE_NAME: This function returns the name of the file referenced by the external data property. This is the file name only, excluding any path information.
- FILE_PATH: This function returns string containing the full path to the location of the file referenced by the external data property, excluding the file name.
- FILE_PATH_AND_NAME: This function returns a string containing the fill path and file name of the file referenced by the external data property.
- FILE_SIZE: This function returns an integer that is the size of the file in bytes referenced by the external data property.

These functions can be used in various situations related to working with the file referenced by an external data property stored on the client device. Some use cases for these functions are provided in the information on implementing this functionality.

Data Synchronization for File Transfer
When defining the synchronization logic to retrieve files from the back end system for storage on the client device, it is necessary for this logic to perform a set of operations related to each file:

1. Retrieve the parent object that will contain the collection of attached documents. This logic is contained in a step type matching the back end system.
2. Retrieve the information about those files to be stored in the parent object, including the file's name, its storage location on the back end system, and excluding the file itself. This logic is contained in a step type matching the back end system.
3. Retrieve the file from the back end system. This logic is contained in a file document management step.

One of the keys to this functionality is that the back end system must have information available about which files are associated with which objects, and the specific location of those files on the back end system so that the Agentry Server can retrieve them. This location must be one to which the Server has read access.

While the above operations includes three distinct steps, it may be possible in some back ends to perform the first two operations in a single step definition. As with other synchronization operations, what data can be retrieved at which point is dependent on the back end system's structure and the interface type in use (i.e. Java, SQL, HTTP-XML). The retrieval of the actual file always requires a the definition of a file document management step.

When defining the synchronization logic to upload files to the back end system for storage in the back end system, it is necessary for this logic to perform a set of operations related to each file:

1. Upload the file to the Agentry Server and then tot he back end system. This logic is contained in a file document management step.
2. Update the back end system with the necessary information about the file, including the business entity with which it is associated, revision/upload date, and any other information the back end requires for the file. This logic is contained in a step type matching the back end system.

Supporting Infrastructure Tasks
In addition to changes within the Agentry application project, there are certain tasks that should be performed in support of implementing this functionality:

• Verify the proper applications are installed to client devices in the environment for the file types expected to be a part of this functionality.
• The file system on the host system fro the Agentry Server must contain a location to which the Agentry Server has read-write access. The Agentry Server typically requires a location

to temporarily store the files being transferred. This location should be noted for reference during the development and implementation of this functionality.

- The location to which files will be written on the client devices should be determined. This location is created by the Agentry Client during synchronization if it does not exist, and can include components based on the parent object of the file, as well as the user's ID.
- The method in which files are stored by the back end system should be known. The processes, tools, and/or other items that system employs should be evaluated to determine if they can be used by the file document management steps of the mobile application during synchronization. Typically such items include command line processes that can be called to check out and check in files from a version control system, or a process that extracts the files from a database when the files are stored in this manner.

Developing File Transfer and Attached Documents: Process Overview

The following are the high-level tasks in implementing the file transfer and attached documents functionality in an Agentry mobile application. Each of these items is discussed in detail in the series of procedures provided after this overview. Refer back to this overview of tasks as a check list of tasks when implementing this functionality.

1. Define the object that encapsulates the files to be transferred and attached (hereafter generically referred to as the "file object") within the mobile application project.
2. Define the downstream synchronization logic to retrieve the files from the back end system for storage on the client.
3. Define the user interface behaviors for view files stored within a given parent object, and any related behaviors for those files, including viewing.
4. Define any transactions to add files to a parent object on the Agentry Client.
5. Define the upstream synchronization for files attached to documents on the client.
6. Define the logic, transactions, and upstream synchronization for files to be updated to the back end when they have been modified on the client after having been downloaded from the back end.

Note that if any of the functionality related to one or more of the above procedures is not to be a part of the mobile application it is not necessary to perform that procedure. For example, if users will not be modifying the files downloaded from the back end on the client, it is unnecessary to define the logic and transactions to check for such changes.

Defining the File Object

Prerequisites

Prior to performing this procedure, the following items must be addressed:

- The storage location of the files on the client device must be determined and noted for reference in this procedure.
- The parent object in which the file object will be stored must already be defined.

- The full path to the storage location of the files must be known

Task

In this procedure an example of defining an object to encapsulate the files being transferred and attached to other objects within the mobile application is provided. The primary focus of this procedure is on the external data property that references the file. Additionally, the overall architecture and components of the file object are explained during this procedure, including the reasons for their implementation and how they may be used in related functionality.

The sample application to be used in this procedure is Mobile Northwind, which contains the module data structure of Customers -> Orders -> OrderItems. Added to this structure will be a collection property for the Document object that is defined. This will be added as child to the Customer object, making it a sibling of the Orders in the aforementioned module data structure.

1. Begin by defining a new object within the module of the Agentry application project. For this example we name the object definition Document.

2. Next add a string property to this object that will contain the name of the file the object encapsulates. Note that this property need not be a string, but can be any other data type desired. It's value is converted to a string when used to name the file being saved. It is recommended it be defined as a string initially to allow for more variability in the file name. In this example we name the property FileName and has the other following attributes. (Attributes not listed here should be left set to their defaults for this example. They can be set as needed for implementation specific requirements):

   - **Minimum Length:** 1
   - **Maximum Length:** none *(note this can be set to a maximum if deemed necessary to the application; however, it should be sufficiently sized to ensure the file name value is not truncated)*
   - **Trim:** true *(this can be important to prevent the file name from containing leading or trailing spaces that could cause issues in numerous file related operations)*

3. Now define a property to contain the location of the file on the file system of the Agentry Server's host system. This string property is used when the source location of the file is on a the file system. Define the property to be a string with. In this example we name the property BackEndFile with the following attributes:

   - **Minimum Length:** 1
   - **Maximum Length:** none
   - **Trim:** true

   This property is always required regardless of where the files are stored, including in a database system or file control system. It is needed during synchronization, as will be apparent in the procedures on this topic.

4. Define any other properties that may be needed for the file object. These could include values for reference purposes, such as the date and time the file was last modified, a

revision number for the file, or other similar information. Keep in mind any values must be accessible from the back end system in order to be downloaded for the object.

5. Next the external data property is defined. This property provides the reference to the file as stored on the client device. Begin by adding a property to the object with a data type of External Data. In this example the property is named File. Set the attributes of this new property according to the following:

a) Set the Client File attributes to specify the source for the file name, behaviors related to if/when to delete the file, whether or not it is read-only, and the behavior of the file dialog when attaching files locally:

These attributes are set to reference the property of containing the name of the file as retrieved from the back end system with the File Name attribute set to the string property FileName. We have also defined the file to never be deleted on the client. It is possible to define the file to be deleted only when it was created by the Agentry Client during data synchronization, or to always delete file when the parent object is deleted. Finally, the file has been defined to be read-only, which means users are not able to modify the file on the client device. The File Extension attribute has been left blank, meaning files will keep the extension as returned during synchronization. It is possible specify an extension here, which changes the file extension of all files to the value

provided, regardless of the extension of the file when it is downloaded or attached locally.

b) Next the attributes specifying where the file is to be stored on the Agentry Client are set:

The settings defined here result in a client device storage location specific to the mobile application and the parent object to which the files are attached. The base path under the Windows 9.x/NT/2000/XP section is set to "Absolute Path." The Relative Path attribute then contains the full path to the location where the files are to be stored. Other options include the various standard windows lactations, such as My documents, My Pictures, etc. For the Windows CE section, the attribute Use Path is selected, which will replicate the path for Windows desktops on the Windows Mobile devices. The drive letter is removed from the path.

6. Complete the creation of this external data property clicking the **[Finish]** button. Then, review the properties for the document object. For this example, the following now exist (others may be included in file objects like this based on need, with these being considered the bare minimum:

- **FileName:** A string property that contains the name of the file as it will be stored on the client device. Note that this may different from the file name in the back end system and is set during synchronization, which can include both the back end name and/or other

values available at run time. The synchronization procedures provided for this topic address this behavior.

- **BackEndFile:** This property contains the full path and file of the file to be downloaded from the back end system. This is referenced by the server processing for synchronization.
- **File:** External data property that references the full path and file name for the file stored on the client device.

7. The next step for the file object is to set the key property of the file. View the Object tab in the Properties view of the Agentry Editor. Change the Key Property attribute here to the property that stores the name of the file (in our example the FileName property).

8. Finally, create navigate in the Agentry Editor to the object that will contain the instances of the file object. In our example, this is the Customer object. Add a property to this parent object of type collection and define it to stored instances of the file object just created.

With the complete of this procedure the object to encapsulate the files associated with (or "attached to") some other object has been defined. The parent object has been defined to contain a collection of these objects, which is to be populated during synchronization and also when files are attached on the client device.

Next

The next area of functionality to implement is the downstream synchronization processing. See the procedure on this topic for guidance on this procedure.

Defining the Download Logic for File Transfer

Prerequisites

The following items must be addressed prior to performing this procedure:

- The object definition encapsulating the files to be downloaded (hereafter referred to as the file object) must be defined. The collection property to store the file object instances must already be defined in the parent object.
- This procedure assumes a fetch exists within the application project to retrieve the parent objects of the file object.
- The location of the files in the back end system must be known and the manner in which they are to be retrieved from that location should be determined. This is especially true if a version control system is to be accessed or if the files are stored in a database.
- Agentry Client Agentry Server

Task

This procedure describes how to define the steps and step usage definitions to retrieve files from the back end system to be transferred to the Agentry Client. In the example used here the files are stored in the version control system Subversion. As a part of downloading the files,

the command line process `export` is used. This process extracts a revision of a specific file from the repository and copies it to a designated location on the file system. This copy is not maintained by the version control system, which is a desirable behavior for this logic as the copy we create will be temporary and will be removed when the Agentry Server has finished transferring it.

This procedure describes what information is needed prior to the retrieval of the actual steps. How this information is retrieved will vary from one system and one back end type to the next. However, the general information required for this processing is the same.

As will be illustrated in the portion of this procedure related to the object read steps, one of the key items is to define the proper run behavior for the Document Management step that actually retrieves the files. Files are retrieved one at a time. Therefore, the document management step must be defined to run once for each file to be retrieved. This is controlled by the Run attribute of the object read step definition that uses the document management step. Note that only the object read step definition type contains the proper setting for the Run attribute. This step cannot be run to retrieve files in a fetch step usage definition.

In order for the document management step to run properly, therefore, the files to be retrieved must be known in advance. This is a part of the general information retrieved from the back end, where presumably the information relating the files to the business objects is located. this information is used to instantiate the file objects and, therefore, must include the value of the key property for each of those objects. In the architecture recommended here, this is the name of the file to be retrieved as it will be stored on the Agentry Client.

