Business Object Developer Guide

Versata Logic Suite - WebSphere 4.0 Edition

VERSATA, INC.
300 LAKESIDE DRIVE
SUITE 1500
OAKLAND, CA 94612-3534
PHONE: 510.238.4100
INTERNET: HTTP://WWW.VERSATA.COM
VS55E40-BDG-03

Copyright

Table of Contents

Preface

Preface

Versata Logic Suite documentation

The Versata Logic Suite documentation is electronically provided in `.pdf` and `.hlp` file formats during installation of the system. Review the following sections for documentation file descriptions, installation locations, and viewing instructions.

Versata Logic Suite Library

The Versata Logic Suite Library consists of `.pdf` (portable document format) files, an .hlp file, a chm file, and a readme.txt file. These files are automatically installed in the `\Help` subfolder of the default directory during installation.

The ***Versata Logic Suite Library*** (`Library.pdf`) is the main page Provides links to all of the `.pdf` manuals, hlp file, chm file, readme, and full-text search of all of the `.pdf` manuals.

To launch `Library.pdf` after installing Versata Logic Suite and Acrobat Reader:

On the desktop, click the Start button → Programs → Versata Logic Suite 5.5 <edition_Name> → Versata Logic Suite Library.

Note: Each `.pdf` file should be viewed, searched, and printed using the 4.05 version of Adobe® Acrobat® Reader with Search to ensure that the full-text search feature functions correctly and that graphics display properly.

This software is available for installation from the main Versata installation screen, or you may download it at `www.adobe.com`.

Note: This version of the Versata Logic Suite allows integration of transaction logic and process logic in your business objects. The integration features and documentation for those features is only available if you have purchased the Process Logic Add-On.

Versata Logic Suite Library PDF Manuals

The following `.pdf` files comprise the Versata Logic Suite Library:

- ***Getting Started Guide*** (`GettingStarted.pdf`). Provides basic installation and configuration steps for the Versata Logic Studio, Versata Logic Server, and other products needed to run the Versata Logic Suite.
- ***Tutorial*** (`Tutorial.pdf`). Steps you through features of the Versata Logic Suite. It also describes Java and HTML sample applications and shows you how to create your own Java and HTML applications (with presentation design only).
- ***Architecture and Project Guide*** (`Architecture&ProjectGuide.pdf`). Introduces the system architecture, project development process, and team development functionality of the Versata Logic Suite. This guide also contains a glossary of Versata Logic Suite, Java™, and database terms.

- ***Business Object Developer Guide*** (`BusinessObjectDeveloperGuide.pdf`). Describes how to use the Versata Logic Studio to design a data model and transaction logic for applications. Sections of this manual explain data object and query object definition, business rules development, data model and transaction logic deployment, and rules testing.
- ***Application Developer Guide*** (`ApplicationDeveloperGuide.pdf`). Describes how to use the Versata Logic Studio to create the user interface for applications (with presentation design only). Sections of the manual explain HTML and Java application development, application deployment, application testing, and application delivery.
- ***Administrator Guide*** (`AdministratorGuide.pdf`). Describes how to administer deployed objects and define security in the Versata Logic Server through the Versata Logic Server Console and server code.
- ***Reference Guide*** (`ReferenceGuide.pdf`). Contains reference information, including the Versata Logic Studio user interface help, a high-level summary of system class libraries, details about repository `.xml` and `.dtd` files, and a glossary of terms.
- ***Migration Guide*** (`MigrationGuide.pdf`). Provides guidelines for upgrading to release 5.5 of Versata Logic Suite from a previous version.
- ***PDX Guide*** (`PDXGuide.pdf`). Describes how to use the user interface development features included in PDX. These features have now been integrated into the core Versata Logic Suite product.
- ***Using PDX Frameless Archetypes*** (`Using PDX Frameless Archetypes.pdf`). Describes how to use the Frameless Archetypes feature included in PDX. This feature has now been integrated into the core Versata Logic Suite product.

Versata Logic Suite User Interface Help

The Versata Logic Suite User Interface Help is provided in a Microsoft HTML help file called `vstudio.chm`. This help file provides context-sensitive help with detailed descriptions of the frames and fields in the Versata Logic Studio.

To launch `vstudio.chm`*:*

1. Focus on a window or frame in the Versata Logic Studio and press F1 to launch a context-sensitive help topic for that element.

   OR

1. Choose Help → Versata Logic Suite Library in the Versata Logic Studio.
2. In the Versata Logic Suite Library, click the Versata User Interface Help link.
3. Click yes when prompted to open the file.

OR

1. Choose Start → Programs → Versata Logic Suite 5.5 <edition_name> → Versata Logic Suite Library.
2. In the Versata Logic Suite Library, click the Versata User Interface Help link.
3. Click yes when prompted to open the file.

Versata Class Libraries Help

The Versata Class Libraries are provided in a WinHelp file called `vstudio.hlp`. This `.hlp` file describes all of the classes and methods included in the Versata Logic Suite packages.

To launch `vstudio.hlp`*:*

1. Focus on a class name, method, or a string in the Code Editor in the Versata Logic Studio and press F1 to launch a context-sensitive help for that element.

OR

1. Choose Help → Versata Logic Suite Library in the Versata Logic Studio.
2. In the Versata Logic Suite Library, click the Versata Class Libraries Help link.
3. Click yes when prompted to open the file.

OR

1. Choose Start → Programs → Versata Logic Suite 5.5 <edition_name> → Versata Class Libraries Help.

Versata Logic Suite Readme

The `readme.txt` file provides latebreaking release notes about the Versata Logic Suite.

To launch `readme.txt`*:*

- During installation of the Versata Logic Studio, click the `Yes` button when prompted to view the readme.
- Choose Start → Programs → Versata Logic Suite 5.5 <edition_name> → Versata Logic Suite Readme.

Conventions for documentation and user interface help

The following conventions are used in the documentation to convey special meaning.

- Code, such as folder names, file names, and example code snippets, is shown in Courier New font, `like this.`
- Brackets, < > around part of a file name, or path, indicate that the information between the brackets should be filled in as appropriate. For example, the default directory path `\Archetypes\<application_name>\<repository_name>` indicates that text between the brackets depends on the name of the currently opened application and repository. Note that these are different from the angle brackets that appear around tags and macro code.
- Toolbar menu options in procedures are shown in this format: On the desktop, click the Start button → Programs → Versata Logic Suite 5.5 → Versata Logic Server Console. The first option is the top menu in the hierarchy. Succeeding options progress to submenus.
- Menu commands and tab names are shown with their full paths:
  - **Menus**. From the File menu, choose New → Repository.
  - **Tabs**. This option is set on the Properties: Attributes tab of the Transaction Logic Designer.
- A Caution or Warning is important advice that you should read carefully.

The Versata User Interface help uses these additional conventions.

- Click F1 while focus is on a dialog, window, menu, or toolbar in the Versata Logic Studio to launch context-sensitive help.
- Contents, Index, and Find tabs are provided on the left pane of the Help window when you launch the Versata Logic Suite help file.
  - Click the Contents tab to look in the table of contents. It lists the modules and main sequences of the help.
  - Click the Index tab to search the index.
  - Click the Find tab to search the text of the entire help system.
- Browse buttons (Back and Forward) are provided in the Help toolbar, and Previous and Next hyperlinks are provided in the Help topics. Use these options to scroll through sequences of related topics.
- To print a help topic, click the Print button at the top of the Help window. To print an entire book of help topics, select the book on the Contents tab and click the Print button.

Additional documentation

IBM WebSphere™ Application Server documentation

For information about features specific to the IBM WebSphere™ Application Server, consult the documentation at the following Web sites:

- To access the IBM Documentation Center, visit the following Web site:
  `<as_root>\web\doc\begin_here\index.html`
- For the latest corrections and additions to this information, consult the IBM WebSphere™ Application Server Web site. To view the latest Release Notes, visit the Library page of the Web site: `http://www.ibm.com/software/webservers/appserv/`
- For instructions to help you enable debugging, tracing, logging, and monitoring to detect and diagnose problems in both the IBM WebSphere™ Application Server and your own programs, refer to the WebSphere™ online help and the FAQ at `http://www.ibm.com/software/webservers/appserv/library.html`
- For information about the IBM HTTP Server, visit the following Web site:
  `http://www.ibm.com/software/webservers/`
- For more information on using and configuring DB2, visit the following site:
  `http://www.software.ibm.com/cgi-bin/db2www/library/pubs.d2w/report#UDBPUBS`

To correctly display the preceding documentation, you need Netscape Navigator 4.07 or Microsoft Internet Explorer 4.01 or higher.

Versata Logic Suite resources

The following resources are available to help you learn more about the Versata Logic Suite.

Sample database and sample applications

The Versata Logic Studio includes a sample database with examples of business requirements, functions, and rules. Extensive sample applications are provided to illustrate features of the HTML and Java applications generated by the Versata Logic Suite (with presentation design only). These sample applications include example code for you to use to implement complex features more easily.

The samples and sample database are located in the `Samples` directory where you install the Versata Logic Suite. The `vsamples.hlp` file, located in the `Help` directory, provides detailed descriptions of the sample database, rules examples, and sample applications (with presentation design only). In addition, the most recent description of each sample application is located in the `About_*.app.rtf` file in each sample application folder in the Versata Logic Studio Explorer.

To access the Versata Logic Suite sample applications (with presentation design only):

1. Launch the Versata Logic Studio.
2. Open `sampDB1.xml` (the sample database) as your repository.
3. Expand the `Client Applications` folder in the Versata Logic Studio Explorer.
4. Select a particular sample application and run it.
5. Review the `About_*.app.rtf` file in that sample application folder (Files tab) or choose Help → Samples to launch `vsamples.hlp` for information about that sample application.

Versata Web site

Browse the Versata Web site at `www.versata.com` for the latest information about:

- Versata Logic Suite products, upgrades, and demos
- Sales
- Employment opportunities
- Training
- Professional Services

Versata Knowledge Base

The Versata Knowledge Base is available to help with your technical questions about the Versata Logic Suite. You can search through our growing library of technical articles or participate in our online Developer Discussions forum.

To access the Versata Knowledge Base:

1. Visit the Versata Web site at http://www.versata.com.
2. Click the Training and Support tab, then select Versata Knowledge Base.

Versata Developer Discussions

Access Versata Developer Discussions on the Versata Web site. Sign up to view the postings and subscribe to the mailing list to receive the latest news about the Versata Logic Suite automatically. These technical discussions provide a forum for customers, partners, distributors, and Versata internal employees to post technical and general questions, suggestions, and solutions about development, run-time, and production features of the Versata Logic Suite.

To access the Versata Knowledge Base:

1. Visit the Versata Web site at http://www.versata.com.
2. Click the Training and Support tab and select Developer Discussion.

Versata Customer Support

You may use any of the following methods to contact Versata Customer Support.

- **Internet.** At the Versata website (`www.versata.com`), click on the Training and Support tab to find information about Versata Customer Support.
- **Phone.** 510.238.4100. Between 7:00am and 5:30pm, Pacific Time, Monday-Friday
- **E-mail.**
  - For software issues: `techsupport@versata.com`
  - For documentation issues: `docs@versata.com`

Technical support for IBM WebSphere Application Server

If you experience a problem that is specifically related to the IBM WebSphere™ Application Server, call:

- Your IBM systems integration consultant, if your implementation is being assisted by IBM Global Services
- IBM Software Service Support: 1-800-237-5511

To learn more about IBM Software Support, see the IBM support page at:
`http://www.ibm.com/Support`

You can also e-mail IBM directly with your suggestions and requirements for future releases. Report noncritical defects that do not require a personal interaction or formal support to:

`WASTEAM@US.IBM.COM`.

CHAPTER 1

Introduction

Overview

This guide explains how you can use the Versata Logic Suite to develop business objects that execute transaction logic for run-time Java and HTML enterprise applications. Versata Logic Suite business objects run on Versata Logic Server integrated with a J2EE application server. Business objects' transaction logic can be used with application user interfaces created in the Versata Logic Studio or in a JSP development environment.

This guide walks you through the steps needed to create a Versata repository data model that maps to an external data source, and to define business rules on data model objects that implement transaction logic on the Versata Logic Server. Tasks covered include how to reengineer from and deploy to external data sources, how to modify data object, attribute, and relationship characteristics in the Versata Logic Studio, how to create and modify query objects, the non-persistent data sources used for display purposes in applications, how to define different types of business rules to execute transaction logic, how to build and compile business rule input into deployable files, how to package and deploy business object files to a Versata Logic Server and J2EE application server, how to extend transaction logic with custom code, and how to implement data source connectivity for objects on Versata Logic Server.

Prerequisites

Before you begin designing transaction logic in the Versata Logic Studio, you should have a good understanding of the following:

- The business requirements for application data. To determine requirements for an application, you need to define the processes to be automated and the data to be stored. Process automation and data storage both affect the business object model and the transaction logic that you need to define in the Versata repository. You need to define the requirements to enforce for processes and data. You can do this through a variety of methods, including use case diagrams and context diagrams.
- The logical data model to be used for repository business objects. You can use standard logic data modeling (LDM) methods to produce a logical business object model. A logical model represents the overall logical structure of the data to be stored, independent of any software or data storage structure. A logical model gives a formal representation of the data needed to run an enterprise or a business activity. Versata does not restrict the tools you can use to produce this logical model. Note that Versata offers a separate product that provides integration with Rational Rose.
- The physical database(s) against which enterprise applications will run. You then can determine the best way to produce this model in a Versata repository.
- System requirements for the project development environment and production environment, including the J2EE application server setup.