In this example, the Customer object contains a collection of file objects named Documents. A fetch already exists to synchronize the Customers collection, including it's nested collections Orders and Products. To this processing we will add the steps to get the document objects and the actual files to be referenced by those objects.

1. First, define the step to retrieve the property values for the file object, other than the actual external data property. In our example the file object includes the values of the FileName and BackEndFile properties. As with the synchronization of any nested collection, this processing must also include the key property of the parent object, which in this case is the CustomerID. For the sample Mobile Northwind application a SQL statement is written that selects the BackEndFile location and the FileName values from the table CustomerDocuments in the database. The step definition is a SQL step named GetDocumentInfo.

2. Next the document management step must be defined to retrieve the actual files to be transferred. This consists of several pieces, which are broken down over the next steps of this procedure. Begin by adding a step to the module of type File Document Management that uses the file system connection within the project. In the second wizard screen, select the attributes to specify which object is to contain the files it retrieves. In our example application the selection for this attribute is "Object - Document." Steps of this type need to know for which definition they will be retrieving data to provide access to the proper

values to the logic they contain, as will be demonstrated in the following steps. Set the name and group attributes and then finish the wizard.

3. Now that the step is created, the next task is to define the document mapping(s) for the step. Document mappings define the relationship between the properties of the definition selected when the step was created, and the command the step will execute at run time. In the Properties view of the Editor, select the tab Document Mappings. To add a new mapping, select the add button above the empty list. The Add Document Mapping wizard displays.

Set the attributes of Property, which should be the external data property for the file object. Then, set the Output Type, which is how the file to be transferred is output by the command we will write to retrieve the file from the back end. In our case, the file is retrieved from the version control system and saved to the local file system of the Agentry Server, so the proper selection is "File created by command." Other options are to capture the processes output to standard error or standard out, or the processes exit code. These are discussed in the later, optional steps to this procedure. Leave this wizard open to set the next attribute, File Name, as discussed in the next step of this procedure.

4. First, check the box below the File Name attribute marked Delete File. This setting removes this temporary file from the file system for the Agentry Server once the Server has completed processing it. The File Name attribute specifies the name of the file as it will exist on the file system of the Agentry Server once it has been created by the document management step. This file name can be set to any value, as it is only temporary and does not need to match the file name as it will be stored in the Agentry Client. This attribute can, and likely will include SDML data tags. In our example, the file name is a combination of the fileName property of the file object, the CustomerID of the parent Customer object, and the user's login value. This combination guarantees a unique file name for the file on the Server. This attribute can, and in most cases should include the path information to where the file is to be stored. This location should be one to which the Agentry Server has read-write access and is typically one created specifically for use by the Agentry Server. The full value of this attribute in our example, then, is:

```
C:\MobileNorthiwndFileTransfer
\<<document.fileName>><<customer.CustomerID>><<user.agentryID>>.t
mp
```

Note that any property value from the object is valid with the exception of the external data property. This value, then creates a file in the specified location, with a name guaranteed to be unique for each user and customer object combination. As demonstrated shortly, the values used here to create the file name should be noted, as they will be needed in the creation of the command for the document management step. Once the File Name attribute has been set, finish the wizard.

5. OPTIONAL: Additional Document Mappings can be defined for this document management to capture other information from the command it executes. The Output Type setting controls this behavior, with the options Command Exit Code, STDERR, and STDOUT. Each of these values produced by the command can be mapped to a property

within the definition for which the document management step was defined, e.g., the Document object in our example. The Command Exit Code and STDERR options are typically selected to capture any errors to support processing them accordingly. STDOUT is selected when the process executed by the document management step streams the file data to standard output rather than creating a file on the file system. STDOUT can be mapped to the external data property when the process behaves in this manner.

Create any additional document mappings, as needed. The attributes File Name and Delete File are not available when these options are selected, as there is no physical file included in the processing.

6. Next the command to retrieve the file from the back end is written. In the Properties View select the Document Management Script tab. The command can be written either directly in the Command attribute (set to <<script>> by default) or in a separate script file. To run a separate script, the default <<script>> data tag should be left as the value for the Command attribute. The separate script file is the default behavior, and the Command field is typically only used when a single short command is needed. In most real-world applications the command is stored in the separate script file. Click the edit button tot he right of the File attribute on this screen to display this script, which is shown in the text editor view.

7. The command executed can be any series of one or more command line utilities that may be needed to retrieve the file from the back end storage system. In our example we assume the version control Subversion contains a repository of files for customers in the Northwind system. We extract the files, therefore, from this system. We use the export command in Subversion for this processing, writing the file to the file system for the Agentry Server. Regardless of the tools used, the command written here must extract the file from the back end system, making use of the information retrieved previously regarding which file and where it is located. the command can then either write the file to the file system, or stream it to standard output for the Agentry Server to capture. The proper behavior must be matched to the defined document mappings for the Document Management step. The following is the example command used for the Mobile Northwind application:

```
export <<document.backEndFile>> C:\MobileNorthwindFiles
\<<document.fileName>><<Customer.CustomerID>><<user.agentryID>>.t
mp
```

The above command executes the export utility for the Subversion system. It extracts the file from the location where it is stored in the repository, which is the value returned by <<document.backEndFile>>. The file is then written to the location where the Agentry Server expects it. The file name is then set to match the values of the name of the file in the back end (<<document.fileName>>), the parent CustomerID property values (<<customer.CustomerID>>), and the user's client login (<<user.agentryID>>).

8. Now that steps are defined, they must be used. For file transfer functionality this requires the use of object read steps. In our sample Mobile Northwind application there already exists a fetch named GetCustomers, which targets the top-level Customers collection. Therefore, the read steps will be added to the Customer object and read into the Documents collection property of that object. First, the GetDocumentInfo step must be run. Remember that this is the first step we defined in this procedure and is the one to retrieve the information about the file to be transferred. Create a read step within the object to run this step. The Run attribute for the mobile application is Run one time, as all information about the files can be retrieved in a single query. For other systems, determine if the logic can perform this type of batch processing and set the Run attribute accordingly.

9. Next we define the read step for the document management step. In the Mobile Northwind application this is the GetCustomerFiles step. When defining this read step, the attributes should be set to reference the document management step and to read into the collection of file objects. The Run attribute must then be set to Run Once per Collection Object. This will execute the document management step once for each of the file objects contained in the collection, returning the file based on the other property values of that object.

10. Once these read steps are defined, verify the proper order. The first step run should be the one that retrieves the information about the files to be transferred. The result of this step's execution is the creation of the file object instances to be stored in the collection. This should then be followed by the document management step that retrieves the actual files. Since this step is executed once per collection object, the step to create the file objects must come first, followed by the document management step.

With the completion of this procedure the file transfer functionality is added to the mobile application. If following the steps and general architecture of this procedure, the following is the overall processing and data flow of this change:

1. A transmit is initiated by the Agentry Client. The fetch for the top level collection is processed as it was prior to this modification.

2. The object read steps of the object type being targeted by the fetch are processed by the Agentry Server. When the step to retrieve the information about the files to be transferred is run, the Agentry Server processes the results of that step and instantiates the file objects, storing them in the collections of the parent object instances. These file objects include the information about where the file is located in the back end, and the name of that file.

3. The document management step is run next. It is executed once for each object instance in the collection of file objects. For each execution, the file is retrieved from the back end system and stored in a temporary location for the Agentry Server to access it. This location is determined by the command in the document management step.

4. When the command completes execution, the Agentry Server uses the information in the document management step's document mappings to read the file from the location the mapping specifies. It sets the property the mapping specifies within the file object to reference the file it reads in.

5. The file is transferred by the Agentry Server to the Agentry Client, where the file is stored on the client device according to the storage location defined in the external data property.

6. The above steps concerning the execution of the document management step are repeated once for each file object until all have been processed.

Next

If not already accomplished, the user interface to display a list of the files downloaded, as well as to possibly allow the user to display those files and edit them needs to be implemented. See the procedures in this tutorial on the user interface and actions related to the files downloaded to the Agentry Client.

Defining the User Interface for Attached Documents

Prerequisites

The following items must be addressed prior to performing this procedure:

- The object that encapsulates the files stored on the Agentry Client (hereafter referred to as the file object) must already exist and all properties it is to contain must be defined.
- Though not a technical requirement, it is generally good form to define the synchronization logic for the objects prior to defining the user interface.
- A part of the functionality implemented in this procedure includes the ability for users to select a file from a list of those on the Agentry Client and to display that file in the application associated with the file type. This functionality can be implemented in the Agentry application project at any time. However, it will not function properly unless the application is installed to the client device and is associated with the file type. This requirement must be met before deployment of this functionality.

Task

This procedure described how to build user interface definitions and functionality around files that have been transferred to the Agentry Client and/or attached to objects locally. The Agentry Client does not display files directly within the client interface, nor does it provide a means to edit these files to the user. However, the files can be displayed in an application for the file type, provided that application has been installed and is associated with the file type on the client device.

To list the files attached to a parent object, a list screen or one of the list types of detail screen fields can be used. The definition of these lists is the same as for the display of any other collection property. The collection is selected to be listed, and the properties from the object type within the collection are selected for display in columns for fields.

In this procedure a list is defined to display a collection of file objects. An action is then defined and a button is added to execute that action that displays the file currently selected in the list. This is accomplished with the definition of an action step of type Windows Command. This type of action step is used to execute commands on the client device. If the path to a non-executable file is defined as the command for the step, the Windows OS will execute the application associated with that file type and open the specified file.

In the example used in this procedure, the module data structure is Customers -> Documents, where Documents is a collection of Document objects to encapsulate the files attached to Customer objects. A screen set to display details of a selected Customer object is already defined and includes detail screens for the other properties of the Customer object. It is to this screen set, named ShowCustomerDetails, that the list screen will be added. The functionality to be implemented includes a list screen with columns displaying the name of the file stored on the client, and another to display the fill file path. On this list screen an action can be executed via either a double-click of an item in the list, or via a button click that displays the selected file in the application associated with its type.

Note that similar behavior can be defined to list the Documents object using other list controls available in Agentry, including List Tile View fields and List View fields, or any display definition that lists collections.

1. We will begin by defining the action to display the selected file in its associated application on the client device. Begin by adding a new action to the module. Set the Name and Display Name as deemed appropriate. The For Object attribute should be set to the file object containing the external data property. In the sample Mobile Northwind application this is the Document object definition.

2. Once the action is defined, add a new step to it of type Windows Command. Again set the Name as Desired. Then set the attributes of this action step according to the guidance provided below this example:

- **Command:** To display a file, the value here must be the fill path and file name of the file to open. A format string can be used here for the External Data property of the file object (In the Mobile Northwind this is the File property), e.g. `%File`
- **Wait:** This attribute controls whether or not to wait for the executed process to complete before continuing execution of the action. When selected, a timeout value is required. For this operation such behavior does not make sense, as there is no way to know how long the user would view the file and setting a wait behavior would prevent the user from accessing the Agentry Client while the external application is still open. Therefore, leave this attribute set to false.
- **Error Message:** This is displayed when an error occurs executing the command. For this step, the message can indicate there was a problem opening the file. Note that the operating system may also return an error message should this operation fail and both will be displayed to the user.
- **Timeout Message:** This message has no affect on this step definition and can be left blank, as the Wait attribute is set to false.
- **OK and Cancel Labels:** These are the buttons displayed in the error message dialog. In most cases either the OK or Cancel would be enabled, but no both. In the Mobile

---

Northwind application this action contains only the single Windows Command step, with no additional steps being executed. Therefore, the OK is sufficient to acknowledge the error message on the Client. If additional steps are included in the action after the Windows Command step, then it should be determined if the user should be allowed to continue the action or not, and to enable the buttons appropriately.

This step will now pass the full path and file name to the operating system, which will then result in the OS attempting to open the file in the application associated with the file type.

3. Now the list to display the file objects stored in a collection can be defined. This can include a list screen, a List View detail screen field, or a List Tile View detail screen field. The collection should be the collection property of file objects. In the Mobile Northwind application project this is the Documents property of the Customer object. Typically the file name and file path are displayed. For the Document object this would be the FileName string property and the File external data property, respectively. The external data property type always displays the fill path and file name of the file being referenced when a UI definition targets it for display.

4. Finally the control(s) are defined to execute the new action. This can include a button definition on the list screen, as defined in this example, that targets the selected Document object in the list. Additionally, or in place of the button, a double-click on-item action can be defined for the list screen that executes the OpenFile action as well, targeting the Document object. If a List View or List Tile View field is defined to list the Documents collection, a button can be defined on the parent detail screen that targets the selected object in the list field.

Once this procedure is complete, a list of the file objects for a given parent object is defined. An action also exists that allows the user to display the selected file in the application on the client device associated with that file type.

Defining Locally Attached Documents Functionality

Prerequisites

The following items must be addressed prior to performing this procedure:

- An object must exist in the application project to encapsulate the files to be attached to a parent object (hereafter referred to as the file object).
- Options exist to specify the location in which files can be selected on the client device. If this behavior is desired, this location should be determined and note for this procedure.
- Options exist to restrict the type of file a user can attach on the client device. This is controlled by the file extension. If this is desired behavior, the file type and extension should be noted for this procedure.
- The manner in which files should be stored in the back end system should be researched, including details on how to add the files to that storage location or repository. This processing must be a part of the logic defined to upload the file to the back end.

Task

In this procedure the tasks necessary to allow users to attach files to an object on the client, selecting that file from the client's file system, are provided. This functionality includes the use of an Add transaction to create a new file object and to capture the properties of that object, including the file to be attached, from the user. A transaction screen set is defined that includes a filed to display the external data property. The field for this property type displays a file dialog to the user when it is selected in the wizard, allowing the user to navigate to the file to be attached.

The synchronization of this data includes steps that update the back end system with the information about the file, including it's name, the parent object it is attached to, and other information as may be required by the back end system. Typically this information is updated to the back end using a step definition of the type matching that back nd, i.e., a Java, SQL, or HTTP-XML step. A document management step is defined to perform the actual file upload and back end processing necessary to store that file in the location or repository where files are to be stored.

1. The first task is to define an Add transaction for the file object. As with most add transactions, this one should include transaction properties matching all defined object properties. In the Mobile Northwind example application, the Document object encapsulates the file attached to a parent object. An Add transaction named AttachDoc is defined that targets this object type. It contains the following properties, matching those in the object:

   - **File:** External data property, initialized to empty, i.e. Initial Value: "Auto-Initialize." There are some optional changes to the default attribute settings for this property, discussed in the next step of this procedure.
   - **FileName:** String property. This property is modified from the default initialization to a rule after data entry. A subsequent step in this procedure describes this modification.
   - **BackEndFile:** String property, initialized to an empty string. This value is not needed for transaction processing, as it only relates to downstream synchronization (e.g., fetch processing). However, it is good form to initialize the value in the add transaction.

   When the transaction is created, assuming the properties from the object were all selected in the add transaction wizard, there will be matching properties in the new transaction definition. In subsequent steps, a property to capture the parent object's key property value is added to this transaction. First, the initial value of the FileName property must be defined to capture the name of the file on the client device.

2. OPTIONAL: It is possible to provide a white list of files for file types the user can select, and to also provide a black list of specific files or locations from which the user may not make a selection. Once the external data property has been defined, viewing it in the Properties View of the Agentry Editor displays the three attributes of **File Filters**, **Filter Description**, and **Restricted Files**. **File Filters** can be set to one or more file types the user can select. Only files of that type will be displayed in the file dialog, and wild cards can be used (e.x., `*.doc;  *.jpg` - Only MS Word Documents and JPEG images). The

Restricted Files attribute must be set to an absolute path to the directories, or specific files, that the user will be allowed to select. Multiple paths can be provided and must be pipe delimited. (e.g. `\\Windows | \\Program Files` - No files can be selected from any path under these two locations.)

If these attributes are set, the file dialog on the Agentry Client will present only those files and file types defined in the **File Filters** list, and will prevent users from selecting the files found in the **Restricted Files**. This is optional behavior and is not a requirement of this functionality. However, it is recommended that options for these attributes be at least considered to prevent users from attaching files that should not be transferred, e.g., executables, resource files, etc.

3. The FileName property must contain the name of the file on the client device's file system. This value is not needed during synchronization, but is used for display and reference purposes on the Agentry Client. The property is therefore modified by changing the **Initial Value** attribute to "Rule - after data entry." The rule is then defined for this initialization to use the FILE_NAME function, taking the external data property in the transaction as its sole parameter. It returns just the name of the file, to store in the FileName property as required:
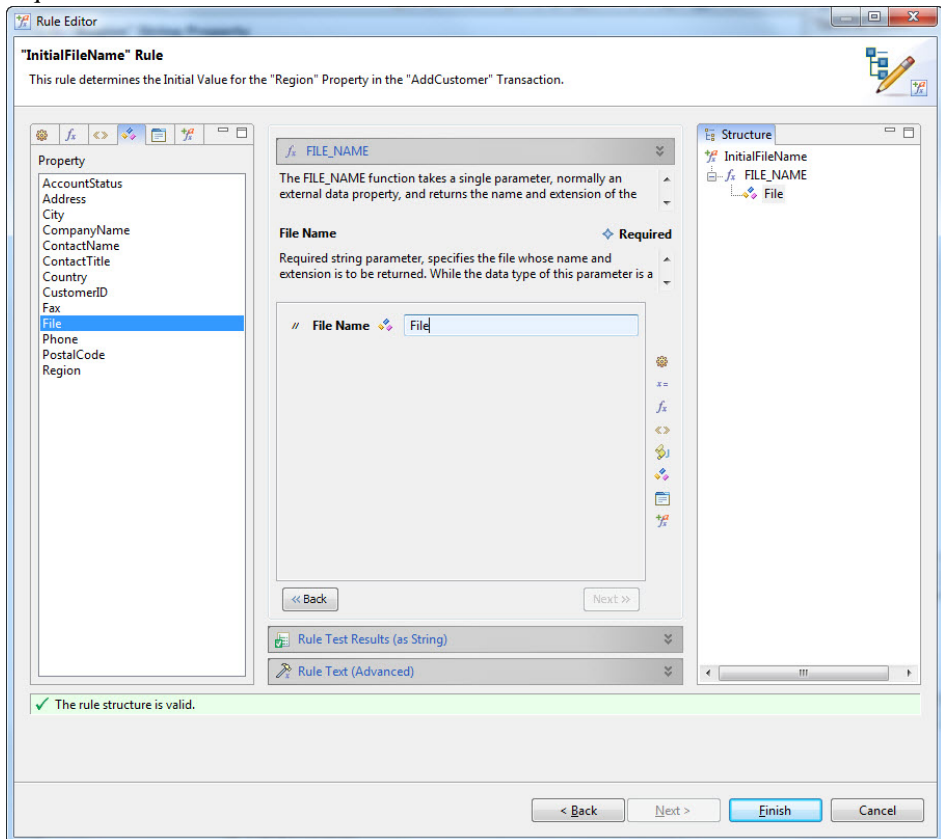
With the change to the property's initial value attributes, and the definition of the rule, the FileName property within the transaction is now defined to capture the name and extension of the selected file as stored on the client device. The rule contains a simple structure consisting of the FILE_NAME function that takes the external data property (File in the Mobile Northwind example) as it's single parameter, returning the name of the file it references.

4. Now that the transaction has been defined, the synchronization logic to update the captured information can be defined. We begin with the document management step responsible for updating the actual file to the back end system. Create a step of type File Document Management. Use the defined File system connection. Advance the wizard and on the enxt screen, set the **Used By** attribute to "Transaction - *AddDocTransactionName*", where the second portion is the name of the transaction defined to add a file to a parent object. Set the **Name** and **Group** as desired and finish the wizard.

   In the Mobile Northwind example application the step is defined to be used by the transaction AttachDoc. The name of this step is CommitCustomerDoc. Next document mappings for this new step must be defined.