You may want to review the *Architecture and Project Guide* and the *Reference Guide* to get an overview of how the Versata Logic Suite works. The *Architecture and Project Guide* provides an introduction to developing transaction logic and applications in the Versata Logic Studio, including how to manage projects and facilitate team development. The *Reference Guide* contains details about the Versata Logic Studio development environment and the `.xml` and `.dtd` files used to store Versata repository source information. This guide also includes a glossary of terms and other general reference information.

If you plan to extend business object transaction logic code, you require knowledge of Java programming concepts, EJB architecture, and J2EE specification requirements. The depth of knowledge required depends on the complexity of the custom code. Also, you should understand the class libraries provided with the Versata Logic Suite. Details about these classes and their methods are available in the Javadoc API help installed with the product.

How to use this guide

This guide includes the following information.

- The Preface describes the documentation accompanying the Versata Logic Suite and points to additional resources, such as the sample database and applications included in the sample repository, SampDB1.xml, that gets installed with the product.
- This chapter outlines the purpose, prerequisites and contents of this guide.
- Chapters 2-5 explain how to produce a data model in a Versata repository and modify it in the Versata Logic Studio. This model represents physically stored data accessed by applications and governed by transaction logic.
  - Chapter 2, “Developing a Data Model” on page 31, provides guidelines for data model objects and attributes, including naming conventions and data type mappings. This chapter describes the alternate methods for creating a repository data model, explains how to validate a data model, discusses the use of groups to divide data model objects into manageable subsets, and explains the use of attribute templates to provide data object inheritance.
  - Chapter 3, “Working with Data Objects” on page 81, provides instructions for creating and modifying data objects in the Versata Logic Studio. Topics covered include coded values lists, attribute persistence, data type, and other properties, relationships, and indexes.
  - Chapter 4, “Deploying Data Models” on page 121, explains how to use the Server Manager wizard to deploy Versata Logic Studio data model definitions to supported RDBMSs.
  - Chapter 5, “Working with Query Objects” on page 145, discusses how and why to use query objects, special reusable presentation objects, to display data in applications.
- Chapters 6-11 and Appendix A discuss how to define rules on business objects to implement transaction logic, and describe business object files’ contents and how to extend them.
  - Chapter 6, “Understanding Transaction Logic” on page 183, provides an overview of the declarative business rules used to implement transaction logic on Versata Logic Server. This chapter includes an outline of transaction logic processing, descriptions of the different types of rules, and a discussion of how to translate requirements into rules.
  - Chapter 7, “Defining Business Rules” on page 211, provides guidelines for defining business rules, outlines the Versata Logic Studio business rule definition process, describes the Transaction Logic Designer, provides instructions for defining specific types of rules, and details rule expression syntax.

- Chapter 8, “Building and Deploying Business Objects” on page 255, explains how to build and compile Java classes or EJBs for business objects, and how to package these files in a J2EE enterprise application (EAR). This chapter describes how to use the available wizard to package these files and deploy them to a development Versata Logic Server running on IBM WebSphere Application Server Single Server Edition, and how to copy files and run a batch file to set them up on a production Versata Logic Server running on IBM WebSphere Application Server Advanced Edition.
- Chapter 9, “Understanding Business Object Files” on page 285, outlines the contents of the files that Versata Logic Studio generates for business objects, explains how to make external files available in a Versata repository, and describes the tools available to review and modify file properties and code.
- Chapter 10, “Extending Business Object Code” on page 321, discusses extensions to business object code. This chapter describes key generated code, including object instantiation code, event-handling code, SQL expression evaluation code , security management code, and remote object access code. This chapter provides procedures and examples for some common code extensions, including custom event-handling and subclassing.
- Chapter 11, “Working with Versata Connectors” on page 381, describes Versata Logic Suite’s eXtensible Data Access (XDA) structure and the Versata Connectors used for business objects’ database connectivity.
- Appendix A, “Transaction Logic Examples” on page 397, provides examples of different business requirements and illustrates how they can be enforced through Versata Logic Suite business rules.

CHAPTER 2

Developing a Data Model

Chapter overview

Read this chapter to understand how to complete tasks to create and modify your data model in the Versata Logic Studio. This chapter discusses the data model components stored in a Versata repository, outlines basic reference information to consider before you begin data modeling, and provides instructions for the different methods for creating a data model. After reading this chapter, you should be able to use the Versata Logic Studio to develop a data model for the data to be displayed and modified in your applications.

This chapter includes the following:

- "Data model overview" on page 33, introduces the basic structure and contents of a Versata data model.
- "Data model reference information" on page 36, provides some basic rules to follow as you are developing a data model in a Versata repository, including the following:
  - "Data model design guidelines" on page 36
  - "Denormalizing for performance" on page 37
  - "Naming conventions for data objects and attributes" on page 38
  - "Data type mapping between the Versata Logic Suite and RDBMSs" on page 40
  - "Sequential numbering in the Versata Logic Suite" on page 53
- "Building a data model" on page 56, provides instructions for creating a data model in a Versata repository, including the use of the Repository Exchange Manager and Reengineering Manager. This section also describes the file structure for Versata repositories.
- "Working with groups" on page 68, describes how to create and use groups to subdivide the objects in your data model.
- "Working with attribute templates" on page 73, describes how to create abstract attributes at the repository level and use them to implement attribute inheritance in repository data objects.

Note: For information about adding and modifying data objects in a data model, see "Working with Data Objects" on page 81.

For information about adding and modifying query objects in a data model, see "Working with Query Objects" on page 145.

Data model overview

A data model provides a logical representation of the way data is organized in a physical data source, illustrating the structure of the data and the relationships among the data. The data model is the basic building block for development. A data model must be present in a repository in order for you to define transaction logic (business rules) and design application user interfaces in the Versata Logic Studio.

Data models versus repositories

Data models are stored in repositories. Because each repository can contain only one data model, the terms data model and repository sometimes are used interchangeably and may get confused. Keep in mind that the repository is really more than the data model. Initially, a repository holds just the data model. Then, as you define transaction logic and, if you are using the Versata Logic Studio for presentation design, build the application user interface, these objects are added to the repository. The repository thus becomes a container for metadata about all of the application components that are defined in Versata Logic Studio.

Repository metadata is stored in `.xml` files. Each repository has its own `.xml` file. Each first-level repository object also has its own `.xml` file. First-level repository objects are data objects, relationships, query objects, applications, and forms. Of these, data objects, relationships, and query objects are considered part of the data model. The attributes for each data object also are part of the data model. Attribute information is stored in each data object's `.xml` file. For more information about Versata Logic Suite `.xml` files, see the *Reference Guide*.

You have the option of creating groups to contain the objects within your data model. Each group serves as a logical container for repository objects and their files in the Versata Logic Studio and as a physical container for object files on the filesystem. Your repository folder includes a subfolder for each group you create, and object files are stored within these group subfolders. The creation of groups eases work with large repositories. Groups usually correspond to functional areas or types of objects.

Object definitions

Following are some definitions of the objects in the data model:

- **Data object.** A representation of an object stored in a physical data source. Every data object usually maps to one relational table, but it may also represent an object from another type of data source, such as a packaged or legacy application. In the Versata Logic Studio, you view and modify data objects in the Transaction Logic Designer. You can define new data objects with the New Data Object wizard, accessed through the right-click sub-menu.

- **Attribute.** A characteristic of a physically stored data object, with defined values for different instances of the object. For objects stored in relational databases, each attribute maps to a column. The values for an attribute must be of a particular data type. In the Versata Logic Studio, you define, view, and modify attributes on the Attributes tab of the Transaction Logic Designer.
- **Relationship.** An association between two data objects based on matching values for an attribute that is included in both data objects. In the Versata Logic Studio, you define, view, and modify relationships on the Relationships tab of the Transaction Logic Designer.
- **Query object.** An object based on a SQL query that selects attributes from one data object or multiple related data objects. Serves as a reusable presentation object in applications, useful for limiting or grouping data displayed on forms. Query objects are not physically stored. In the Versata Logic Studio, you view and modify query objects in the Query Object Designer. You can define new query objects with the New Query Object wizard.

Data objects, including their attributes and relationships, are physically stored. To run applications from a Versata repository, data objects from that repository must be deployed to one or more physical data sources and connectivity must be established to these data sources. The Deployment Manager automates deployment for relational databases (including Microsoft SQL Server, Sybase, Oracle, Informix, and DB2 Universal Database). For information, see “Deploying a data model to a database server” on page 126. For these relational databases, the Versata Logic Suite also includes Versata Connectors that provide connectivity with applications. For other types of data sources, you need to manually manage data model deployment, and you need to write custom Connectors to provide connectivity. For information, see “Working with Versata Connectors” on page 381. Because query objects are not physically stored, they are not deployed to a database server. Query objects are a different class of objects than data objects, so information about working with them is in a separate chapter.

Both data objects and query objects are considered business objects. As such, they are exposed as distinct objects in the Versata Logic Studio Explorer. Transaction logic is defined on data objects. To run Versata Logic Studio-generated applications, you must build and compile business objects, then deploy them to the Versata Logic Server and the IBM WebSphere Application Server. The Versata Logic Studio automates these processes.

During the build and compile process, the Versata Logic Studio creates various `.java` files for each data object and each query object. The files for each data object include transaction logic execution code to implement rules defined on that data object. The implementation files for each data object and query object include those required to create Enterprise JavaBeans (EJB) for each one. Each data object can be built into an entity Bean, while each query object can be built into a session Bean. You set deployment properties in the Transaction Logic Designer and Query Object Designer to indicate whether to implement objects as Beans or simply as Java classes. During the deployment process, business object files are copied to locations accessible to the Versata Logic Server and accessible to IBM WebSphere Application Server. For more information about the files created when you build, compile, and deploy business objects, see “Building and Deploying Business Objects” on page 255.

Note: This chapter includes information about all RDBMSs supported by the Versata Logic Suite. Every release of the Versata Logic Suite may not support every RDBMS discussed in this chapter. For information about the RDBMSs supported by this release, see the *Getting Started Guide*.

Data model reference information

Review this section before you begin working on your Versata data model.

Data model design guidelines

When you build or modify a data model for use with the Versata Logic Suite, observe the standard guidelines for your environment and modeling tools. Before you begin work with your data model, work on defining your business requirements. For information about defining business requirements, see “Analyzing business requirements” on page 206. For information about the Versata Logic Suite development process, see the *Architecture and Project Guide*.

Data model design in the Versata Logic Suite is iterative. You can refine the data model as needed as you define business rules and application user interfaces, and discover additional requirements. When you modify the data model, you simply need to redeploy to the database server to implement changes. In most cases, you can use the Deployment Manager to automate this task.

You need to consider Versata Logic Suite-specific characteristics, because the data model serves as the starting point for transaction logic and applications.You must consider the attributes required to build business rules. Also, you must consider the data to be displayed on application forms or pages. The Versata Logic Studio provides techniques for you to refine your data model as needed for transactino logic and data display without changing the physically stored data. You can define virtual attributes, attributes that are calculated for use in transaction logic but not physically stored. You can create query objects, data sources that are instantiated as needed for data display rather than physically stored.

The following are additional issues to consider:

- You may need to define multiple query objects in your data model. Most applications use query objects as data sources instead of data objects. Query objects are generally more effective because they exclude unnecessary data attributes and include join data. Use the New Query Object wizard to define new query objects. Use the Query Object Designer to modify existing query objects. Define as many query objects as possible at the beginning of your development process, adding more after you prototype your application user interface and clarify which data to display on forms or pages. You can define query objects at any stage in the development process, and you can replace data objects with the new query objects at any time. For information about query objects, see page 145.

- You can use coded values lists to validate user selections of attribute values rather than picks or other referential integrity methods. Coded values lists are preferred over referential integrity rules when the number of values is small and the values do not change often. Define coded values lists early so that they appear in your prototype applications. You can add coded values list values at any point in development – you will not need to edit or recompile applications. For more information, refer to page 95.

  Note that data values for a data object used as a coded values list are not stored in the data object's `.xml` file, but in a `.csv` file of the same name.
- Give careful consideration to how you denormalize your data model. In many cases denormalization may simplify rule definition and improve application performance. For information, see "Denormalizing for performance" on page 37. The Versata Logic Suite provides virtual attributes so you can create attributes that are used for rule processing but not stored in the database. For information, see page 104.
- Use junction data objects to implement many-to-many relationships. For more information, see "Many-to-many relationships" on page 108.
- Implement type hierarchies in Store with Super data objects. For more information, see "Type hierarchies" on page 108.
- Observe the Versata Logic Suite naming conventions. Avoid the use of server-reserved words; for example, do not name a data object `Sort` or `Order`. Use singular names for data objects. Avoid the use of non-standard characters, such as embedded spaces, that require quoted identifiers. The Versata Logic Suite supports quoted identifiers, but they often are not supported in basic interactive query tools. It also is useful to set your own conventions in naming objects. For example, you might assign prefixes for coded values lists and query objects.
- Give careful consideration to attributes' data types. In some cases, you may be able to achieve performance improvements by modifying data types. For example, a memo attribute may require multiple SQL statements for an insert, but if you redefine that attribute to be text, variable length, only one statement is required.

Denormalizing for performance

Denormalization is often an issue when declaring business rules. In the Versata Logic Studio, we generally recommend that you denormalize data if it improves the performance of your applications. For example, you might make a data object that duplicates customer names and current balances if obtaining that information from the normalized data objects is particularly slow.