5. The document mappings for a document management step used by a transaction define the source property in the transaction referencing the file being transferred, and how that file is to be provided to the command run by the document management step. For the Mobile Northwind example application, the File property contains the reference to the file being transferred. The command being run includes command line processes within the Subversion version control system. These commands expect the file to be stored on the file system. Therefore, the document mapping is defined as follows:
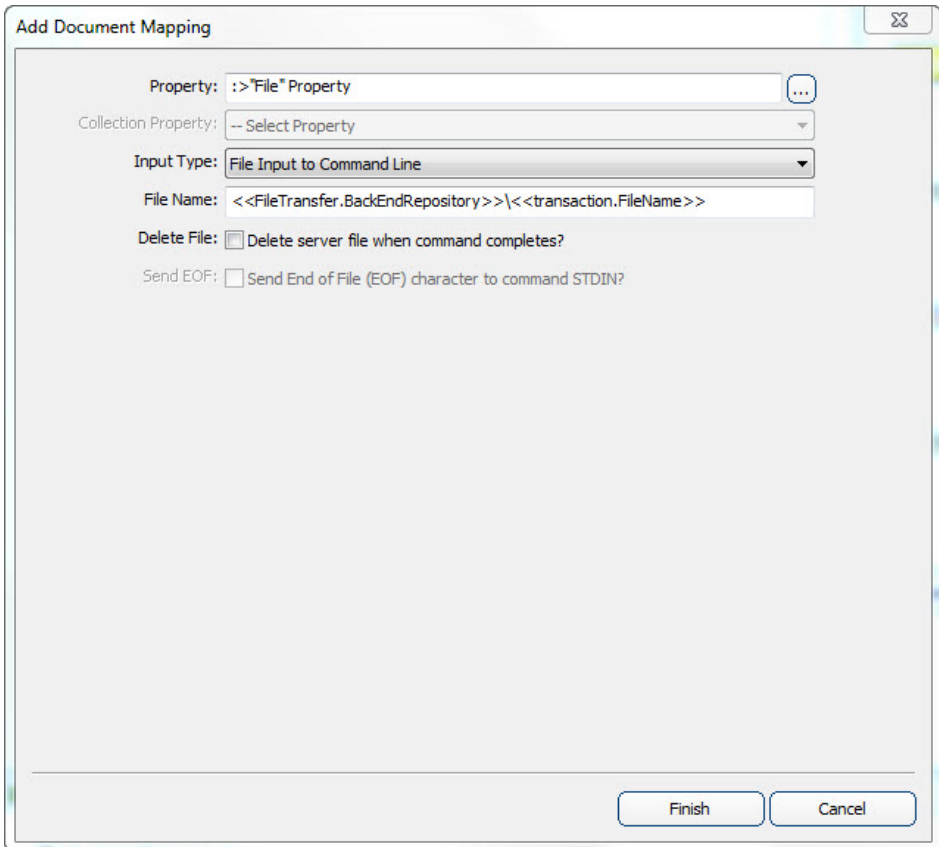
- **Property:** The external data property of the transaction.
- **Input Type:** How the file is to be provided to the command run by the step; for this example this is set to "File Input to Command Line," meaning the file is stored on the file system by the Server prior to executing the command for this step. The other option is to stream the file data from the Agentry Server to a command being executed.
- **File Name:** Where the Agentry Server should write the file to on the file system. This is disabled if streaming the file to the command. Note that the FileName property is provided for this attribute as a matter of convenience. This attribute specifies the name the file should be given when written to the file system by the Agentry Server. The value specified here can be any desired using any values available to the document management step. Like other steps, this is any value in the transaction, any globals defined in the application project, and any of the other available values for back end processing.
- **Delete File:** Whether or not the file should be deleted. In this example the file is being written to the working copy location of the Subversion repository and therefore should not be deleted. In other cases, if this location is a temporary storage to allow the file to

be processed and moved by the command, then this option should be selected to remove the file when its processing has been completed.

Once the document mapping is defined, the Agentry Server will process the file as that mapping dictates, including which property contains the file, how to provide the file to the document management step's command, and whether or not to clear the file from the file system when processing is completed.

6. The command for the document management step must now be written. This command processes the file provided by the Agentry Server, either via a file written to the file system, or via a stream to standard input of the command being executed. The command written for the Mobile Northwind application is stored in a batch script and contains the Subversion commands add and commit. The add command adds a new file to the Subversion repository and the commit command commits that addition. The command is then written as follows:

```
add <<FileTransfer.BackEndRepository>>/
<<transaction.FileName>>
commit <<FileTransfer.BackEndRepository>>/
<<transaction.FileName>>
```

When this command is defined, the file is first added to the repository and then subsequently committed to the repository. This is a required set of operations for Subversion. Other version control systems will handle new files differently. This processing assumes the file did not exist in the repository prior to this operation. If files are to be updated in any system, Subversion or others, the order of operations would be different. In general this command should be written only with a full understanding of how the back end stores files and how different situations are handled, e.g., new files vs. updates, etc.

7. In most systems it is necessary to create the logical link between the file and the business entity to which it is attached. For example, in the Mobile Northwind application the table Customer Files exists in the Northwind database and contains the path to the file's location in the version control repository and the Customer ID of the customer with which the file is associated. For this application a SQL step is defined containing an insert statement that adds the needed record to the table for a new file. Other information can be included, such as date and time information, file size, or any other meta data that may be needed.

8. Once the steps have been defined, they must be used by the transaction. A server update step is added for each step involved in the processing of an attached file. In the Mobile Northwind application this includes the document management step and the SQL step discussed in this procedure. In most cases the recommendation is to first execute the Document Management step, followed by any steps that provide the link or other information about that file. This order of operations prevents there being any "empty links" should an error occur with processing the file.

9. The final step in this procedure is to define the user interface for this functionality. This procedure is the same as defining the wizard for most other transactions. The screen set is defined, including the proper platforms, and finally the detail screen(s) and fields to

capture the data from the user. The one item of note is the definition of the field for the external data property. The edit type of this field can be set to "External Data", or left set to "-- Default --". At run time, the Agentry Client recognizes that the target property of the field is an external data property and automatically displays a field that includes a control to launch a file dialog to allow the user to select a file from the file system. However, selecting the External Data field type explicitly can make it clearer when returning to this definition in the future. Once the wizard has been defined, the action for it and the transaction can be defined, followed by the control(s) to execute that action.

When this procedure is complete, users will have the ability to navigate to and select files on the client device. These files are then attached to the parent of the file object. During synchronization, the transaction processing will include uploading the file to the Agentry Server and then storing that file in the back end system

Next

A variation on the above functionality is the ability to allow users to select multiple files from the file dialog and attach them in one operation to a parent object. This requires the following differences in the definition of the transaction:

- Define a collection property of file objects in the transaction in place of the external data property
- When the detail screen field is defined for the wizard where files are selected, its edit type must be set to "External Data" before the collection property in the transaction can be selected.
- The Document Mapping for the document management step must be defined for the collection of file objects in the transaction. An additional attribute, Collection Property, is enabled where the external data property from the object in that collection is selected.
- The Document Management Script's Command (or more likely the script containing the commands to be executed) must include iterative processing for each object in the collection. This processing can be provided using the SDML function tag `<<foreach...>>`. The commands execute within this loop then would operate on a single file. `<<foreach...>>` provides the context of each object instance in the collection so that each file can be processed individually. see the example below for the pseudocode representing how this command script would be written.

```
<<foreach Documents

    copy <<my.FileName>> to <,FileTransfer.RepositoryDirectory>>

    add <,FileTransfer.Repository>>\<<my.FileName>> to repository

    commit <<FileTransfer.Repository>>\<<my.FileName>> to
repository
```

>>

The Agentry SDK

The Agentry SDK is a collection of resources provided to developers to support inter process communications between the Agentry Client and another process or application running the same client device.

Note: The ActiveX API and Agentry Client API should both be considered deprecated in the SAP Mobile Platform 3.0 release. They are provided solely for backwards compatibility in support of exisiting implementations in which either of these resources were used. With the release of the SAP Mobile Platform 3.0, the OpenUI SDK should be used for all new development work.

There are two options available for interacting with external processes. These options consist of ActiveX, or using the Client API. The ActiveX interface to the Agentry Client has been available for quite some time and has been expanded and augmented with continuing increases in the exposure of the Agentry Client functionality and data. The Agentry Client API exposes the ability to execute actions, instantiate and apply transactions, and request a rule evaluation, all from an external process running on the same device as the Agentry Client.

Included in the Agentry SDK, and in addition to the resources needed to make use of the API's it provides, are a handful of samples. It may be beneficial for the uninitiated to review these samples, and possibly even compile and build one or more of them in order to become more familiar with the overall structure and logic involved in building processes or controls that interact with the Agentry Client.

Agentry ActiveX Client API
The Agentry ActiveX Client API (ActiveX API) provides numerous resources for creating an ActiveX control that is displayed on the Agentry Client's user interface and that can interact with the Agentry Client in several different ways, including passing various types of data between the Agentry Client and ActiveX control, requests made by the control of the client to execute actions, and for the Agentry Client to be aware of various control-related events such as data entry and changes in focus.

The ActiveX API provided by Syclo includes numerous resources needed by the ActiveX control that must be included in the build and compile stages. There are methods within the Agentry Client that are exposed to the ActiveX control, as well as methods that are expected to exist within the control that will be called at various times by the Agentry Client to notify the control of certain events related to the control and/or the user's interaction with it.

Agentry Client API
The Agentry Client API has been provided to expose certain functionality within the Agentry Client to external processes. Similar to the ActiveX API, the external process can request the

Agentry Client to execute actions. In addition, the process can also request the Agentry Client to instantiate a transaction using values provided by the external process, and to then apply that transaction. The external process can also call through the Client API to request a rule evaluation and to receive the value returned by that rule.

A significant difference between the Client API and the ActiveX API is how the communications are supported. When using the ActiveX API it is a requirement that an ActiveX control be used and displayed on the Agentry Client's user interface. In some situations this is either not practical, or such a control makes no sense for the intended purpose. The Agentry Client API allows for interaction between an external process and the Agentry Client with such a control. The external process must be built using the provided resources in the Agentry SDK, and call the methods provided by the Client API to perform the desired processing.

System Support, Usage, and API Differences

Both the Agentry ActiveX Client API and the Agentry Client API are available for use with Agentry Client's running on Windows devices, desktops, and laptops. ActiveX is a Microsoft protocol provided exclusively for their family of Windows operating systems, and therefore cannot be used in conjunction with Agentry Clients for platforms other than Windows. The Agentry Client API is at this time available only for Windows platforms as well.

The primary driver for selecting the Agentry Client API for external processes or the Agentry ActiveX Client API is whether or not a control should be or is needed to be displayed on the Agentry Client's user interface. If there is no need for such a control, or if such a control does not make sense in the context in which the processing or work flow is performed, then the developer should investigate using the Agentry Client API for external processes. If, however, a control is needed, then the Agentry ActiveX Client API must be used.

Other differences include the functionality exposed and available by each API. The Agentry Client API for external processes allows for the execution of actions, the evaluation of rules, and the instantiation and application of transactions. All three of these processes are performed in the context of the module MainObject.

A Note on Changes to the Agentry SDK

For developers familiar with the Agentry ActiveX Client API, it is important to note some items related to the recent addition of the Agentry Client API for external processes. The most important difference is that the Client API does not utilize any of the resources provided by the SDK for ActiveX, nor does it use the methods, field types, action types, or other components provided to support ActiveX controls. The Agentry Client API for external processes is a separate entity and all support and resources related to its usage are mutually exclusive from the Agentry ActiveX Client API related resources.

No changes were made to the ActiveX API related to the addition of the Agentry Client API. Any existing ActiveX controls built using the ActiveX API provided previously are still supported and no change to them is needed.

Technical Overview - ActiveX Controls and the Agentry Client

The Agentry Client is capable of interfacing with an external ActiveX control installed to the same host device. This functionality is supported through the implementation of several separate but tightly coupled components within the Agentry architecture:

- The **External Field - ActiveX** Control detail screen field edit type
- The Agentry Client API containing methods that can be called by the ActiveX control
- The ActiveX control's proper implementation of the interface points (methods) expected by the Agentry Client
- The client action step type **External Field Command**

Using the above components together, it is possible for the Agentry Client to display an ActiveX control within the Client's user interface, to pass data from the Client to the Control, to pass data from the ActiveX control to the Agentry Client, to execute actions on the Agentry Client at the request of the ActiveX control, and to issue commands from the Agentry Client to the ActiveX control.