Another example of when to denormalize is when you need to assure that a customer's account balance does not exceed the credit limit. In a fully normalized database, the application would issue aggregate queries to sum each of the order items and orders. This querying would reduce performance noticeably, so the classical response is to calculate and store customer balance as a stored attribute and update it when orders are updated. In hand-coded systems, the trade-off for denormalizing has been a risk to database integrity—since the adjustment logic must be placed in multiple transactions—and disk space. In the Versata Logic Suite, such performance denormalizations are maintained with guaranteed integrity, and disk space is rarely an issue.

Note that performance-oriented denormalizations are in sharp contrast to structural denormalizations. Structural denormalizations are usually database design errors that result in hiding the proper number of data objects. The two most common errors are repeating fields and collapsing parent data in child rows. The denormalizations that we recommend are those that improve the performance of transaction logic execution, such as using a parent replicate derivation rule rather than a join.

Use the same procedures to denormalize the data that you would use in other development environments. The Versata Logic Suite maintains referential integrity for the denormalized data, so the only real cost of denormalization is disk space.

The Versata Logic Studio allows you to define derived attributes as virtual, meaning their values are calculated as necessary to provide values for rules, but they are not physically stored. This feature provides you with another option if you want to avoid denormalizing. For more information, see "Virtual attributes" on page 104.

Naming conventions for data objects and attributes

Naming conventions vary among data modeling tools. The following points explain how the Versata Logic Suite addresses names and recommends conventions for data models. We also recommend that you run the data model validation utility to check the data object and attribute names in your data model.

Note: Every release of Versata Logic Suite may not support every RDBMS discussed in this section. For information about the RDBMSs supported by this release, see the *Getting Started Guide*.

General naming conventions

- The Versata Logic Suite does not support data object names shorter than 4 characters if you intend to deploy the data object to the database server using Versata Studio.
- Data object names cannot begin with an underscore character.

- Data object names should not end with the text “Base”. This ending would cause the object’s implementation file name to look like a base implementation file, resulting in confusion.
- The Versata Logic Suite does not support attribute names longer than 31 characters.
- Spaces and underscores are the only special characters supported in data object or attribute names. If you use spaces in names, deploy the data model using quoted identifiers, and note that many tools do not support quoted identifiers.
- Use singular nouns for data objects (for example, EMPLOYEE rather than EMPLOYEES). They make better default captions in your applications.
- Each business object is defined in the interface files `<Object_name>.java` and `<Object_NameHome>.java`, and is implemented in the class file `<Object_NameImpl>.java`. Naming of custom object files follows the same pattern.
- By default, the attribute name is used for the attribute control caption on a form. You can define more meaningful captions (or captions with special characters) in the Transaction Logic Designer to override the default.
- Avoid using SQL reserved words (such as `Order` and `Date`) for data object and attribute names. Note that you are not prevented for using these types of reserved words for names. However, these errors are found when the data model is validated.
- For junction data objects (also known as "intersection" data objects), we recommend names that combine the names of the primary data objects. For example, you could use the name EMPLOYEESKILL for the junction data object linking EMPLOYEE and SKILL data objects.
- The Versata Logic Studio allows you create a data object with the same name as a user-defined Java file in the repository. In this case, if the data object is enabled for remote access, duplicate `.java` files exist. To avoid this duplication, do not give a data object a name that matches a repository Java file.

Informix naming conventions

- The Versata Logic Studio uses the first 13 characters of a data object name for code generation. To avoid package name duplication errors, make sure that the first 13 characters of your data object names are unique.
- The Versata Logic Studio uses the first 13 characters of an attribute name for code generation. To avoid duplication of variable names in packages when attributes are used in business rules (key and derived attributes), make sure that the first 13 characters of your attribute names are unique.
- Data object names and attribute names should not be longer than 18 characters.

Oracle, Sybase, and Microsoft SQL Server naming conventions

- The Versata Logic Studio uses the first 17 characters of a data object name for code generation. To avoid package name duplication errors, make sure that the first 17 characters of your data object names are unique.
- The Versata Logic Studio uses the first 17 characters of an attribute name for code generation. To avoid duplication of variable names in packages when attributes are used in business rules (key and derived attributes), make sure that the first 17 characters of your attribute names are unique.
- Data object names and attribute names should not be longer than 30 characters for Sybase or Microsoft SQL Server, and should not be longer than 29 characters for Oracle.

Data type mapping between the Versata Logic Suite and RDBMSs

When you reengineer an RDBMS database into a Versata Logic Suite repository, the data types of the native RDBMSs are mapped automatically to Versata Logic Suite data types.

When you use the Deployment Manager to deploy a Versata Logic Suite data model to a database server, the Versata Logic Suite data types are mapped automatically to the database server's native data types native to the database server.

For attributes that were reengineered from an RDBMS database, the Versata Logic Suite saves the original data type in the repository. If you do not change the attribute's data type, the saved data type is deployed back to the database server. If an attribute was changed after reengineering, was created in the Versata Logic Studio, or is being deployed to a different type of database server than the one from which it was reengineered, Versata uses the default data type from the mapping table for deployment.

Data type mappings are global in a repository and cannot be controlled on a per-object, per-attribute basis. Conversion of data types during reengineering is determined by the requirements of each RDBMS and cannot be modified.

For each Versata Logic Suite data type, there is a default attribute archetype used for applications designed in the Versata Logic Studio. This default is used to construct attributes in scalar displays. If you want to build a scalar display with non-default attribute archetypes, you can override the default archetype in the Transaction Logic Designer.

Note: Determine the correct data type for attributes that are keys early in your development process. You may not be able to change the data type of a key in the Versata Logic Studio. If you need to change the data type for an attribute that is a key, you need to drop the key, change the data type, then recreate the key and its index. You also need to review any relationships involving that key and any rules dependent on those relationships, and recreate them if necessary. For instructions for changing data types, see "Changing an attribute's data type" on page 103.

Every release of the Versata Logic Suite may not support every RDBMS discussed in this section. For information about the RDBMSs supported by this release, see the *Getting Started Guide*.

Some of the data types listed in the following mappings tables include terms in parentheses after the name of the type. For numerical data types, the first term in the parentheses represents the precision of the data type (the total number of digits it contains). The second term represents the scale of the data type (the number of decimal places it contains). For non-numerical data types, the term in parentheses represents the number of characters allowed for an attribute value of that data type. Also, the term p in parentheses represents the precision. The term (s) represents the scale of the data type.

Oracle and Versata Logic Suite data type mappings

The following mappings are used when you reengineer an Oracle database to a repository and when you deploy a Versata Logic Suite data model to an Oracle database.

Note: CLOB support is provided for Oracle 8, with the following guidelines: Read is fully supported for all CLOB attributes. Pre-populated CLOB attributes allow text updates of unlimited size. CLOB attributes that were not previously populated or that are being inserted as new records have a maximum text size of 4kb.

Reengineering from Oracle to the Versata Logic Suite

Original data types in Oracle RDBMS	Reengineered data types in Versata Logic Suite repository
Char (1-255) VarChar (1-255) VarChar2 (1-255) NChar (1-255) NVarChar2 (1-255)	Text

Original data types in Oracle RDBMS	Reengineered data types in Versata Logic Suite repository
Char (256-2000) VarChar (256-4000) VarChar2 (256-4000) NChar (256-2000) NVarChar2 (256-2000) Long CLOB NCLOB	Memo
Number (1-2,0)	Number, Size=Byte
Number (3-4, 0)	Number, Size=Integer
Number (5-9,0)	Number, Size=Long Integer
Float (63-126) Number (1-38,5-126)	Number, Size=Double
Float (0-62)	Number, Size=Single
Number (where no other mapping applies)	Number, Size=Decimal (p,s)
Date	Date/Time
Number (15,4)	Currency
Raw (1-255) LongRaw BLOB BFile Other unmapped	LongBinary

Deploying from the Versata Logic Suite to Oracle

The following table lists default data type mappings for deployments to Oracle. Mappings apply to attributes with data types that were not reengineered.

Data types in Versata Logic Suite repository	Data types as deployed to Oracle
Text	VarChar2
Memo	VarChar2 (1500)
Number, Size=Byte	Number (3,0)
Number, Size=Integer	Number (10,0)
Number, Size=Long Integer	Number (10,0)
Number, Size=Double	Float (126)
Number, Size=Single	Float (63)
Number, Size=Decimal (p,s)	Number (p,s)
Date/Time	Date
Yes/No	Number (3,0)
Currency	Number (15,4)
LongBinary	LongRaw
AutoNumber	Number (10,0)

Note: Decimal calculation errors may occur when a large value is entered for a column of data type Single in a repository deployed to Oracle.

Microsoft SQL Server and Versata Logic Suite data type mappings

The following mappings are used when you reengineer a Microsoft SQL Server database to a repository and when you deploy a Versata Logic Suite data model to a Microsoft SQL Server database.

Note: As of release 7.0, Microsoft SQL Server provides support for large character fields, up to a maximum length of 4000. Versata Logic Suite does not enforce its size requirements, so developers have responsibility for providing the data type sizes to meet these requirements. If errors occur during reengineering or deployment, developers need to fix them.

Reengineering from Microsoft SQL Server to the Versata Logic Suite

Original data types in Microsoft SQL Server RDBMS	Reengineered data types in Versata Logic Suite repository
Char (1-255) VarChar for SQL Server 6.5 and earlier (1-255) VarChar for SQL Server 7.0 and later (1-4000) NChar (1-255) NVarChar (1-255)	Text
Char (256-8000) VarChar (256-8000) NChar (256-8000) NVarChar (256-8000)Text NText	Memo
TinyInt	Number, Size=Byte
SmallInt	Number, Size=Integer
Int	Number, Size=Long Integer
Numeric Decimal Float (0-126)	Number, Size=Double
Real (0-62)	Number, Size=Single
Decimal or Numeric (where no other mapping applies)	Number, Size= Decimal (p,s)
DateTime SmallDateTime TimeStamp	Date/Time
Bit	Yes/No
Money SmallMoney	Currency
Binary (30-8000) VarBinary (30-8000) Image Other unmapped	LongBinary

Deploying from the Versata Logic Suite to Microsoft SQL Server

The following table lists default data type mappings for deployments to Microsoft SQL Server. Mappings apply to attributes with data types that were not reengineered.

Data types in Versata Logic Suite repository	Data types as deployed to Microsoft SQL Server
Text	VarChar
Memo	Text
Number, Size=Byte	TinyInt
Number, Size=Integer	SmallInt
Number, Size=Long Integer	Int
Number, Size=Double	Float
Number, Size=Single	Real
Number, Size=Decimal (p,s)	Numeric (p,s)
Date/Time	DateTime
Yes/No	Bit
Currency	Money
LongBinary	Image
AutoNumber	Int

Sybase and Versata Logic Suite data type mappings

The following mappings are used when you reengineer a Sybase database to a data model/ repository and when you deploy a Versata Logic Suite data model to a Sybase database.

Reengineering from Sybase to the Versata Logic Suite

Original data types in Sybase RDBMS	Reengineered data types in Versata Logic Suite repository
Char (1-255) VarChar (1-255) VarChar2 (1-255)	Text
Text	Memo
TinyInt	Number, Size=Byte
SmallInt	Number, Size=Integer
Int	Number, Size=Long Integer
Numeric Decimal Float (0-126)	Number, Size=Double
Real (0-62)	Number, Size=Single
Decimal or Numeric (where no other mapping applies)	Number, Size=Decimal (p,s)
DateTime SmallDateTime TimeStamp	Date/Time
Bit	Yes/No
Money SmallMoney	Currency
Binary (30-255) VarBinary (30-255) Image Other unmapped	LongBinary

Deploying from the Versata Logic Suite to Sybase

The following table lists default data type mappings for deployments to Sybase. Mappings apply to attributes with data types that were not reengineered.

Data types in Versata Logic Suite repository	Data types as deployed to Sybase
Text	VarChar
Memo	Text
Number, Size=Byte	SmallInt(5,0)
Number, Size=Integer	SmallInt
Number, Size=Long Integer	Int
Number, Size=Double	Float
Number, Size=Single	Real
Number, Size=Decimal (p,s)	Numeric (p,s)
Date/Time	DateTime
Yes/No	Bit
Currency	Money
LongBinary	Image
AutoNumber	Number (10,0)

Informix and Versata Logic Suite data type mappings

The following mappings are used when you reengineer an Informix database to a repository and when you deploy a Versata Logic Suite data model to an Informix database.

Reengineering from Informix to the Versata Logic Suite

Original data types in Informix RDBMS	Reengineered data types in Versata Logic Suite repository
Char (1-255) Character (1-255) VarChar (0-255) Character Varying (0-255) NChar (1-255) NVarChar (1-255) Interval (50)	Text
Char (256+) Character (256+) VarChar (256+) Character Varying (256+) NChar (256+) NVarChar (256+) Text	Memo
Number (1-2,0)	Number, Size=Byte
SmallInt	Number, Size=Integer
Int Integer Serial	Number, Size=Long Integer
Dec Decimal Numeric Double Precision Float	Number, Size=Double
Real SmallFloat	Number, Size=Single
Date DateTime	Date/Time, SubType=Date and Time
Money	Currency
Byte	LongBinary

Deploying from the Versata Logic Suite to Informix

The following table lists default data type mappings for deployments to Informix. Mappings apply to attributes with data types that were not reengineered.

Note: Indexed attributes that you plan to deploy to an Informix database must have a length (size) of less than 255. You cannot deploy to Informix if any attributes have indexed attributes greater than or equal to 255.

Data types in Versata Logic Suite repository	Data types as deployed to Informix
Text	VarChar
Memo	Text
Number, Size=Byte	SmallInt
Number, Size=Integer	SmallInt
Number, Size=Long Integer	Integer
Number, Size=Double	Float
Number, Size=Single	Float
Date/Time, SubType=Date and Time	DateTime Year to Second
Yes/No	SmallInt
Currency	Money (15,4)
LongBinary	Byte
AutoNumber	Serial

DB2 Universal Database and Versata Logic Suite data type mappings

The following mappings are used when you reengineer a DB2 Universal Database to a repository and when you deploy a Versata Logic Suite data model to a DB2 UDB database.

Note: DB2 UDB does not support some precisions that are supported for corresponding Versata Logic Suite data types, for example for VarChar and for Float.

Reengineering from DB2 Universal Database to the Versata Logic Suite

Original data types in DB2 UDB	Reengineered data types in Versata Logic Suite repository
Character (1-255) VarChar (1-255) Graphic (1-255) VarGraphic (1-255)	Text
Long VarChar Long VarGraphic VarChar (256-32672) VarGraphic (256-16336) CLOB DBCLOB	Memo
SmallInt	Number, Size=Integer
BigInt Integer	Number, Size=Long Integer
Decimal (Numeric) Double (0-126) (Float)	Number, Size=Double
Real (0-62)	Number, Size=Single
Date	Date/Time, SubType=Date
Time	Date/Time, SubType=Time
TimeStamp	Date/Time, SubType=Date and Time
BLOB	LongBinary

Deploying from the Versata Logic Suite to DB2 Universal Database

The following table lists default data type mappings for deployments to DB2 UDB. Mappings apply to attributes with data types that were not reengineered.

Data types in Versata Logic Suite repository	Data types as deployed to DB2 UDB
Text	VarChar Graphic
Memo	CLOB
Number, Size=Integer	SmallInt
Number, Size=Long Integer	Integer (10)
Number, Size=Double	Double
Number, Size=Single	Real
Date/Time, SubType=Date	Date
Date/Time, SubType=Time	Time
Date/Time, SubType=Date and Time	TimeStamp
Yes/No	SmallInt
Currency	Decimal (15,4)
LongBinary	BLOB

Note: DB2 Universal Database supports the AutoNumber data type used in Versata Logic Suite.

ANSI SQL and Versata Logic Suite data type mappings

The following table lists the ANSI SQL data types corresponding to the Versata Logic Suite data types.

ANSI SQL data types	Versata Logic Suite data types
VarChar	Text
Long VarChar	Memo
TinyInt	Number, Size=Byte
SmallInt	Number, Size=Integer
Int	Number, Size=Long Integer
Double Precision	Number, Size=Double
Real	Number, Size=Single
Decimal (p,s)	Number, Size=Decimal (p,s)
TimeStamp	Date/Time
Bit	Yes/No
N/A	Currency
Long VarBinary	LongBinary
N/A	AutoNumber

Sequential numbering in the Versata Logic Suite

Sequential numbering is a property of columns in database tables. It enables a numeric column to generate a sequential list of unique numbers upon insert. Across RDBMS platforms, this function is generally implemented through some form of sequence parameter or data type defined on the column.

When you reengineer an RDBMS table containing a sequence number column, the Versata Logic Studio converts it to the AutoNumber data type. When you deploy a data model to a database server, the Versata Logic Studio generates corresponding RDBMS objects and code for attributes with AutoNumber data types. The Versata Logic Studio also displays and fetches appropriate values when a user creates new rows that contain sequence numbers.

Note: Every release of the Versata Logic Suite may not support every RDBMS discussed in this section. For information about the RDBMSs supported by this release, see the *Getting Started Guide*.

Sequential numbering in Oracle

In Oracle, the `Create Sequence` command is used to assign unique numbers (such as customer IDs) and to create a sequence that can be accessed by `insert` and `update` statements. Here is an example of the `Create Sequence` command:

```
create sequence CustomerID
increment by 1
start with 10000;
```

The default increment value is `1`. A positive increment causes ascending incrementing of the sequence number. A negative increment causes descending incrementing. "`Start with`" establishes the seed value with which the sequence will begin.

For example, if you increment by 5 starting with 10000, the value for the first row upon insert will be `10005`, the value for the second row will be `10010`, and so on.

Typically, a sequence in Oracle is used in a set of statements like the following:

```
select <sequence_name>.NextVal from dual;
insert into <data_object_name> (<column_name1>, <column_name2>,
<column_name3>)
values (<'column_value1'>,
<'column_value2'>,<sequence_name>.CurrVal);
```

In this example, the sequence number property `NextVal` is associated with the column designated as the sequence number column and tells Oracle to generate the next sequence number value. Hence, the number is guaranteed to be unique. Upon insert, the sequence number property is retrieved each time a value is to be assigned, the value of the last row is referenced, and the next value is inserted.

Sequential numbering in Microsoft SQL Server and Sybase

For Microsoft SQL Server and Sybase, the identity property is assigned to a column at the time of table creation. Using this property, the database server automatically generates a sequence number and assigns it upon insert. As in Oracle, a starting value and an increment value must be assigned as parameters to the identity property.

Note: Once the identity property is assigned to a column in Microsoft SQL Server or Sybase, the only way to remove the property is to drop the data object and recreate it without the property. You cannot remove it with the `ALTER TABLE` command.

Identity Columns

When you deploy a data model to Microsoft SQL Server or Sybase, the Versata Logic Suite converts all AutoNumber attributes to Identity columns. The columns create unique, sequential numbers for rows upon `insert`. The following restrictions apply to Identity columns:

- You cannot remove the Identity property from a column using the `ALTER` command. The only way to remove the property is to drop the data object and recreate it without the property.
- You cannot update Identity columns. You must delete the old row and insert a new row. When you add a row, the value in the Identity column is automatically generated.

- Explicit values may be entered in Identity columns with the following guidelines:
  - If you load data through SQL scripts, you must manually set the server's `identity_insert` flag to `ON` for each applicable data object and specify the `insert` values for each row.
  - The database server may not be able to create sequential numbers, based on the initial value, increment value, and the failure of transactions to complete. Any one of these can contribute to gaps in Identity values created by the server. For specific information on how to address gaps, please refer to Microsoft SQL Server or Sybase documentation.
  - If the `identity_insert` flag is set to `ON`, the Identity column does not validate unique values. Under these conditions any user with permission can insert values into the Identity column. These values can be duplicates if there is no unique index or unique constraint defined on the Identity column.

Sequential numbering in DB2 Universal Database

DB2 Universal Database 7.x supports the AutoNumber data type used in the Versata Logic Suite.

If you choose not to take advantage of the AutoNumber capability, values for the attribute are generated by the `getcounter` method in the Versata Connectors code when you deploy the data model. This method queries the database, selects the largest value for the attribute and increments it by 1.

Building a data model

To get started in the Versata Logic Studio, you need to have a repository that contains a data model. The first step is to create a new repository, represented by a `<repository_name>.xml` file. For instructions, see "Creating a new repository" on page 58. Once the repository is open in the Versata Logic Studio, you can do any of the following to populate the repository with a data model:

- Use the Reengineering Manager to convert a supported RDBMS database to Versata Logic Suite `.xml` files. For instructions, see "Reengineering data objects into a repository" on page 60. After reengineering, use the data model validation utility to check your data model, as described in "Validating a data model" on page 62.
- Use the Repository Exchange Manager to import objects from another Versata Logic Suite repository. For instructions, see "Using the Repository Exchange Manager" on page 65.
- Use the Versata Logic Studio's menu options to add groups to contain repository business objects. For information, see "Working with groups" on page 68.
- Use the Versata Logic Studio's wizards to add data objects and query objects to the repository. For instructions, see "Adding data objects" on page 84 and "New Query Object wizard" on page 153.

Data objects' data are stored in `.csv` files with the same names as the data objects' `.xml` files. The `.csv` files contain any test data you enter as well as stored and display values for coded values lists. You can enter coded values list values in the Versata Logic Studio. However, you must enter other test data in an external tool that supports the `.csv` file format, such as Microsoft Excel. When you deploy a data model, you can transfer test data, but the Server Manager does not perform any data type or other validation for test data.

Note: If you want to work with a repository that is located on a machine that is remote from your Versata Logic Studio installation, you need to map the machine to a local network drive. If you do not perform this mapping, you may encounter errors.

Repository file structure

Release 5.5 of Versata Logic Suite utilizes a different structure for storing repository files on the filesystem. This new structure is designed to simplify repository file management and team development, and to facilitate integration with source control management systems. This physical structure approximates the logical structure of the Explorer's Files tab.

When you create a new repository, files are saved according to the new structure. When you open an existing repository in this release's Versata Logic Studio, you are asked to provide a location for saving the repository's files in this new structure.

Wherever you elect to save a new repository file or an upgraded repository file (`repository.xml`) on your filesystem, a subfolder with the same name as the repository is created within the chosen folder. The `<repository>` subfolder contains the subfolders displayed in the following figure.

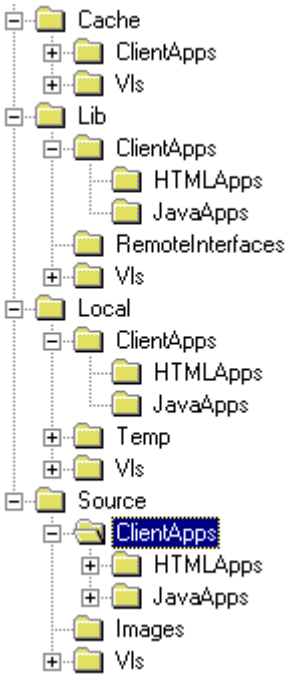

Figure 1 **Repository file structure**

- The `Source` subfolder is the main one you will be concerned with as you develop objects in the repository. This subfolder contains `.xml` source files, generated `.java` files, and any user-defined files for repository objects; application files are stored within application subfolders and business object files are stored within the `Vls` subfolder. The `Images` subfolder includes system-supplied image files.

  These files are the main files you will want to maintain under source control.

- The `Cache` subfolder contains cached versions of repository files including the cached version of the whole repository, the `<repository>.vdb` file; and a file used for business object deployment, the `repository.VJDeploy` file.

  In some cases you may want to maintain these files under source control, in order to optimize performance.
- The `Lib` subfolder contains compiled class files for repository objects.

  In some cases you may want to maintain these files under source control, in order to optimize compile times.
- The `Local` subfolder includes temporary files used by the system. You should never need to maintain these files under source controls.

Note: For information about managing repository files in a source control system in a team development environment, see the *Architecture and Project Guide*.

Creating a new repository

You create a new repository with the File→New Repository menu option.

To create a new repository:

1. Start the Versata Logic Studio.
2. Choose File → New Repository to open the Create New Versata Repository dialog.
3. Navigate to the folder where you want to create the repository folder. You can click the folder button to create a new folder.
4. Enter the name of the repository (`<repository.xml>`) or accept the default name, then click the OK button.

   The new repository structure is created within a new subfolder of the same name in the specified folder. This new subfolder MUST have a name identical to the repository name.

Upgrading an existing repository

The Versata Logic Studio provides a menu option you can use to upgrade an existing repository for this release. You may need to make further modifications after this conversion, particularly to any custom code you added to repository objects. For information about related migration issues, see the *Getting Started Guide*.

To upgrade an existing repository:

1. Start the Versata Logic Studio.
2. Choose File → Convert Repository to open the Convert to Versata 5.5 Repository dialog.
3. Navigate to the folder containing the repository.

4. Select the <repository.xml> file, then click the Open button.

Note: After converting your repository, Versata Logic Suite release 5.5 validates all file names at repository load time. This validation process ensures there are no name conflicts with potential data object or query object artifacts. The following file names are checked:

- <file_name>.xml
- <file_name>Impl.java
- <file_name>BaseImpl.java
- <file_name>.csv
- <file_name>.java
- <file_name>Home.java
- <file_name>DD.xml

Using the Reengineering Manager

Reengineering, sometimes referred to as "reverse engineering," is creating or modifying a Versata Logic Suite data object or data model by converting a non-Versata Logic Suite object or database. Use the Reengineering Manager for reengineering data models and additional wizards for reengineering objects.

- When you reengineer a data model or data object, the system maps attribute data types from their native server type to a Versata Logic Suite data type by using the mapping entries in the repository data object VSVBImportDatatypes. For information about data type mappings, see "Data type mapping between the Versata Logic Suite and RDBMSs" on page 40.
- Typically, you reengineer a data model to start the process of defining a new model. After you reengineer, you can modify the data model in the Versata Logic Studio, then you can begin adding transaction logic and developing applications in it. You also can redeploy the reengineered model as a new database or back to the same database, and the data from the original database can be loaded into the new one afterwards.
- You can reengineer data objects, which you are likely to do at any time in the development process.

To start the Reengineering Manager, choose Managers →Reengineering Manager.

Reengineering Manager user interface

The Reengineering Manager includes the following fields:

- **Selected Schema**. Shows the database schema of the current RDBMS. This list box may be empty until you are connected to a database.

- **Server Type**. Shows the server type of the RDBMS to connect to. To reengineer from an Informix database, the user performing the reengineering must have dba level permissions on the database. Also note that the Oracle7 option means Oracle7 or later.
- **Database**. Shows the databases on the current server. This option is enabled only if you are connecting to Microsoft SQL Server or Sybase.
- **Server Data Objects**. Lists the data objects in the current database.
- **Data Objects to Import**. Lists the data objects you have selected to import into your data model.
  - Click the Connect button to login to the server. The Select Data Source dialog opens.
  - Click Import Data Objects to begin the reengineering process.

Select Data Source dialog

Use the Select Data Source dialog to select the DSN (data source name) of the connecting database.