Provide Data to the ActiveX Control

The External Field - ActiveX Control screen field edit type includes in its definition a list of objects and properties, known as the Agentry Values for that field. Each of these properties is selected and added as an Agentry Value for the screen field. When a value is added, the specific property or object is selected and given an arbitrary name.

The ActiveX control can call into methods provided in the Agentry Client API to retrieve these values. The value to retrieve is specified via the name given to it in the Agentry Values list. All property values, regardless of the data type within the Agentry application project, are provided to the ActiveX control as strings.

Pass Data to the Agentry Client

The ActiveX control can call methods in the Agentry Client API to notify it of a change in its current value, or to indicate that the value has been fully entered. A call to these methods results in an immediate call by the Agentry Client back to the ActiveX control to retrieve the current value. The value returned by this subsequent call is then set as the current value of the External Field - ActiveX Control screen field. If no further changes are made, this value will set the value of the property target by the External Field if that field is displayed in a wizard screen for a transaction or fetch.

Execute Actions in the Agentry Client

The ActiveX control can call into the Agentry Client API to execute actions defined within the mobile application. This behavior requires the action or actions the control may execute to be listed within the External Field - ActiveX Control screen field. Only those actions listed within the detail screen field can be executed by the ActiveX control. The ActiveX control calls the appropriate method within the Agentry Client API, passing the name of the Action to be executed.

Issue Commands to the ActiveX Control
The Agentry Client can issue a command to the ActiveX Control. An action step of type
External Field Command can be defined to pass a string value to the ActiveX control. The
ActiveX control receives this command string via a method called by the Agentry Client when
the action step is executed. The action step defines the External Field - ActiveX Control screen
field that references the ActiveX control. This screen field must reside on a detail screen
within a screen set defined to display an Object. Screen sets displaying transactions and
fetches will not be valid options in the action step when it is defined.

External Field - ActiveX Control

The external field-ActiveX control edit type is defined to call out from a field to an ActiveX
control. Values may be passed to this control from the Agentry Client.

Use of this field requires an ActiveX control exist on the client devices and that control be built
using the Agentry ActiveX Control API, including the implementation of all Expected
Methods.

Using the **Agentry Data** and **Actions** tabs allows an ActiveX control to query Agentry for
data and for an ActiveX control to call for Agentry to execute actions. Agentry can also query
the ActiveX control for any values listed in the **External Values** tab.

External Field - Active X Control Attributes
The following attributes are specific to the External Field - ActiveX control field edit type.
These are in addition to the common field attributes:

- **ActiveX Class Name (Prog ID):** This attribute contains the class name that the Agentry
  Client will interface with for the ActiveX control.
- **Allow Scanning as Input:** This attribute specifies whether or not the field displayed will
  accept barcode scan values as input. This attribute will only impact fields displayed on
  detail screens used by a platform that supports scanner behavior and on client devices
  equipped with a barcode scanner. When value is scanned for the field, the ActiveX control
  expected method `AgentryUpdateScanData` to pass the barcode value to the ActiveX
  control.
- **External Values Tab:** The External Values tab is a list of values provided by the ActiveX
  Control. This will allow the Agentry Client to query the control for data. From the tab, you
  can add and delete value names from the list. The ActiveX control referenced by the detail
  screen field must include the proper processing within the
  `AgentryGetSpecificValue` method to return the value(s) associated with each of
  the External Values listed in this tab.
- **Agentry Values Tab:** The Agentry Values tab is a List of names and target paths for values
  within Agentry, made available to the ActiveX Control. From the tab, you can link Agentry
  data with the external values for the ActiveX Control. Both primitive data types as well as
  object instances and collection properties can be made available to the ActiveX control.
  The name associated with the selected data item is the identifier exposed to the ActiveX

control, which can call the `GetPropertyFromMappings` Agentry Client-Side API method, passing the name to retrieve the desired value.
*   **Actions:** Allows the ActiveX control to call for Agentry to execute actions. The Properties tab gives you a list of Actions and target paths. Within this list actions can be added and deleted. When an action is added it must also specify a target object for the action. The ActiveX control can call the `ExecuteAgentryAction` Agentry Client-Side API method, passing the name of the action to be executed.

Action Step Type: External Field Command

The External Field Command action step issues a command to an ActiveX control when executed. It references the External Field - ActiveX Control field to specify the control to which the command is to be issued. The action step passes the value of the defined command string to the ActiveX control, which is then responsible for receiving and processing the string command accordingly.

The defined command string within this action step type is passed by the Agentry Client to the ActiveX control through the expected method `AgentryExecuteCommand`. This method should be implemented to process the provided command string in the manner deemed appropriate for that control.

External Field Command Step Attributes
*   **Step Name:** This attribute contains the unique internal name of the action step definition. This must be unique among all steps within the same parent action.
*   **Screen Set:** This attribute specifies the screen set containing the detail screen within which the External Field - ActiveX Control field is defined. Valid selections for this attribute include any screen set defined to display an object definition. Screen sets for transactions and fetches are not valid.
*   **Screen:** This attribute specifies the detail screen containing the External Field - ActiveX Control field.
*   **External Control:** The External Field - ActiveX Control detail screen field that references the ActiveX control to which the command string is to be issued.
*   **Command:** The string to be passed to the ActiveX control's `AgentryExecuteCommand` method. This attribute value can be entered into the attribute field directly, or can be set to the return from a rule definition. A rule referenced by this attribute is evaluated in a string context and in the context of the action to which the action step is being added and the object for which that action is defined.

ActiveX Control - Features Log

The following provides a quick overview of the progression of the ActiveX control functionality supported by the Agentry Mobile Platform, and which release these features were implemented. Features and behaviors related to the ActiveX control are available beginning with the release in which it is listed, and in all subsequent releases unless otherwise noted.

Agentry Mobile Platform 5.2.8

The following features were added and changes were made to the 5.2.8 service pack release of the Agentry Mobile Platform in relation to the ActiveX control functionality:

- The ability to pass objects and object collections to the ActiveX Control. This is supported with the addition of the following client-side API methods:
  - `AgentryActiveXPropertyType`
  - `GetPropertyFromMappings`
  - `GetPropertyFromObject`
  - `GetPropertyType`
  - `PropertyAsString*`
  - `NextCollectionProperty`
  - `CollectionHasNextProperty`
  - `RewindCollection`
- The ability to issue a command to the ActiveX Control. This is supported with the addition of the following:
  - New Action Step Type: **External Field Command**
  - New Expected ActiveX Control Method: `AgentryExecuteCommand`

* - The `PropertyAsString` method is a replacement for the client-side API method `GetAgentryString`, which has been deprecated. This method is still supported for backwards compatibility, but should not be used in new development. Where possible, it is recommended that existing implementations are modified to use the `PropertyAsString` method in place of `GetAgentryString`.

Agentry Mobile Platform 5.1

The following features were added and changes were made to the 5.1 minor release of the Agentry Mobile Platform in relation to the ActiveX control functionality:

- Implementation of the Agentry Client-Side API to expose various aspects of the Agentry Client to the ActiveX control. Many of the following new features are supported with the implementation of this new API for the Agentry Client. The API itself is made available to the ActiveX control via the following:
  - `IAgentryActiveXControlHost` COM interface
  - `AgentrySetActiveXControlHost` - ActiveX control expected method (provides `IAgentryActiveXControlHost` COM interface to the ActiveX control)
- The ability to pass data from the Agentry Client to the ActiveX control, based on a request by the control. Support for this functionality is provided with the following additions:
  - Agentry Data List added to the **External Field - ActiveX Control** detail screen field edit type.
  - `GetAgentryString` - Agentry Client-Side API method

- Support for the ActiveX control to request an action be executed on the Agentry Client. This functionality is provided with the following additions:
  - Agentry Actions List added to the **External Field - ActiveX Control** detail screen field edit type.
  - `ExecuteAgentryAction` - Agentry Client-Side API method
- Support for the ActiveX control to notify the Agentry Client when its value changes or is fully entered. This functionality is supported with the following additions:
  - `ActiveXControlValueChanged` - Agentry Client-Side API method
  - `ActiveXControlValueEntered` - Agentry Client-Side API method
- Support for the Agentry Client to request values from the ActiveX Control via target paths. This functionality is supported with the following additions:
  - External Data List added to the **External Field - ActiveX Control** detail screen field edit type.
  - `AgentryGetSpecificValue` - ActiveX control expected method

Agentry Mobile Platform 5.0

The following features were added and changes were made to the 5.0 major release of the Agentry Mobile Platform in relation to the ActiveX control functionality:

- Support for the Agentry Test Script functionality, including recording and playback features provided in the Agentry Test Environment. This includes the following client-side API methods:
  - `AgentrySetScriptValue`
  - `AgentryGetScriptValue`

Agentry Mobile Platform 4.3

In the 4.3 minor release of the Agentry Mobile Platform, the ability to pass values read into the Agentry Client by the device's barcode scanner to the ActiveX was added. This feature is exposed in the External Field - ActiveX Control detail screen field type. This definition type was modified to include the Scanning attribute. When set to true, barcode values read in by the Agentry Client are passed to the ActiveX control referenced by the detail screen field.

Agentry Client ActiveX API Methods

The following methods are available within the Agentry Client and can be called from an ActiveX control installed to the same client device. This assumes the ActiveX control has been referenced and loaded by an External Field - ActiveX Control detail screen field within the mobile application running on the Agentry Client.

The information provided for each method includes its intended purpose and the description of the method parameters.

Included in these methods are those called to retrieve values from the mobile application, execute actions defined within the mobile application, and to notify the application that the value of the ActiveX control has changed or has been fully entered.

In order to retrieve data values from the application and to execute actions within it, these items must be listed in the External Field - ActiveX Control detail screen field definition within the application project. Only those values and actions listed as available are accessible to the ActiveX control at run time.

`ActiveXControlValueChanged`

This method should be called to notify the Agentry Client that the value of the ActiveX control has changed. The Agentry Client will evaluate any update rules currently in context for the detail screen. Note that this differs from the `ActiveXControlValueEntered` method, which should be called when the value has been completely entered in the ActiveX control. Rather, `ActiveXControlValueChanged` is called for each value change that Agentry Client should be aware of in order to process the changed value within update rules defined for other fields on the same detail screen.