- The System Data Source tab lists your system DSNs. A system DSN is a shared data source. All network users may use it with the appropriate data driver.

  Click the New button to define a new system DSN. The Create New Data Source wizard opens.

- The User Data Source tab lists your user DSNs. A user DSN is a data source that may be used only on the current computer. It may be a user DSN (specific to one user) or a system DSN (any user).

  Click the New button to define a new user DSN. The Create New Data Source wizard opens.

Note: For help with the Create New Data Source wizard, see the help for your ODBC control panel(s) or open the help file itself, usually found in `C:\<Windows>\<System>\Odbcinst.hlp`.

Reengineering data objects into a repository

To reengineer a data model from a relational database:

1. Open the repository where you will copy the data model.
2. Choose Managers → Reengineering Manager to open the Reengineering Manager.
3. In the Reengineering Manager, select the type of database server from the Server Type drop-down list box.

   If you want to reengineer from an Informix database, the user performing the reengineering must have `dba` level permissions on the database.

   Note that the Server Type “Oracle7” means any Oracle database that is version 7 or later.

4. Click the Connect button to initiate communication with the database server. The Select Data Source dialog opens. For information about this dialog, see page 60.
5. Select a data source on the File Data Source or Machine Data Source tab, or define a new one.
6. Log in to the database server.
7. When the connection is made, select a user schema from the Selected Schema drop-down list box.
8. For Microsoft SQL Server and Sybase, you also must select a database.
9. In the Server Data objects list, select the data object(s) to import.
10. Click the > button to move the data object to the Data objects to Import list.
11. Click Import Data Objects to import the data objects into the data model. The Versata System rebuilds the data model to add the new data objects.
12. Click the Disconnect button to end communication with the server.

If any portion of the reengineering process fails, the system cancels the entire process. Revise the server data model and try again.

After you have completed the reengineering process, you should validate the reengineered data model. For information, see “Validating a data model” on page 62.

Note: Reengineering is not supported for Oracle tables that include objects.

If you reengineer from an Informix database that contains two tables of the same name, one in upper case, and one in lower case, only the table with the lower case name is reengineered successfully. Case-sensitive reengineering is not supported.

Notes on reengineering data models

- The Reengineering Manager imports data models on a per-schema basis, with each import occurring as a single transaction. There is no limit to the number of data objects or schemas, but all data object names must be unique.
- The Reengineering Manager does not import the following:
  - Multiple schema relationship constraints (relationships that span data objects in different schemas)
  - Data (records)
  - Defaults
  - Views
- Instead of views, you can use the New Query Object wizard to define query objects. For instructions, see page 145.

- Indexes and relationships implemented using relationship constraints are imported along with the data objects. Relationship constraints between the reengineered data objects are imported; other constraints are not. As an example, suppose data object `A` on the database server and data object `B` on the database server have relationship `C` between them. If you reengineer data object `A`, relationship `C` is not imported. If you later reengineer data object `B`, relationship `C` is still not imported. The only way to import relationship `C` is to import both `A` and `B` in the same reengineering.
- The Reengineering Manager imports relationships that are enforced in the database, if they are enforced declaratively with reference constraints on children that refer to primary key constraints in the parents. Existing triggers are not reengineered, so relationships enforced by triggers are not converted.
- After reengineering, you can add any missing relationships in the Versata Logic Studio. For instructions, see "Adding relationships" on page 113.
- You can add new attributes to data objects in the Versata Logic Studio. These attributes can be either physically stored or derived attributes used to calculate computations required for business rules processing. These derived attributes, called virtual attributes, allow you to take advantage of the Versata Logic Suite's derivation rules without denormalizing your data model or storing unnecessary data. For information, see "Adding attributes to data objects" on page 102 and "Virtual attributes" on page 104.

Validating a data model

The Versata Logic Studio provides a simple data model validation utility that can be run against the currently open repository. You should validate your data model whenever you have made substantive additions to the set of data objects and attributes in the repository through reengineering. Also, validation can catch naming errors such as using SQL reserved words.

The validation utility produces both a Validate Repository Data Object/Attribute Names Log that appears on screen and a `ModelValidation.log` file located in the repository directory. You can use either of these logs to review errors.

The file `ModelValidationCommands.txt` is an editable file in the Versata Logic Suite installation directory that defines the various checks performed by the validation utility. By default, this file defines checks for Microsoft SQL Server validation, checking data object and attribute names in the entire data model for:

- Embedded spaces
- Reserved words
- Invalid characters
- Invalid first characters
- Maximum name length
- Unique leftmost characters

You can edit the commands file as required to match your data source requirements. Also, if you are developing using quoted identifiers, you can ignore error messages for embedded spaces, reserved words, and invalid characters or edit the data model validation utility commands file to remove these checks.

Note: Another way to validate objects is to attempt a build. For information, see "Building and Deploying Business Objects" on page 255.

To validate a data model:

1. Launch the Versata Logic Studio and open the repository for which you want to validate the data model.
2. Choose File → Validate Repository Model to run the utility.
3. The Validate Repository Data Object/Attribute Names Log appears when the validation is complete. In this dialog, review any errors in the data object and attribute names in the repository. You also can review this information in the `ModelValidation.log` file, located in the directory that contains the repository file.

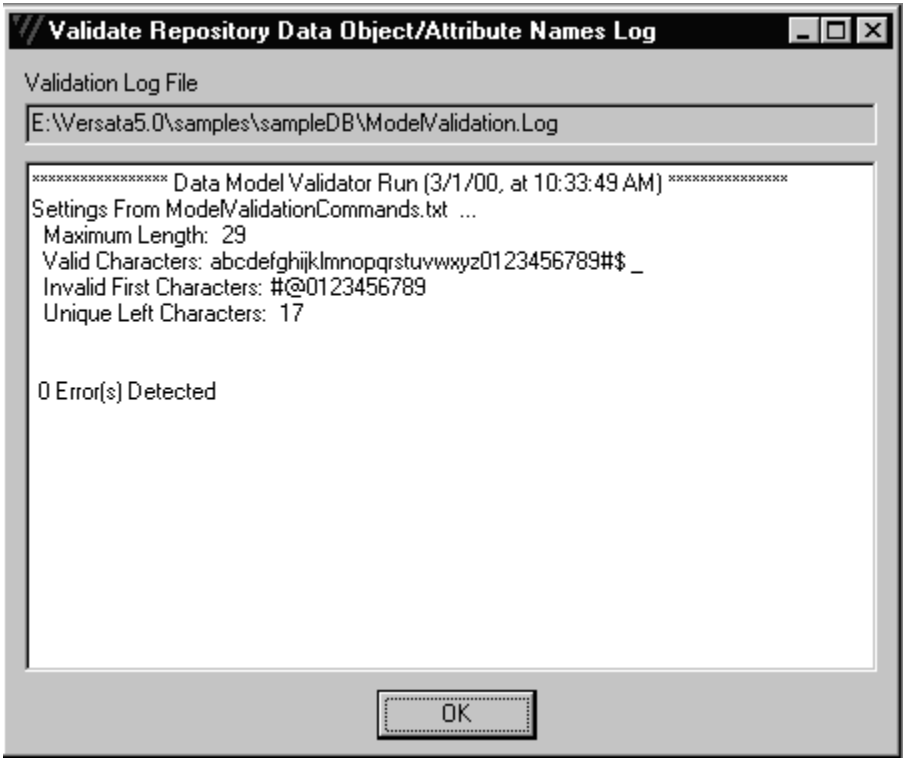

Figure 2 Validate Repository Data Object/Attribute Name Log

Note: If the disk is full when you attempt to validate a repository, a run-time error occurs.

Editing the data model validation utility commands file

The `ModelValidationCommands.txt` file is an editable file in the Versata Logic Suite installation directory that defines the various checks performed by the validation utility. By default, this text file defines checks for Microsoft SQL Server validation. You can edit this file as required to customize the check commands for your data source.

To edit the `ModelValidationCommands.txt` *file:*

1. Open `ModelValidationCommands.txt` (located in the product installation directory).
2. Modify the command lines (each begins with a !) to define the specific checks required for your data source.
   - Specify invalid characters for data object and attribute names. For example, names could be checked to ensure that they do not contain alphabetic, numeric, "#", "$", or "_" characters.
   - Specify invalid first characters for data object and attribute names. For example, names could be checked to ensure that they do not begin with "$" or "#".
   - Specify maximum length of data object and attribute names.
   - Specify the number of unique leftmost characters for data object and attribute names. For example, if your data model truncates data object or attribute names that have more than 10 characters, you could check to make sure that the first 10 characters of each data object or attribute name is unique.
3. Modify the list of reserved words at the end of the file to specify the reserved words you want to check for in data object and attribute names. Note that reserved words are case insensitive.
4. Save the changes and exit Microsoft Notepad.
5. Once you have edited the `ModelValidationCommands.txt` file as desired, make a copy of it, since the edited file will be replaced by the default file automatically each time you install a new version of the Versata Logic Suite.

Using the Repository Exchange Manager

The Repository Exchange Manager allows you to import repository objects from other Versata Logic Suite repositories to the current repositories. The Repository Exchange Manager copies the repository definitions of data objects, relationships, query objects, and applications, but not the data object data.

To use the Repository Exchange Manager, choose Managers →Repository Exchange Manager. After you navigate to the folder containing repository to be imported, the Import dialog opens.

Note: If the disk is full when you attempt to use the Repository Exchange Manager, a Versata termination error occurs.

Import dialog

This dialog displays objects (`.xml` files) in the source repository (where files will be copied from) in the left list box and displays objects in the destination repository (where files will be copied to) in the right list box.

- The Data Objects tab lists the data objects in the source repository on the left and the destination repository on the right.
- The Relations tab lists the relationships.
- The Query Objects tab lists the query objects.
- The Applications tab lists the applications.
- Choose the Show Groups option button to list repository groups only. This option is helpful if you plan to import objects by group. Choose the Show All option button to list repository objects individually. This option is helpful if you plan to import individual objects.
- Enable the Maintain Groups for Import check box if you want objects' containing groups to be imported along with the objects themselves.

Importing repository objects

Importing repository objects copies `.xml` files for data objects, relationships, query objects, and applications from another Versata repository directory to the current repository directory, also making changes to the repository `.xml` file as needed. You can use the Repository Exchange Manager for this task.

Note: No data verification is performed during an import. Therefore, you should check first to be sure that the import is not overwriting useful data in the current repository and that the objects referenced by the new information are available to the current repository.

Also note that the Repository Exchange Manager does not preserve read-only flags for imported object files.

To import repository objects:

1. In the Versata Logic Studio, open the repository to which you want to import.
2. Choose Managers → Repository Exchange Manager to open the Repository Exchange Manager.
3. In the Import from dialog, navigate to the repository from which to import objects. The Import dialog opens.
4. In the Import dialog, click the tab for the type of object you want to import. Enable the Maintain Groups for Import check box if you want objects' containing groups to be imported along with the objects themselves. Enable the Show All or Show Groups options as necessary.
5. Select one or more objects in the left list box and click >>>Import>>>. Use SHIFT and CTRL to select multiple objects.
6. Repeat step 5 on the same tab or other tabs to import additional objects as desired.

Note: If you import objects that have been deployed as EJBs to WebLogic, or deployed to a CORBA version Versata Logic Server, the repository may contain extra deployment descriptor files and interface files that are not usable.

Working with groups

A group is a container for a subset of repository business objects. The creation of groups eases work with large repositories. Groups usually correspond to functional areas or types of objects. Each group serves as a logical container for repository objects and their files in the Versata Logic Studio, and as a physical container for object files on the filesystem.

- In the Versata Logic Studio Explorer, group subfolders are located within the Business Logic folder on the Objects tab, and within the Versata Logic Server folder on the Files tab.
- On the filesystem, group subfolders are located within your repository's `Source\Vls` folder.

You can nest multiple levels of groups, so that group subfolders contain other, subgroup folders. It is recommended that you create groups early in the development process to define the basic structure for your repository. If you need to alter this structure, you can use the menu options or the Business Objects and Files Manager provided by the Versata Logic Studio. For information, see "Moving objects among groups" on page 69.

Note: Groups may not be listed in strict alphabetical order in the Versata Logic Studio Explorer. All groups with names beginning with upper case letters are listed before all groups with names beginning with lower case letters.

Adding groups

You can add a group to any business object folder in a repository, simply by going to that folder and choosing a right-click menu option.

To add a group to a repository:

1. On the Objects tab of the Versata Logic Studio Explorer, right-click the Business Logic folder or an existing group folder and choose New Group, or

   On the Files tab of the Versata Logic Studio Explorer, right-click the Versata Logic Server folder or an existing group folder and choose New Group.
2. In the Add New Group dialog, enter a name for the group and click OK.

   A folder for the group appears in the Explorer.

Note: You are not allowed to enter a duplicate group name, but no checking is done against object names, so you are allowed to create a group with the same name as a business object.

Moving objects among groups

The Versata Logic Studio provides right-click menu options and a Business Objects and Files Manager that you can use to change the contents of repository groups. Use the menu options to move a single business object, group, or user-defined file from one group to another. Use the Business Objects and Files Manager to move multiple objects or files among groups.

Note: If the disk is full when you attempt to move objects among groups, a "path not found" error occurs.

Moving a single object

To move a single object from one group to another:

1. On the Objects tab of the Explorer, right-click the object and choose Move.
2. If an informational dialog appears, click OK to dismiss it.
3. In the Choose Group dialog, select the group where you want to move the object and click OK.

   If you want to move the object to a new group, select the group to contain the new group, click the New button and complete the Add New Group dialog, then select the group and click OK.

Moving a single file

You can move a single user-defined file from one group to another. Because all files for a business object need to be contained in the same group, you cannot move a single generated file. You can move all of an object's files by moving the object, or you can use the Business Objects and Files Manager.

Note: Explorer icons for user-defined files contain red lines, while Explorer icons for generated files contain black lines.

To move a single user-defined file from one group to another:

1. On the Files tab of the Explorer, right-click the file and choose Move.
2. If an informational dialog appears, click OK to dismiss it.
3. In the Choose Group dialog, select the group where you want to move the file and click OK.

   If you want to move the file to a new group, select the group to contain the new group, click the New button and complete the Add New Group dialog, then select the group and click OK.

Moving a group

When you move a group, all of its contents, including any subgroups, are moved with it.

To move a group from one group to another:

1. On the Objects or Files tab of the Explorer, right-click the group and choose Move Group.
2. In the Choose Group dialog, select the group where you want to move the group and click OK.

   If you want to move the group to a new group, select the group to contain the new group, click the New button and complete the Add New Group dialog, then select the group and click OK.

Note: If the disk is full when you attempt to move a group, "Could not create folder" and "path not found" errors occur.

Using the Business Objects and Files Manager

Use the Business Objects and Files Manager to move multiple objects and files among repository groups.

To use the Business Objects and Files Manager, from the Versata Logic Studio main menu, choose Managers → Business Objects and Files Manager.

- The Objects tab of this manager lists repository business objects in the left list box and repository groups in the right list box.
- The Source Files tab of this manager lists business object files and user-defined files in the left list box and repository groups in the right list box.
- Choose the Show By Groups option button to list repository groups in the left list box. This option is helpful if you need to see which groups currently contain particular objects and files.
- Choose the Show All option button to list repository objects or files individually in alphabetical order.

To move one or more objects or files:

1. Select the object(s) or file(s) in the left list box.
   - You can use the SHIFT and CTRL keys to select multiple objects or files.
   - If the Show By Groups option is selected, double-click a group folder to make its contained objects or files available for selection.

2. Select a group in the right list box.
   - If you do not select a group before you click the > button, the selected object(s) are moved to the Business Logic folder, or the selected file(s) are moved to the Versata Logic Server folder by default.
   - If you want to move objects or files to a new group, select the group where you want to create the new group, then click New Group. Complete the Add New Group dialog, then select the new group.
3. Click the > button.
4. Close the Manager by clicking the x in the upper right corner. Then review the Explorer to ensure moves were completed correctly.

Note: You cannot use the Business Objects and Files Manager to move a group from one group to another. Use the right-click menu option for this purpose.

Renaming groups

To rename a group:

1. On the Objects tab or Files tab of the Versata Logic Studio Explorer, right-click a group and choose Rename Group.
2. Enter the new name in the Rename Group dialog.

Deleting groups

When you delete a group, all of its contents, including any subgroups, are deleted.

To delete a group:

1. On the Objects tab or Files tab of the Versata Logic Studio Explorer, right-click a group and choose Delete Group.
2. To confirm the delete, click Yes in the first Action Choice dialog that appears.
3. If the group contains objects or files, a second Action Choice dialog appears, asking whether to move contained objects to the deleted group's parent group.
   - To move objects or files to the parent group, preventing them from being deleted, click Yes.
   - To delete objects or files along with the deleted group, click No.

Note: If the disk is full when you attempt to delete a group, "Could not create folder" and "path not found" errors occur.

Finding objects and files

The Versata Logic Studio includes a Find utility that can be helpful for locating business objects and files in a large repository.

To use the Find utility:

1. On the Objects tab of the Explorer, right-click the Business Logic folder or one of its subgroup folders, or

   On the Files tab of the Explorer, right-click the Versata Logic Server folder or one of its subgroup folders.
2. Enter the name of the object or file that you want to find and click OK. (This dialog is not case-sensitive.)

   If the object or file exists in the repository, it is selected in the Explorer.

Building and compiling group files

You can elect to build and/or compile files for all objects in a group. For more information about building and compiling business objects, see “Building and Deploying Business Objects” on page 255.

To build all of a group’s objects:

1. On the Files tab of the Versata Logic Studio Explorer, right-click and choose the Rebuild menu option.

To compile all of a group’s objects:

1. On the Files tab of the Versata Logic Studio Explorer, right-click and choose the Compile menu option.

Note: Errors may occur if you choose this option before all objects in the repository have been compiled at least once. Because classes may reference each other, you may have to compile the entire repository before you can compile an individual group.

Working with attribute templates

You can include attribute templates in your repository to implement the inheritance of an abstract attribute's properties by multiple attributes in referencing data objects. You thus should consider whether to include attribute templates as you are developing your data model. You need to understand the following terms.

- **Attribute Template**: An attribute that does not belong to a particular data object. Attributes in referencing data objects can inherit the properties of this attribute.
- **Attribute Group Template**: A named collection of attributes that do not belong to a particular data object. Attributes in referencing data objects can inherit properties of this group of attributes as if it were an attribute template, and the entire group is inherited.
- **Propagate**: To force inherited properties in the inheriting attributes to be set to the same values as those in the referenced attribute templates or attribute group templates.

Attribute templates and attribute group templates can be referenced by any data objects in a repository. Definition information about attribute templates and attribute group templates is stored at the repository level. Each attribute template and attribute group template has its own `.xml` files. These `.xml` files conform to the `AttributeTemplate.dtd` and `AttributeGroupTemplate.dtd` files provided with this release. Reference (inheritance) information is stored in new properties in `.xml` files for data objects, relationships, and query objects. For information about these files, see the *Reference Guide*.

- To create an attribute template or attribute group template, you create an `.xml` file for it.
- To designate data object attributes to inherit from attribute templates or attribute group templates, you define properties in data object `.xml` files, and as appropriate, relationship and query object `.xml` files.
  - You specify that a data object attribute inherits from an attribute template by setting the `InstanceOf` attribute of the attribute element in the data object `.xml` file. For information about issues to consider, see "Issues with attribute templates" on page 75.
  - You specify that a data object inherits from an attribute group template by adding an `AttributeGroupInstance` element and setting its attributes in the data object `.xml` file. For information about issues to consider, see "Issues with attribute group templates" on page 77.
- To set properties in inheriting attributes to the same values as those in the referenced attribute templates or group attribute templates, use the `AttributePropagator.exe` provided with this release. For information, see "Propagating templates" on page 74.

Propagating templates

The `AttributePropagator.exe` utility is installed in the root installation directory when you install Versata Logic Studio. You can run this utility to propagate values from attribute templates and attribute group templates to inheriting attributes in the Versata Logic Suite repository.

Be sure to close the Versata Logic Suite repository before you run the utility.

To propagate attribute template and attribute group template values to a repository:

1. Create or update `.xml` files for attribute templates and attribute group templates.
2. Add or update inheritance information as necessary in data object and relationship `.xml` files.
3. Create a `<repository>\AttributeGroupTemplates\DeletedAttributes.txt` file to list attributes that are missing from an attribute group template and should be treated as deleted. The format for listing should be: `<Attribute_Group_Template_Name>.<Attribute_Name>`, with one listing per line.
4. Run `AttributePropagator.exe`.
5. In the first dialog, enter the name of the repository where propagation is to occur. You can click the button to browse to the repository `.xml` file.

   The propagation utility will create two log files:

   - `<repository>_propation.log` contains the changes made in all .xml files.
   - `<repository>_propagation_errors.log` contains any errors that occur during propagation.

   By default, these files are displayed on screen after the propagation process is complete, as well as written to the folder containing the repository `.xml` file. You can disable the check boxes to skip this display.
6. Click the OK button to start the propagation process.

   A Synchronization Process dialog displays status information. After propagation completes, a message is displayed. If enabled, log files also appear.

Note: Missing reference errors may occur if you delete or rename an attribute template or incorrectly specify the attribute template name in the data object `.xml` file. These errors are written to the log during propagation.

For information about special processing and errors that may occur as part of attribute group template propagation, see “Propagation of attribute group template changes” on page 78.

Issues with attribute templates

This section describes issues related to attribute templates.

You specify that a data object attribute inherits from an attribute template by setting the `InstanceOf` attribute of the attribute element in the data object `.xml` file.

- You can change an uninheriting attribute to one inheriting from an attribute template.
- You can change an inheriting attribute to an uninheriting attribute.
- You can change the attribute template from which an attribute template inherits.

You can use inheriting attributes in the same manner as other data object attributes. They can be used in indexes, as relationship primary keys or foreign keys, in rules, in query objects, and in forms or pages by using containing data objects or query objects as RecordSources (with presentation design only).

Property inheritance

When a data object attribute inherits from an attribute template, some attribute properties are always inherited from the attribute template, other properties are never inherited, and other properties may or may not be inherited in particular cases. The following sections describe property inheritance, listing properties according to their XML elements and attributes.

Always inherited properties

The following properties in inheriting attributes always must have the same values as in the attribute template.

- `ValueRequired`
- `DataType` element
  - `DataType`
  - `Size`
  - `Precision`
  - `Scale`
- `Validation` element
  - `ValidationType`
  - `CodedValuesList`
  - `Condition`
  - `ErrorMessage`

Never inherited properties

The following properties in inheriting attributes can never be inherited from the attribute template:

- Name
- Persistence
- LayoutByDefault
- ServerDataType element
  - Type
  - Size
  - ServerOfOrigin
- Derivation element
  - DerivationType, if other than None, Formula, or Default (all types available for local customization)
  - RelationshipSurrid
  - ParentReplicateIsMaintained
  - SourceAttribute
  - QualificationExpression
  - ExtendedProperties
  - HiddenProperties

Sometimes inherited properties

The following properties in inheriting attributes may or may not be inherited from attribute templates. Inheritance must be identified per inheriting attribute case. If a property is not inherited but is specified in the attribute template .xml file, the property needs to be listed in the data object .xml file as an Override element value for the attribute. Otherwise the value will be inherited.

- Caption
- Format
- MicroHelp
- ArchetypeName
- Description
- Comments
- PreventUserUpdate
- DerivationType, if None, Default, or Formula

Data type changes

Data type changes that occur in .xml files as a result of inheritance changes currently do not appear in the Versata Logic Studio.

- These changes may create invalid relationships if the primary key and foreign key data types do not match after changes. The Versata Logic Studio reports when this problem occurs.

- These changes may make expressions in query objects invalid. If this problem occurs, you need to edit the query expressions.
- These changes may make rules invalid. If this problem occurs, you need to edit rules.

Implementing changes in RecordSources

If a data object or query object containing inheriting attributes is used as a RecordSource and propagation causes changes in these attributes, you need to either rebuild the form or page containing the RecordSource, or drop and re–add the attribute to the form or page in order to implement changes.

Issues with attribute group templates

This section describes issues related to attribute group templates.

You specify that a data object inherits from an attribute group template by adding an `AttributeGroupInstance` element and setting its attributes in the data object `.xml` file.

- A data object cannot inherit individual attributes from an attribute group template, but can inherit only the entire group.
- A data object can inherit from an attribute group template more than once. For example, an attribute group template representing an address can occur twice in a data object, once for a home address and once for a work address.

  For each instance of inheritance from an attribute group template, an `AttributeGroupInstance` element with a unique `Name` must be added to the data object `.xml` file.
- Individual attributes in an attribute group template will be able to be referenced in SQL queries, to be referenced in business rule expressions, and to have locally customized properties as provided in attribute templates.
- If both data objects in a relationship inherit from an attribute group template, and you want the relationship to inherit from the attribute group template, then the relationship's `.xml` file must identify the `ParentAttributeGroupInstance` and the `ChildAttributeGroupInstance` attribute for the `Relationship` element, and these must inherit from the same attribute group template.
- If attributes in an index inherit from an attribute group template, the instance name must be specified in the `AttributeGroupInstance` attribute for the `Index` element in the data object `.xml` file.
- If a query object includes an attribute inherited from an attribute group template, all attributes in the attribute group template instance must be included in the query object.

Propagation of attribute group template changes

The same `AttributePropagator.exe` that propagates changes from attribute templates to repository data objects also propagates changes from attribute group templates. The propagation of attribute group templates presents some additional features.

Propagation of grouping

- If the `InheritGrouping` attribute of the `AttributeGroupInstance` element in the data object `.xml` file has a value of `True`, the group of inheriting attributes in the data object is placed together wherever the first attribute from the attribute group template is encountered.
- If the `InheritGrouping` attribute has a value of `False`, the placement of attributes in the object is not affected by the propagation process.

Propagation of order

- If the `InheritOrder` attribute of the `AttributeGroupInstance` element in the data object `.xml` file has a value of `True`, the group of inheriting attributes in the data object is placed in the same order as in the attribute group template.
- If the `InheritOrder` attribute has a value of `False`, no ordering of attributes is enforced.

Propagation of missing attributes

- If grouping is inherited and order is inherited, the new attribute is placed in the proper order within the group.
- If grouping is inherited and order is not inherited, the new attribute is placed at the end of the group.
- If grouping is not inherited, the new attribute is placed at the end of the data object.

Potential propagation errors

The following errors may occur during propagation from attribute group templates:

- Missing attribute group template instance name: Can occur when you delete or rename an attribute group template instance or incorrectly specify an instance name in an `.xml` file. The propagation utility reports this error.
- Missing attribute group template name: Can occur when you delete or rename an attribute group template or incorrectly specify an attribute group template name in an `.xml` file. The propagation utility reports this error.
- Missing attribute name in the attribute group template: Can occur when you delete or rename an attribute in the attribute group template or incorrectly specify the attribute name from the attribute group template in the inheriting attribute.