Parameters
None

`ActiveXControlValueEntered`

This method should be called to notify the Agentry Client the value of the ActiveX control has been fully entered. The Agentry Client will evaluate any update rules currently in context for the detail screen, and will perform any additional operations based on the auto-next or auto-focus behaviors defined for the detail screen fields.

Parameters
None

`ExecuteAgentryAction`

This method can be called to execute an action on the Agentry Client. This action must be listed in the Actions list for the External Field – ActiveX Control detail screen field. This method blocks until:

- A wizard screen is displayed
- The Action completes execution
- The Action is canceled by an action step of type message
- The Agentry Client reports that it cannot execute the action

Note that the method will **not** wait for the completion of a wizard screen set. When such a screen set is displayed, the `ActionResult` parameter will contain a value of `Action_Pending`.

Parameters

- `ActionName` - Contains the definition name of the Action to be executed

---

- `ActionResult` - This value is set after the action is executed and indicates the status of the action execution. This will be one of the following enumerated values:
  - `Action_BackUp`: Reserved for future use.
  - `Action_Error`: Returned when the action could not be executed for any reason. Common causes for this return include if the action named is not one defined for the screen field, if another action is currently being executed on the Client, or if the defined target object for the action cannot be resolved.
  - `Action_Cancel`: Returned of the action is cancelled. This will only be returned if the action is canceled in an action step of type Message. This method does not block when wizard screens are displayed and therefore will not capture the cancellation of a wizard screen set by the user.
  - `Action_Pending`: Returned if the action executed successfully and displayed a wizard screen set.
  - `Action_Complete`: Returned if the Action executes and completes successfully without having displayed a wizard screen set.

GetPropertyFromMappings

This method retrieves the property named in the name parameter. The property is returned in the `property` parameter. This method can return any value listed in the Agentry Values of the External Field - ActiveX Control field within the mobile application. The name parameter is set to the name of the value as defined in the Agentry Values list. As of version 5.2.8 of the Agentry Mobile Platform, these properties can be of type collection and object in addition to the other property types support in previous versions.

Prototype

```
void GetPropertyFromMappings(BSTR name, VARIANT* property)
```

Parameters

- `name` - The name of the object or property to be retrieved, as defined in the External Field - ActiveX Control's Agentry Values list.
- `property` - The property retrieved by the method.

Return Value
None *(see the property parameter description)*

GetPropertyFromObject

This method returns the named property from the previously retrieved object. This method is called after `GetPropertyFromMappings`, with the `property` parameter to that method passed to `GetPropertyFromObject` as the `object` parameter. Note that this method will return an invalid result if the object parameter is provided any value other than an object.

The name of the property to be returned is provided as the definition name of that property definition within the application project. The property is returned in the `property` parameter.

Prototype

```
void GetPropertyFromObject
```

```
    (VARIANT const object, BSTR propertyName, VARIANT* property)
```

Parameters

*   `object` - The object from which the property is to be retrieved. This parameter should be the value returned in the `property` parameter of the `GetPropertyFromMappings` method when that parameter references an object; or from a previous call to `GetPropertyFromObject` when the property it returns is an object property.
*   `propertyName` - The name of the property definition to be retrieved within the object. This is the definition name of the property within the Agentry application project.
*   `property` - This parameter will be set to the property referenced by the `propertyName` parameter. The type will be set to `AXPT_Invalid` if the `propertyName` parameter contains a name not found in the object's properties.

Return Value
None *(see `property` parameter description)*

GetPropertyType

This method returns the property type of the item provided in the `property` parameter. Note that this can include objects if a single object instance is defined as a property to another object. To avoid confusion, a collection property containing objects will return the collection property type, not object.

The types returned by this method will be one of the values in the enumerated list `AgentryActiveXPropertyType`. See the description of this enumerated list for details of the values it defines.

Prototype

```
void GetPropertyType
    (VARIANT const property, enum AgentryActiveXPropertyType* type)
```

Parameters

*   `property` - The property for which the type is to be determined.

- `type` - The enumerated value of the property type, as defined in the `ActiveXPropertyType` enumerated list.

Return Value
None *(see `type` parameter description for method return)*

PropertyAsString

This method takes the `property` parameter and returns the value of the property it references as a string. This string is assigned to the `value` parameter. Providing an invalid property type, including an object or collection property, returns an empty string in the `value` parameter.

The `GetPropertyFromMappings` or `GetPropertyFromObject` methods must be called prior to calling `PropertyAsString`. The `property` parameter set by the `GetPropertyFromMappings` or `GetPropertyFromObject` methods should be passed as the `property` parameter to `PropertyAsString`.

Prototype

```
void PropertyAsString (VARIANT const property, BSTR* value)
```

Parameters

- `property` - The property for which the value is to be retrieved.
- `value` - The string representation of the property value.

Return Value
None *(see the `value` parameter description for this function's return value)*

NextCollectionProperty

This method returns the next member of the collection property referenced in the `collection` parameter. The method also updates the position pointer within the `collection` parameter to point to the next member of the collection property. The returned member is referenced in the `property` parameter to this method. If there is no next member, the type for this property is set to `AXPT_Invalid`. This can be checked using the `GetPropertyType` method, passing the `property` parameter to it.

The `GetMappedProperty` method should be called prior to this method. When the property returned in the property parameter is a collection, `NextCollectionProperty` is called to retrieve the members of that collection. When these members are object instances, the `GetPropertyFromObject` method is called subsequent to `NextCollectionProperty` to retrieve the property values found within the object instance.

Prototype

```
BOOL NextCollectionProperty (VARIANT* collection, VARIANT*
property)
```

Parameters

- `collection` - This parameter is the reference to the property returned by the `GetMappedProperty` method when the property it returns is a collection.
- `property` - The next member of the collection property referenced by collection. This property type should always be checked with the `GetPropertyType` method before attempting to reference it to ensure it is not set to `AXPT_Invalid`. This type is returned when there is no next member in the collection.

Return Value
Boolean value indicating whether the position pointer references a valid collection member, or the end of the collection:

- `true` - Returned if the position pointer of the collection parameter points to a valid member of the collection.
- `false` - Returned if the position pointer of the collection parameter indicates the end of the collection has been reached.

CollectionHasNextProperty

This method returns `true` if the current position pointer within the `collection` parameter is pointing to a valid member of the collection; that is, if a call to `NextCollectionProperty` will return an actual instance from the collection. If the position pointer is at the end of the collection, this method returns `false`.

Prototype

```
BOOL CollectionHasNextProperty (VARIANT const collection)
```

Parameters

- `collection` - The collection property to be evaluated for a valid next member based on the collection's position pointer.

Return Value
Boolean value indicating whether the position pointer references a valid collection member, or the end of the collection:

- `true` - Returned if the position pointer of the collection parameter points to a valid member of the collection.

- `false` - Returned if the position pointer of the collection parameter indicates the end of the collection has been reached.

RewindCollection

This method resets the `collection` parameter's internal position pointer to the first member of the collection. The `NextCollectionProperty` method will return the first member of the collection property in the next subsequent call.

Prototype

```
void RewindCollection (VARIANT* collection)
```

Parameters

- `collection` - The collection whose internal position pointer should be reset to the first member of that collection.

Return Value
None

GetAgentryString

Note: This method has been deprecated with the 5.2.8 service pack release of the Agentry Mobile Platform. The method `ProperyAsString` should be used in its place. This method is supported for backwards compatibility only. New development should use the `PropertyAsString` method in all cases, as `GetAgentryString` may be removed at a future time.

This method can be called to access a value on the Agentry Client. This value must be defined as an Agentry Data item in the External Field - ActiveX Control detail screen field. The name of the value, as defined in the Agentry Data list, is passed to the methods `DataItem` parameter. The value of that item is returned in the `agentryString` parameter as a string value regardless of the property's data type within the mobile application.

Prototype

```
HRESULT GetAgentryString (BSTR DataItem, BSTR agentryString)
```

Return Value
The `HRESULT` return indicates the status of the method call.

Enumerated List: AgentryActiveXPropertyType

The following list contains the members of the enumerated list `AgentryActiveXPropertyType`, along with the corresponding property type in the

Agentry application project. One of these values is returned by the `GetPropertyType` method within the Agentry Client ActiveX API indicating the data type of the referenced property.

AgentryActiveXPropertyType **Members**

- `AXPT_Invalid` - Returned when the property parameter does not reference a valid property
- `AXPT_Collection` - Property is a collection property
- `AXPT_ComplexTableSelection` - Property is a complex table selection
- `AXPT_Boolean` - Property is a Boolean
- `AXPT_DataTableSelection` - Property is a data table selection
- `AXPT_Date` - Property is a date
- `AXPT_DateAndTime` - Property is a date and time
- `AXPT_DecimalNumber` - Property is a decimal
- `AXPT_Duration` - Property is a duration
- `AXPT_ExternalData` - Property is an external data
- `AXPT_Identifier` - Property is an identifier
- `AXPT_Image` - Property is an image
- `AXPT_IntegerNumber` - Property is an integral number
- `AXPT_Location` - Property is a GPS location
- `AXPT_Object` - Property is an object instance
- `AXPT_Signature` - Property is a signature
- `AXPT_String` - Property is a string
- `AXPT_Time` - Property is a time

Expected Methods Implemented in ActiveX Control

In order for the Agentry Client to interface with an ActiveX control, it is a requirement of that control that it implements certain methods with the proper prototypes. Following is a description of each of these methods, their prototypes, and when the method is called by the Agentry Client at run time.

It is important that each of these methods is implemented, even those that are provided for functionality not currently implemented. Those methods not expected to be used should include at least a stub implementation within the ActiveX control.

Currently the Agentry Client can directly integrate with ActiveX controls built using C++ and Visual Basic. Included in the following sections are two lists of method prototypes for each of the expected methods. The first is for C++ implementations, and second is for Visual Basic.

Note: See the section at the end of this technical bulletin on integrating ActiveX controls built on `.NET`.