You can create an optional `<repository>\AttributeGroupTemplates\DeletedAttributes.txt` file that contains references to attributes in attribute group templates for which missing references should be deleted automatically during propagation.

- If the missing reference is not in this text file, then the propagation utility reports an error and you have a chance to correct the reference and prevent the loss of customizations.
- If the missing reference is in this text file, it is deleted, deletions are propagated, and any related customizations are lost. If the deletion causes the deletion of the last attribute in an index, the index is deleted.

Implementing changes in RecordSources

There is no explicit specification of inheritance from attribute group templates into RecordSources. RecordSources indirectly inherit based on inheritance into the data objects and query objects used as RecordSources.

If propagation of attribute group template changes into data objects results in new RecordSource attributes, these are displayed, by default, if any other attributes from the same attribute group template instance are displayed on the form or page.

HTML pages inherit changes to attribute group templates by rebuilding page layouts, just as changes to attribute templates are inherited. Additional attributes may appear on rebuilt pages as a result of changes. Customized archetypes can be used to control the appearance of the generated page so no further customization is required. The archetypes can apply to entire pages or to portions of pages, with some pages never being rebuilt and others being rebuilt whenever attribute template changes are propagated. Attribute group templates can be set up as portions of pages that can be rebuilt at will.

CHAPTER 3

Working with Data Objects

Chapter overview

Read this chapter to understand how to complete tasks to create and modify data objects in the Versata Logic Studio. This chapter includes the following:

- "Data object overview" on page 83, provides an introduction to Versata data objects.
- "Adding data objects" on page 84, explains different ways to add data objects to a repository.
- "Modifying data objects" on page 87, explains how to modify data objects, including setting optimistic locking and other properties and defining coded values lists.
- "Working with attributes" on page 98, describes how to add and modify data object attributes, including defining virtual attributes.
- "Working with relationships" on page 107, describes how to add and modify data object relationships.
- "Working with indexes and primary keys" on page 117, describes how to add and modify data object indexes.

Note: For information about importing data objects from other Versata repositories, see "Using the Repository Exchange Manager" on page 65.

For information about importing data objects from supported RDBMSs, see "Using the Reengineering Manager" on page 59.

For information about deploying Versata repository data objects to RDBMS database(s), see "Deploying Data Models" on page 121.

Data object overview

Data objects correlate to objects in a physically stored database. Data objects, their attributes, and their relationships provide the basic input for application and business rule design. They are the first thing to define in your repository.

You can reengineer data objects from an RDBMS or add data objects and define their characteristics in the Versata Logic Studio. After definition, each data object is represented in the repository as an `.xml` file (`<data_object_name>.xml`). The format of this file conforms to the `DataObject.dtd` file included with the product, located in the product installation directory. The `.dtd` file lists all of the nested elements and attributes that define the characteristics of each data object. Each data object `.xml` file includes values for these nested elements and attributes. For more information about Versata Logic Suite `.dtd` and `.xml` files, see the *Architecture and Project Guide*.

Once you have defined a data object in the Versata Logic Studio, you can use its definition as a basis to define business rules and applications. To use data objects in running applications, you need to build them into usable files that can be copied to the database server and application servers.

The Versata Logic Studio includes a Server Manager wizard to deploy data objects to the database server. The Server Manager compiles information from data object `.xml` files into SQL scripts that can be run against a supported type of RDBMS to create corresponding tables there. For more information, see “Deploying a data model to a database server” on page 126.

Note that the `.xml` file for a data object stores the data object’s metadata, not its data. Data, including coded values list values and test data, are stored separately in a `.csv` file named for the data object. If you want to work with test data for a data object, you can input data through Microsoft Excel or some format compatible with `.csv` files. During data object deployment to the database server, the Server Manager provides an option to automate test data transfer.

The Versata Logic Studio also provides menu options to build and compile each data object definition into files that run on the application server(s). The next step is to deploy these files to a development Versata Logic Server on IBM WebSphere Application Server Single Server Edition for testing purposes. The Versata Logic Studio includes a Versata Logic Server Deployment wizard that handles this deployment. You set a deployment property in the Transaction Logic Designer to indicate whether to deploy data objects as Enterprise JavaBeans (EJBs) or simply as Java class files. The deployed files contain data object definition information as well as transaction logic information defined as business rules. After they have been tested in the development environment, you can copy files to a production Versata Logic Server on IBM WebSphere Application Server Advanced Edition. For more information about building and deploying data objects, see “Building and Deploying Business Objects” on page 255.

Adding data objects

You may add data objects to a repository in the following ways:

- **Create**. Build the object in the Versata Logic Studio.
- **Import**. Use a data object `.xml` file from another Versata Logic Suite repository.
- **Reengineer**. Use an object that has to be reengineered into an `.xml` file before it can be imported. You may reengineer database tables, CORBA objects, COM objects, and JavaBeans and classes.
- **Add from XML**. Import an `.xml` file representing an object created outside of the Versata Logic Suite.

Note: You may need to create custom Versata Connectors for data objects after you create them.

Before you create a new data object, review "Naming conventions for data objects and attributes" on page 38.

Create New Data Object wizard

Use this wizard to create data objects in the repository. They will be added as standard data objects with standard interface files, unless you specify custom Connectors for them in the Transaction Logic Designer.

To start the wizard, select the Data Objects folder in the Versata Logic Studio Explorer. Then, right-click and choose New Data Object, click the button in the toolbar, or choose the Edit New Data Object menu option.

Creating a data object in the Versata Logic Studio

To build a new data object in the Versata Logic Studio:

1. Start the Create New Data Object wizard.
2. In the first dialog, choose Create.
3. In the Finished dialog, enter the name of the new data object.

   There is no required syntax for the name, but it must be at least four characters in length if you intend to deploy the data object to the database server through Versata Logic Studio. See "Working with coded values lists" on page 95 for information about naming conventions.
4. When you click the Finish button; the Transaction Logic Designer opens to the Attributes tab. Enter the attributes of the data object there. For instructions, see "Adding attributes to data objects" on page 102.

Importing a data object from another repository

To import a data object:

1. Start the Create New Data Object wizard.
2. In the first dialog, choose Import.
3. In the Finished dialog, click the Finish button to launch the Repository Exchange Manager. Use the Repository Exchange Manager to import the data object. For instructions, see “Importing repository objects” on page 66.

Reengineering a data object

You can use the Reengineering Manager to reengineer a data object from a relational database. You also can reengineer other types of objects to be data objects.

Note: Reengineering of EJBs currently is not supported.

To reengineer a data object:

1. Start the Create New Data Object wizard.
2. In the first dialog, choose Reengineer.
3. In the Reengineer New Data Object dialog, choose the type of object to reengineer and click the Next button. The appropriate dialog opens
4. If you choose the Database option, the Finished dialog opens, where you can click the Finish button to open the Reengineering Manager and import a data object from a database. For instructions, see “Reengineering data objects into a repository” on page 60.
5. If you choose another type of object, the Use Registered Object dialog opens.
   - To reengineer a registered object, select it in the list box and click the Next button.
   - To reengineer an un-registered object, click Register New Object. A file browser opens. Use it to select the object.

   The list box and file browser only show files of the type selected in the previous dialog.

   After you have completed this dialog, The Finished dialog opens. Click the Finish button to reengineer and import the data object.
6. If there are no Connectors included with the Versata Logic Suite that can be used for the new data object, you need to create one. You may create it now or later. To create it now, select Create New XDA Connector before you click the Finish button.

Adding a data object from XML

You can directly import an object created outside of the Versata Logic Studio as a data object, if it can be represented in an `.xml` file. The `.xml` file for the external object must conform to the `DataObject.dtd` this file. For information about this file, see the *Reference Guide*.

To add a data object from XML:

1. Review the `DataObject.dtd` and the `.xml` file for the external object, to ensure that the `.xml` file contains all values for all elements and attributes required by the .dtd.
2. Revise the `.xml` file as necessary to conform to the `.dtd`. If the `.xml` file includes elements not contained in the `.dtd`, you can make them into Hidden Property elements so they can be maintained in the `.xml` file for the Versata Logic Suite data object.
3. In the Objects view of the Versata Logic Studio Explorer, right-click the Data Objects folder and choose Add Existing.
4. In the dialog that appears, select the `.xml` file to be added as a data object.

Note: If the disk is full when you attempt to add an `.xml` object , a Versata termination error occurs and the Versata Logic Studio closes.

There is no validation for length of object names or for invalid characters when you add an existing .xml file to the repository as a data object. Be sure that the data object name is longer than four characters and its attribute names are shorter than 31 characters. Also, be sure that names do not contain special characters other than spaces and underscores. For more information about object naming conventions, see “Naming conventions for data objects and attributes” on page 38.

Modifying data objects

Renaming data objects

The Rename menu option allows you to save a data object under another name, deleting the currently named data object.

To rename a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, then the Data Objects folder.
2. Right-click the data object you want to rename and choose Rename.
3. In the Rename Data Object dialog, enter a new name and click the OK button. The data object is regenerated and its new name appears in the Versata Logic Studio Explorer.

Note: The Rename menu option is not available when the Transaction Logic Designer is open.

Once you rename a data object, you must rebuild any query objects based on the data object.

You have the option of using the Save As menu option to save the data object under another name when the Transaction Logic Designer is open. This option preserves the existing data object and creates a copy of it under the new name. When you use this option, be sure to choose the Save As option before you make any changes intended for the newly named data object. The Transaction Logic Designer implicitly saves many changes, so you may unintentionally alter the original object if you make changes before choosing Save As.

If the disk is full when you attempt a Save As, a file of 0 KB is written and no retry option is available.

A Save As of a read-only object does not create a read-only object.

Deleting data objects

To delete a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects and Data Objects folder.
2. Right-click the data object you want to delete and choose Delete.

   Any open applications are closed, to avoid reference problems.
3. Click the Yes button in the Action Choice dialog. The data object is removed from the Versata Logic Studio Explorer listing.

Note: The Delete menu option is not available when the Transaction Logic Designer is open. If you delete a data object, you can no longer use any query objects based on the data object.

If you delete a data object from a Versata repository, source control integration does not automatically delete it from your source control management system. You need to manually delete the object from the source control system. For information about Versata Logic Studio's integration with source control management systems, see the *Architecture and Project Guide*.

If you delete a data object that has been made remotely accessible, meaning it has been set to be deployed as an EJB, some of its files may not be deleted automatically. If you attempt to create another data object with the same name, a "potential conflict with existing data object" may occur. You can avoid this problem by checking for any remaining data object files after the deletion and manually removing them.

Generating an Impact Analysis Report

Before you make changes to a data object such as renaming it, deleting it, or renaming or modifying attributes, it is a good idea to determine which other repository objects are dependent on the data object.

To obtain this information, generate an Impact Analysis Report, which provides a "Where Used" analysis of the data object. This analysis includes information about data objects related to the selected data object, attributes dependent on a data object for use as a coded values list, query objects that include attributes from the selected data object, and applications that display data from the selected data object. Review this report to determine the other objects that may be affected by your data object change and then you can determine how to deal with these effects.

To generate an Impact Analysis Report:

In the Versata Logic Studio Explorer, right-click a data object and choose Impact Analysis Report.

The report process checks all data objects, query objects, and applications in the repository, displaying a Data Object Dependency Log when the process is complete.

Data Object Dependency Log

When you run an Impact Analysis Report to determine possible consequences of changes to a data object, the Data Object Dependency Log is created. It appears on your desktop and also is saved as `<repository_directory>\<repository>_JavaFiles\Components\<data_object>.Log`. You can open the saved file in Notepad.

This log lists the following:

- Data objects that are related to the selected data object
  - Name of each data object
  - Type of relationship
- Query objects that include attributes from the selected data object
  - Name of each query object
  - Application forms/pages that use query object as RecordSource
    - Name of application
    - Name of form/page
- Application forms/pages that use data object as RecordSource
  - Name of application
  - Name of form/page

If you decide to make changes to the data object, it is a good idea to review the objects listed in this report to determine whether changes are necessary.

Setting properties for data objects

You can set properties for repository data objects on the Properties tab of the Transaction Logic Designer.

To set properties for a data object:

1. In the Versata Logic Studio Explorer, double-click the data object to open it in the Transaction Logic Designer.
2. Click the Properties tab of the Transaction Logic Designer.
3. Click the appropriate subtab and complete fields as necessary, then save.

Properties tab of the Transaction Logic Designer

The Properties tab has six tabs to define data object presentation properties and other data object characteristics.