ActiveX Expected Method Declarations - eMbedded Visual C++

When using the Visual C++ wizard to add methods, create the methods with the parameter and return types exactly the same as shown below. The method declarations in the control class header should appear identical to the following, with the exception of any word wrapping resulting from this publication.

```
afx_msg BOOL AgentryInitialize(LPCTSTR initialValue, LPCTSTR
formatString,
    BOOL readOnly, BOOL autoChangeFocus, long parentHwnd, VARIANT
messageIDs);
afx_msg void AgentryDestroy();
afx_msg void AgentryEnable(BOOL state);
afx_msg BSTR AgentryGetValue();
afx_msg void AgentrySetFocus(long type);
afx_msg void AgentryShow(BOOL state);
afx_msg void AgentryUpdateRuleEvaluated(LPCTSTR ruleResult);
afx_msg void AgentryUpdateScanData(BSTR scanResult);
afx_msg BSTR AgentryGetSpecificValue(LONG opcode, VARIANT
specificValue);
afx_msg BSTR AgentryGetScriptValue();
afx_msg void AgentrySetScriptValue(BSTR str);
afx_msg void AgentrySetActiveXControlHost(IUnknown* host);
afx_msg LONG AgentryExecuteCommand(LPCTSTR str);
```

The methods section of the IDL file for the Agentry ActiveX interface should appear identical to the following:

```
methods:
[id(1)] boolean AgentryInitialize(BSTR initialValue, BSTR
formatString,
    boolean readOnly, boolean autoChangeFocus, long parentHwnd,
    VARIANT messageIDs);
[id(2)] void AgentryDestroy();
[id(3)] void AgentryEnable(boolean state);
[id(4)] BSTR AgentryGetValue();
[id(5)] void AgentrySetFocus(long type);
[id(6)] void AgentryShow(boolean state);
[id(7)] void AgentryUpdateRuleEvaluated(BSTR ruleResult);
[id(8)] void AgentryUpdateScanData(BSTR scanResult);
[id(9)] BSTR AgentryGetSpecificValue(LONG opcode, VARIANT
specificValue);
[id(10)] BSTR AgentryGetScriptValue();
[id(11)] void AgentrySetScriptValue(BSTR str);
[id(12)] void AgentrySetActiveXControlHost(IUnknown* host);
[id(13)] LONG AgentryExecuteCommand(BSTR str)
```

ActiveX Expected Method Declarations - MS Visual Basic

The methods expected by the Agentry Client should be declared exactly as listed below, with the exception of any word wrapping resulting from this publication.

```
Public Function AgentryInitialize(initialValue As String,
    formatString As String, readOnly As Boolean, autoChangeFocus As
Boolean,
   parentHwnd As Long, VARIANT messageIDs) As BooleanPublic Function
AgentryDestroy()
Public Function AgentryEnable(state As Boolean)
Public Function AgentryGetValue() As String
Public Function AgentrySetFocus(focusType As Long)
Public Function AgentryShow(state As Boolean)
Public Function AgentryUpdateRuleEvaluated(ruleResult As String)
Public Function AgentryUpdateScanData(scanResult As String);
Public Function AgentryGetSpecificValue(opcode As Long,
    VARIANT specificValue);Public Function AgentryGetScriptValue();
Public Function AgentrySetScriptValue(str As String);
Public Function AgentrySetActiveXControlHost(IUnknown* host);
```

AgentryInitialize

This method initializes the ActiveX control. It is called by the Agentry Client immediately after the External Field - ActiveX Control detail screen field is created. If this method returns `false`, indicating the control failed to initialize, the Agentry Client will not display the ActiveX control.

Parameters

- `initialValue` - The value of the property targeted by the External Field - ActiveX Control field in the Agentry Client.
- `formatString` - The value of the **Format** attribute defined in the External Field - ActiveX Control field in the Agentry Client.
- `readOnly` - The value of the **Read Only** attribute defined in the External Field - ActiveX Control field in the Agentry Client. `true` indicates the field is defined to be read-only.
- `autoChangeFocus` - The value of the **Automatically change focus to next control** attribute of the External Field - ActiveX Control field in the Agentry Client. `true` indicates this attribute has been set.
- `parentHwnd` - The HWND that corresponds to the parent window of the ActiveX control The ActiveX control should use this to send messages to the Agentry Client.
- `messageIDs` - **NOTE:** *This value, while still provided, should be considered deprecated See the Agentry Client ActiveX API methods* `ActiveXControlValueChanged` *and* `ActiveXControlValueEntered` *for the current manner of performing these operations.* A safe array stored within a VARIANT. The safe array contains an array of long values that correspond to each message ID for each message that may be sent to the Agentry Client. Within C++ the array index begins at zero and within the Vidual Basic the array index begins with one.
  - *First index position:* Send this message to the `parentHwnd` to notify the Agentry Client a value has changed within the control and it is time for the Agentry Client to

evaluate the field update rules and enable rules defined for all fields on the current detail screen.

- *Second index position:* Sends this message to the `parentHwnd` to notify the Agentry Client a value has been completely entered in the control and it is time to automatically change focus to the next control.

Return Value

- `true` - This method should be implemented to return `true` when the ActiveX control has been successfully initialized.
- `false` - This method should be implemented to return `false` when the ActiveX control has failed to initialize. The Agentry Client will not display the control on the screen and will not call any other methods within the ActiveX control.

AgentrySetActiveXControlHost

This method provides the pointer to the `IAgentryActiveXControlHost` object to the ActiveX control. This pointer is passed over as an `IUknown` pointer for the control host interface and should be queried to obtain the `IAgentryActiveXControlHost` object. This object provides the interface to the Agentry Client. It contains the methods that make up the Agentry Client-Side ActiveX API.

Parameters

- `host` - `IUknown` pointer to the `IAgentryActiveXControlHost` object. Query this pointer to obtain the control host object.

Return Value
This method should be implemented with a void return.

AgentryDestroy

This method is called by the Agentry Client just before the External Field - ActiveX Control detail screen field is destroyed. This method should be implemented to perform any cleanup that may be necessary before the control is deleted.

Parameters
None

AgentryGetValue

This method is called by the Agentry Client to retrieve the current value of the ActiveX Control. This method is called by the Agentry Client either when the user advances past the screen displaying the External Field - ActiveX Control field in order to obtain the value to set to the target property of the field definition; or when a rule is evaluated by the Agentry Client that references the External Field - ActiveX Control.

Parameters
None

Return Value

- BSTR - String to be returned to the Agentry Client as the ActiveX control's value.

AgentrySetFocus

This method is called by the Agentry Client when the focus is set to the External Field - ActiveX Control detail screen field due to one of the following events:

- **Auto Focus:** The parent screen has just been displayed, that screen's **Focus Field** attribute is defined as Auto, and the External Field - ActiveX Control detail screen field is in the position to receive the focus.
- **Initial Focus:** The parent screen has just been displayed and that screen's **Focus Field** attribute is defined to set the focus explicitly to the External Field - ActiveX Control detail screen field.
- **Auto Change Focus:** The previous detail screen field is defined to automatically change focus to the next field, the External Field - ActiveX Control field is the next field, and the user has just entered a value in the previous field.
- **OS Focus:** The OS has sent a message that the External Field - ActiveX Control should receive the focus. This can occur when the user tabs to the field, selects the field's hot key, and other similar situations.

The value of the type parameter to this method indicates which of the above is the reason for the External Field - ActiveX Control to have received the focus should it be necessary to perform different processing based on the focus event.

Parameters

- type - The value indicating why the field has received the focus. These are numeric values corresponding to one of the above described events:
  - **Auto Focus:** 1
  - **Initial Focus:** 2
  - **Auto Change Focus:** 3
  - **OS Focus:** 4

Return Value
None

AgentryGetSpecificValue

This method is called by the Agentry Client to retrieve a value by name from the ActiveX control. The name passed will be one of those values contained in the External Field - Active X

Control definition's External Data list. This method should be implemented to receive any of the names as defined in the field definition, and to return the appropriate string value represented by that value name.

The External Data values listed in the External Field - ActiveX Control field's definition are available for reference in target paths within the Agentry application project. This method is called by the Agentry Client whenever one of these values is so referenced.

Parameters

- `opcode` - deprecated value that should be a part of the method's prototype but not used within the method's implementation.
- `specificValue` - The string name of the value to be returned by the method, as listed in the External Field - ActiveX Control's External Data list.

Return Value

- BSTR - The string representation of the named value requested by the Agentry Client.

AgentryUpdateScanData

This method is called by the Agentry Client immediately after a value has been scanned in by the client device for the External Field - ActiveX Control screen field. This method passes the scanned value to the ActiveX control in the `scanResult` parameter.

Parameters

- `scanResult` - This is the string value of the barcode value scanned in on the client device for the External Field - ActiveX Control screen field.

AgentryEnable

This method is called by the Agentry Client immediately after the enable rule for the External Field - ActiveX Control field has been evaluated. The state parameter to this method accepts the result of the enable rule's evaluation, which is a Boolean value. This method should be implemented to perform whatever processing may be necessary when the field is enabled and when it is disabled.

Parameters

- `state` - This parameter contains the Boolean value of the enable rule's return. `true` indicates the field is enabled, `false` indicates it has been disabled.

AgentryShow

This method is called by the Agentry Client immediately after the Hidden Rule is evaluated for the External Field - ActiveX Control field. This rule returns a Boolean value indicating

whether or not the field should be displayed or hidden on the screen. The `state` parameter to this method indicates whether or not the field is shown.

Note that the Hidden Rule evaluated by the Agentry Client returns `true` when the field should be hidden, and `false` when it should be displayed. The value passed to the `AgentryShow` method is the inverse of the rule's return, meaning a `state` parameter value of `true` indicates the field is displayed, and `false` indicates it is hidden on the Client.

Parameters
- `state` - This parameter indicates whether the field is shown on the client screen. `true` indicates the field is currently displayed, `false` indicates it has been hidden.

AgentryUpdateRuleEvaluated

This method is called by the Agentry Client immediately after the update rule for the External Field - ActiveX Control detail screen field has been evaluated. The return from the field's update rule is passed to this method in the `ruleResult` parameter. The `AgentryUpdateRuleEvaluated` method should be implemented to process this value as the one currently displayed in the field on the client.

Parameters
- `ruleResult` - The string result of the External Field - ActiveX Control's update rule evaluation.

AgentryGetScriptValue

This method is provided exclusively for support of the Agentry Test Script functionality available in the Agentry Test Environment. This method is called by the Agentry Test Script Recorder when a `<field-expect>` method is recorded for the test script. It is also called during script playback when the `<field-expect>` element is processed for an External Field - ActiveX Control detail screen field. The method takes no parameters and is expected to return the value of the ActiveX control to be evaluated by the `<field-expect>` element.

Parameters
None

Return Value
The value of the ActiveX control as a string (`BSTR`) to be provided to the test script currently being recorded or executed by the Agentry Test Environment.

AgentrySetScriptValue

This method is provided exclusively to support the Agentry Test Script functionality available within the Agentry Test Environment. This method is called by the Agentry Test Environment during script playback when a `<field-set>` element is executed. The value for the

ActiveX control is passed to this method's `str` parameter. The
`AgentrySetScriptValue` method should be implemented to process this parameter
value such that it is set as the current value of the control as entered by a user.

Parameters

• `str` - The value to be set as the current value of the ActiveX control, provided as a string.

Return Value
None

Agentry Client API for External Processes Technical Overview

The Agentry Client API for external processes provides four methods that may be called by an
external process to request information and data from, and to invoke transactions and execute
actions on the Agentry Client. In order to use this API the external process must be built using
the resources provided by the Agentry Client SDK for this API. The resources provided were
built and are maintained using Visual Studio 2008 in the Visual C++ language. These same
tools must be used to build the external process that is to make calls into the API.

The Agentry Client API for external processes does not include any corresponding controls or
other similar components within the Agentry Client. It is limited to the methods made
available to external processes to call.

Each of these methods includes a parameter containing the Agentry Client context object.
This object is provided by the `AgentryInitialilze` method, which must be called prior
to calling any of the other methods, and this object is then passed to each of the other methods
when called.

Retrieving Data from the Agentry Client
Data is returned to the external process via the `EvaluateAgentryRule` method within the
Agentry Client API. The rule to be evaluated and the module in which it has been defined are
passed to the method parameters, along with a string variable in which the return value of the
rule is provided. Using this method rules can be called within the Agentry Client from the
external process to retrieve values from the Client. The returned values can be calculated or
conditional values based on the structure of the rule definition, or they can simply be property
values or other similar data items, again based on the rule structure.

Executing Actions on the Agentry Client
Actions can be executed by the external process via the `ExecuteAgentryAction` method
within the Agentry Client API. The action to be executed and the module in which it has been
defined are passed to the method. The method returns a Boolean indicator of success or failure
to execute the action.

Transaction Processing on the Agentry Client from External Processes

Edit transactions can be instantiated, properties within them populated with values, and subsequently applied on the Agentry Client as a result of a request from an external process via the `ExecuteAgentryTransaction` method. The edit transaction to be processed, the module in which it is defined, and values for one or more of its properties are passed to the method as parameters. Not all properties within the transaction need to be populated by the method call, and any not provided are initialized according to the property definitions just as if the transaction were instantiated via standard Agentry Client processing.

AgentryInitialize

The `AgentryInitialize` method is called to initialize a pointer to an `AgentryClientContext` object. This pointer is a required parameter to all other methods within the Agentry Client API for external processes. The single parameter to this method is a pointer to an object pointer of type `AgentryClientContext`. The pointer should be declared prior to calling the method and initialized to NULL. The address of the `AgentryClientContext` object is provided with the call to the `AgentryInitialize` method.

This method must be called when the external process is executed. The handle it returns (`AgentryClientContext` object pointer) should be preserved and passed to any subsequent Agentry Client API for external processes method calls. This handle should be passed to the `AgentryUnInitialize` method as a part of the external processes's shutdown procedures. See information provided on this method for details.

Prototype

```
bool AgentryInitialize(AgentryClientContext** ppCtx)
```

Parameters
- `ppCtx` - The `AgentryClientContext` object initialized by this method and passed to all other method calls within this API. A NULL pointer should initially be created to such an object, and the pointer should then be passed to this method, as in:

```
AgentryClientContext *ctx = NULL;
AgentryInitialize(&ctx);
```

Return Value

The Boolean return value indicates whether or not the `AgentryClientContext` object was successfully initialized. The method returns false in the event of failure, which can occur if the Agentry Client is not currently running, as well as under other conditions. The return from this method should always be checked prior to using the `AgentryClientContext` object pointer it initializes. The external process should include processing to account for a false return indicating a failed initialization of this handle.

AgentryUnInitialize

the AgentryUnInitialize method is provided to allow the
AgentryClientContext handle object to be properly cleaned up when the external
process is exiting. this method should be called as a part of the processes's shutdown routines.
It's only parameter is the handle, which should be the same as the one passed to a previous call
to the AgentryInitialize method.

Prototype

```
bool AgentryUnInitialize(AgentryClientContext* pCtx)
```

Parameters

- AgentryClientContext - This parameter is passed to the method so that the handle
  for the AgentryClientContext object can be properly cleaned up when the external
  process is shutting down and the handle is no longer needed.

Return Value
The Boolean return value from the method indicates the success or failure of the uninitialize
processing.

EvaluateAgentryRule

The EvaluateAgentryRule method can be called by the external process to request a
named rule be evaluated by the Agentry Client. Included in the parameters to this method are
the internal names of the module in which the rule is contained and the name of the rule to be
evaluated. Also included are the AgentryContext and a string parameter in which the
return value of the rule will be captured.

Rules evaluated by the EvaluateAgentryRule method are evaluated in the context of the
module MainObject of the same module in which the rule is defined. Any rule in the module
may be evaluated via this method, with the rule's return value provided as a string. This value
can then be converted to other data types as needed within the external process.

Prototype

```
bool EvaluateAgentryRule(AgentryContext* pCtx,
                                    const std::tstring& ModuleName,
                                     const std::tstring& RuleName,
                                        std::tstring& Value)
```

Parameters

- pCtx - Pointer to the AgentryContext object returned by a call made to the AgentryInitialize() method.
- ModuleName - The name of the module definition within the Agentry application project in which the rule to be evaluated is defined.
- RuleName - The name of the rule to be evaluated and whose return value is to be captured in the Value parameter.
- Value - Reference to a string value within the external process in which the return value of the rule definition will be contained. Regardless of the rule context or structure, the return value is always provided as a string value and can be cast to other data types within the external process.

Return Value

The Boolean return of this method indicates whether or not the rule was found and evaluated. If this fails for any reason the function returns false and the value of the Value parameter is a null string. The return value should always be checked before attempting to use the Value parameter and the external process should include logic to account for a failed rule evaluation.

ExecuteAgentryAction

The ExecuteAgentryAction method is called to request the Agentry Client execute an action. In addition to the AgentryConext, the method takes parameters specifying the name of the module in which the action to be executed is defined, as well as the name of the action itself. The action is always executed in the context of the module MainObject. The action being executed, therefore, must be defined for the MainObject or for no object. SubAction steps executing actions for other objects can be defined within the action executed by the method should it be necessary to execute an action for a different object type.

Actions may not be executed immediately under certain conditions; specifically, if another action is currently being executed. In such cases the action is queued by the Agentry Client to be executed as soon as it is able. The method will return true in such a case and the action will be executed when the first opportunity arises. The Agentry Client contains only a single action queue in which all queued actions are stored until executed. The other primary situation in which actions can be queued relates to Push Actions. Actions are executed from this queue in a first in-first out order.

Actions will not be executed and the method will return false if the Agentry Client is currently running, but the user has not yet completed the login process successfully, e.g., the login screen is currently displayed, the server selection screen is displayed, etc.; or if the named action or module cannot be found within the business logic currently running on the Agentry Client.

Prototype

```
bool
ExecuteAgentryAction(                                               Age
ntryContext* pCtx,
                                        const std::tstring& ModuleName,
                                            const std::tstring&
ActionName)
```

Parameters

- `pCtx` - Pointer to the `AgentryContext` object returned by a call made to the `AgentryInitialize()` method.
- `ModuleName` - The name of the module definition within the Agentry application project in which the action to be executed is defined.
- `ActionName` - The name of the action to be executed by the Agentry Client.

Return Value

The Boolean return of this method indicates whether or not the action was found and either executed or placed in the pending actions queue to be executed when possible. If this fails for any reason the function returns false and the named action will not be executed by the Agentry Client. The return value should always be checked and the external process should include logic to account for a failed action execution.

ExecuteAgentryTransaction

The `ExecuteAgentryTransaction` method is called by the external process to request an edit transaction be instantiated and applied, i.e., to be processed, by the Agentry Client. In addition to the `AgentryContext` object, the method takes the name of the module in which the transaction is defined, the name of the transaction itself, and a reference to an `AgentryPropertyVector` containing the transaction property values to be set within the transaction.

The transaction to be processed must be an edit transaction and must be defined for the module MainObject, as it is instantiated in the context of that object. Add and delete transactions are not supported.

When a transaction is processed via as a result of a call to this method, the properties of that transaction are first initialized according to the initial value attributes of those properties, with the exception of "Rule - After Data Entry." Next, any values passed to the method call are copied to the transaction properties, which will replace any initialization values that may be present. The transaction is then processed by the Agentry Client. Property values are then set for any properties which are initialized to "Rule - After data entry." The current value of such properties are overwritten with the value returned by the rule. Finally the transaction is applied, which includes setting the values of the object properties targeted by the transaction properties, and the transaction itself is saved to the client device as a pending transaction.

Prototype

```
bool
ExecuteAgentryTransaction(
        AgentryContext* pCtx,
                                            const std::tstring&
ModuleName,
                                            const std::tstring&
TransactioName,
                                                const
AgentryPropertyVector& properties)
```

Parameters

- `pCtx` - Pointer to the `AgentryContext` object returned by a call made to the `AgentryInitialize()` method.
- `ModuleName` - The name of the module definition within the Agentry application project in which the transaction to be processed is defined.
- `TransactionName` - The name of the transaction to be processed by the Agentry Client.
- `properties` - Reference to an `AgentryPropertyVector` containing the property values to be set when the transaction is instantiated. See the section on the API data types for more details on `AgentryPropertyVectors`.

Return Value
The Boolean return of this method indicates whether or not the transaction was found and processed. If this fails for any reason the function returns false and the named transaction will not be processed by the Agentry Client. The return value should always be checked and the external process should include logic to account for failed transaction processing.

Data Types Defined in the Agentry Client API for External Processes

Within the Agentry Client API for external processes there are certain data types defined: `AgentryContext`, which is an object obtained using the `AgentryInitialize()` method, and `AgentryPropertiesVector`, which is established via a type definition as a vector of `AgentryAttrPair` items. `AgentryAttrPair` is a standard pair of strings.

AgentryAttrPair **and** *AgentryPropertiesVector*
The `AgentryPropertiesVector` is provided to allow for property values of a transaction to be set by the external process and passed to the Agentry Client via the `ExecuteAgentryTransaction` method. This data type is declared in the include file `AgentryExternal.h`, which should be included in the project containing the external process logic.

This data type is declared by the following `typedef` statements:

```
typedef std::pair<std::tstring, std::tstring> AgentryAttrPair
typedef std::vector<AgentryAttrPair> AgentryPropertiesVector
```

The first `typedef` statement creates a standard pair of string values identified as `AgentryAttrPair`. This type is then the member type for the vector declared by the second statement, which is identified as the type `AgentryPropertiesVector`.

Within the elements of an `AgentryAttrPair` are stored the name and value of a property within the transaction definition, with the first element of the pair containing the property definition name, and the second containing the value. All values are stored as strings within a given pair and the second element is converted, when necessary, by the Agentry Client to the property data type before assigning the value to the specified property within the transaction. This behavior negates the need to perform any data type conversion within the external process as it would relate to property data types.

AgentryClientContext
This object type is internal to the Agentry Client. A declaration is provided for this object in the `AgentryExternal.h` header file. A handle to this object is provided by the `AgentryInitialize` method, which should be called by the external process during startup. The handle is then a required parameter to all API method calls. The handle should be passed to the `AgentryUnInitialize` method by the external process during shutdown.