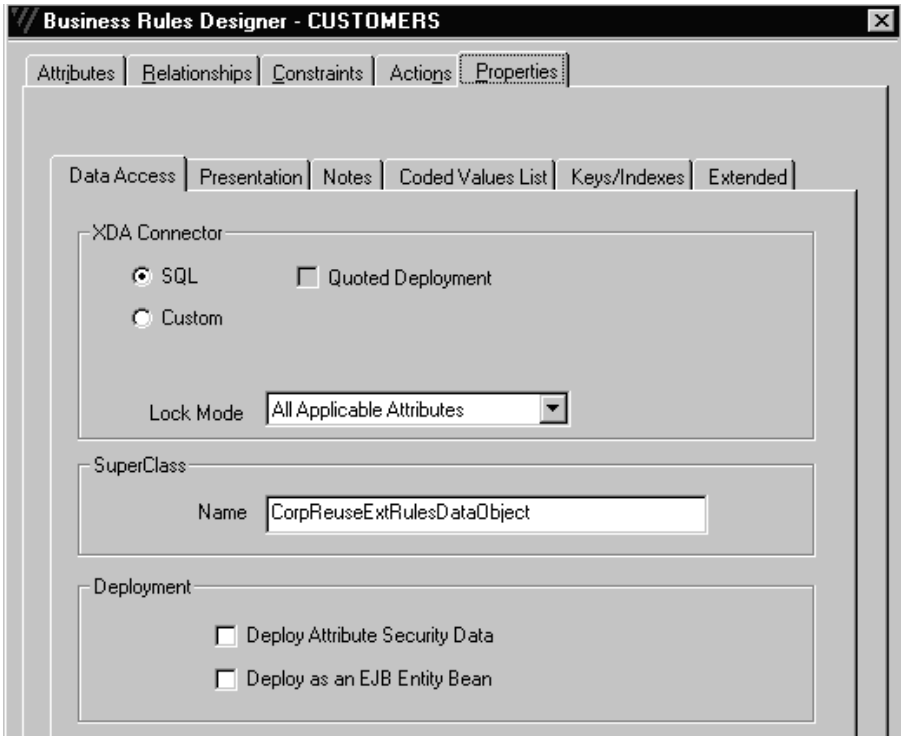

Figure 3 **Transaction Logic Designer Properties tab**

Data Access tab

The Data Access tab on the Properties tab in the Transaction Logic Designer allows you to specify the type of data source that the selected data object represents. By default, a data object represents a relational database table, uses SQL data access, and is a subclass of the `versata.vls.DataObject` superclass.

If data objects contain data from sources other than relational database tables, you need to add custom Connectors to the repository. Once you have added these Connectors, you should select the Custom option on this tab, click the browse button and select the new one from the Choose XDA Connector dialog.

If you want a group of data objects to have additional methods that are not defined in the `DataObject` superclass, you can create a subclass of `DataObject`, define new methods for this class and enter this new class as the superclass for the data object.

For a data object that uses a standard Versata Connector, the Quoted Deployment check box indicates whether the object has been deployed to a database server with quoted identifiers. This check box is not editable here; its value gets set by Server Manager deployment choices. For information about deploying with quoted identifiers, see “Generating quoted identifiers” on page 139.

This tab allows you to indicate a lock mode that determines the level of optimistic locking for the object. You can specify whether to compare values for all applicable attributes in the data object; compare the attributes changed by the current update action; or perform no optimistic locking. For more information about this property, see “Data type mapping between the Versata Logic Suite and RDBMSs” on page 40.

Also on this tab, you can set the data object’s deployment properties for the Versata Logic Server. You can indicate the following:

- Whether to enable attribute level security management for the data object. When you enable this option, additional information must be deployed to the Versata Logic Server, so deployment is slower and application performance can be slower. If you do not enable this option, you will need to assign permissions for the data object as a whole, rather than being able to assign different permissions for different attributes.
- Whether to implement the data object as an EJB. When you enable this option, the data object definition is deployed to the Versata Logic Server and IBM WebSphere Application Server as an entity Bean, so it is remotely available to any applications that can communicate with EJBs. If you do not enable this option, the data object definition is deployed as a Java class only. Deployment as an EJB requires more time than deployment as a Java class.

Presentation tab

Note: This tab is not available if you have not purchased presentation design capabilities with the Versata Logic Suite.

The Presentation tab allows you to override system-supplied defaults by specifying customized singular and plural captions for the selected data object. These captions appear on generated forms or pages where the data object is used as a RecordSource.

This tab also enables you to associate an image with the selected data object. This image appears on generated command buttons on the StartupForm/Page for transitions to a form or page based on this RecordSource. Note that to delete an image reference, you must select its name in the text box and press BACKSPACE or DELETE.

Notes tab

The Notes tab allows you to record a description and comments about the selected data object. This information is especially useful in a team development environment.

Coded Values List tab

The Coded Values List tab allows you to indicate that the selected data object should be used as a coded values list. To do so, enable Use this Data Object as a Coded Values List. A coded values list is a table of values used to restrict valid values for attributes.

After you enable this option, the Coded Values List Attributes dialog appears. Use this dialog to specify an attribute to provide stored values (the values stored in a database) and an attribute to provide display values (the values shown in controls or elements in applications) for a coded values list.

Values from the stored value attribute are stored in the database to represent values from the displayed value attribute. Values from the displayed value attribute are displayed in the generated application as potential values for attributes in any data object that has a validation rule referencing this coded values list.

- Select an attribute from the Attributes list and click an unfold button to enter the attribute in the Stored Value Attribute field.
- Select an attribute from the Attributes list and click an unfold button to enter the attribute in the Display Value Attribute field.
- When you have populated both fields, click the OK button. The selected attributes appear on the Coded Values List tab.
- You can modify these values by clicking the browse button to reopen the Coded Values List Attributes dialog.

After you have specified stored value and display value attributes, a table appears where you can enter valid values for attributes.

- To enter a valid value for an attribute, place the cursor in one of its fields and type the value.
- To modify a value for an attribute record, select it and type a new value.
- To add an attribute to the coded values list, click Add Column.
- To rename an attribute, select it in the table and click Rename Column.
- To delete an attribute, select it in the table and click Delete Column.

For more information about coded values lists, see “Working with coded values lists” on page 95.

Keys/Indexes tab

The Keys/Indexes tab allows you to review information about indexes defined for the data object, modify these indexes, add indexes, and delete indexes. For information about working with keys and indexes, see page 117.

Extended tab

The Extended tab allows you to add data object properties other than those explicitly specified in the Versata Logic Studio. Extended properties are useful in cases where you plan to add custom Java code to a data object. Code for these extended properties is generated in the data object’s Java implementation file. For each extended property, a static string variable is created inside the data object’s constructor code.

The data object’s extended properties perform a similar function to the extended properties for controls on application forms or pages: the properties provide additional behavior to data objects. You can add Java code to a data object that refers to the value for an extended property variable, where each different value causes different behavior in run time. Examples of variables that could be defined as extended properties include: an initialization variable for a class called by the data object, or the name of a DB2 database server.

To add an extended property, click the Add button and complete the dialog. Then, enter a property value in the grid.

To delete an extended pProperty, place the cursor in the grid row for the property and click the Delete button.

For information about the Java files that the Versata Logic Studio generates for data objects, see “Understanding Business Object Files” on page 285. For information about customizing code in data object files, see “Extending Business Object Code” on page 321.

Setting optimistic locking for data objects

The Versata Logic Suite provides support for optimistic locking-based concurrency control to prevent users from overwriting each other's changes when they update the same data. You can configure the type of locking so that you can achieve the right balance between maintaining data integrity without sacrificing performance. By default, the system retrieves data and displays it to the end user of the application without locks. When the end user updates the application by committing changes using the save action, the optimistic locking mechanism is invoked.

On the Properties: Data Access tab, you can choose an option from the Lock Mode drop-down list to determine the level of optimistic locking to be provided for a data object. The following options are available:

- **All Applicable Attributes**. All attributes that are updateable by users and easily compared (non-float data types) are included in the `Where` clause. This is the default setting.
- **Changed Attributes**. Any attributes that have been modified in the current update action are included in the `Where` clause.
- **No Optimistic Locking**. No optimistic locking is provided for updates.

This flexible optimistic locking mechanism uses a `Where` clause in the internal update statement that is executed when a user saves an update to one or more rows of data in the data object to determine whether the update should be completed. The attributes in this `Where` clause are compared to the matching attributes in the database to determine if another user has modified the data. This mechanism avoids the placement of explicit locks on any data that is read. If another user has modified the data in the `Where` clause, the update fails and an error is returned.

You can further customize the optimistic locking mechanism to include or exclude individual attributes in the `Where` clause. You can write custom code using the `inOptLock` methods of the `versata.common.VSMetaColumn` class.

Applicable attributes for optimistic locking are defined as all attributes that users can change and that can be compared easily. This definition excludes derived attributes and attributes with a float data type.

Note: For applications running against DB2 Univeral Database, the Versata Logic Suite attaches a lock to every auto-generated query issued to the DB2 database within a transaction. This lock prevents the current session from reading any uncommitted changes caused by other sessions.

For applications running against Oracle, if a data object contains an attribute of data type Time, no attribute on that data object can be updated. To work around this problem, do not include the Time attribute for optimistic locking. The Time data type is supported only for DB2.

You can set additional transaction isolation levels and optimistic locking properties on data servers in the VLS Console. For information, see the *Administrator Guide*.

Enabling resynchronization with a persistent data source

The extended property, `refreshAfterUpdate`, indicates whether or not to resynchronize data between a business object and its persistent data source after a transaction is committed. If this property is set, it overrides the default values for business objects. The possible values are `true` and `false`. By default, data objects are specified as `false`. To override this default, you can add the extended property to a query object and set its value to `true`.

To enable resynchronization of a data object with its persistent data source:

1. In the Versata Logic Studio Explorer, double-click the data object to open it in the Transaction Logic Designer.
2. Click the Properties tab, then the Extended tab.
3. On the Extended tab, click the Add button.
4. In the dialog that appears, enter `refreshAfterUpdate` and click OK. This entry appears in the Property Name column of the extended properties table.
5. In the Property Value column of the table, enter `false`.
6. Click the Save toolbar button.

Working with coded values lists

Coded values lists are data objects containing lookup values that can be used to validate user entries. Almost any set of finite and relatively permanent data is appropriate for a coded values list. For example, U.S. state abbreviations, credit limit categories, and payment methods are typical uses of coded values lists.

Coded values lists consist of pairs of corresponding values. Each pair has a stored value and a display value. The stored values are stored on the database server; the display values are shown in a combo box to the user. When the user runs the application, he or she selects a display value, and then the corresponding stored value is written into the row and validated on the database server.

We recommend that you use all capital letters for the names of coded values lists and prefix each name with VALID_, as in the sample repository. To use a coded values list, define a validation rule with it in the Transaction Logic Designer.

You can use a data object both as a RecordSource in run-time applications and as a coded values list. Attributes in an existing data object may be used as a coded values list without interfering with the attributes' primary use in the data object. In this case, Versata Logic Studio generates code that checks attributes to maintain referential integrity as well as code that checks the attributes to validate against the coded values list data.

Coded values list values are stored in a `.csv` file with the same name as the data object. You can enter and modify values in the Versata Logic Studio or in another program. To deploy coded values list values to the database when you deploy a data model, be sure to enable the transfer of test data for these objects.

Note: If you make changes to coded values lists, you should use the Rebuild All command to register the changes in applications that display coded values list values.

Defining a coded values list

You can specify that an existing data object should be used as a coded values list on its Properties:Coded Values list tab. You also can enter valid attribute values on this tab. For information, see "Coded Values List tab" on page 92.

To designate a data object as a coded values list:

1. Double-click the data object in the Versata Logic Studio Explorer. The Transaction Logic Designer opens.
2. Select the Properties:Coded Values List tab.
3. Enable the Use this Data Object as a Coded Values List option. The Coded Values List Attributes dialog opens.
4. Select the attribute whose values will serve as the coded values list's stored values and click the > button to copy it to the Stored Value Attribute text box.
5. Select the attribute whose values will serve as the coded values list's display values and click the > button to copy it to the Display Value Attribute text box.
6. Click the OK button to close the dialog.
7. If desired, enter or modify valid values for attributes. Also, you can add, rename, and delete attributes as necessary by using the tab's command buttons.
8. Choose File → Save Transaction Logic.

Caching coded values lists

It is important to note that all coded values are cached in the Versata Logic Server. This improves performance because starting a new instance of the application does not require requerying the database. Any business objects that are based on coded values will automatically flush the cache when they get updated. In addition, once the coded values list is cached on a particular Versata Logic Server, all clients using that Versata Logic Server will share the cache. You can determine whether a coded values list is in the cache by checking in the `VLSout.log` file.

In a running client application, coded values are cached in the client as well, and these values are not automatically updated unless you stop and restart the application. In a situation where you want to update a running application with new coded values, use the following API to refresh the client cache from the server:

```
VSMetaManager.refreshCodeTable(<coded_values_list_name>)
```

Working with attributes

When you build a data model in the Versata Logic Studio, you create a data object to represent each data source for which you need to store records in the database. For each data object, you create attributes to represent characteristics for which you need to store values in the database.

Define data object attributes as completely as possible before you convert your data model to a Versata Logic Suite repository, but as you define declarative business rules and applications, you may discover that you need to make some changes. If you have already opened the data model in the Versata Logic Studio, use the Transaction Logic Designer to add, delete, or rename the attributes of data objects in your data model.

The attribute information from the Versata Logic Studio is included in each data object's `.xml` file. For more information about Versata Logic Suite `.xml` files, see the *Reference Guide*.

Note: If you need to make changes to the attributes included in query objects, use the Query Object Designer. You can include attributes that exist in an underlying data object or define formulas for computed attributes to be in the query object. For information, see "Adding query objects" on page 152.

Before you add or make changes to attributes, review "Naming conventions for data objects and attributes" on page 38.

If multiple data objects need to share the same attributes, these attributes can be inherited from attribute templates. For information, see "Working with attribute templates" on page 73.

Attributes and declarative business rules

The Versata Logic Suite's declarative business rules allow you to define transaction logic for changes to attribute values, so that when a user changes one attribute's value, the values of all related attributes are recalculated automatically. You can use derivation rules to calculate the values of related attributes across multiple data objects.

- Sum and count rules calculate the values of parent data object attributes based on the values of child data object attributes.
- Formula rules calculate the values of attributes based on the values of other attributes in the same data object.
- Parent replicate rules calculate the values of child data object attributes based on the values of parent data object attributes.

For more information about derivation rules, see "Types of business rules" on page 189.

As you define rules, you may need to create new attributes to store calculations that can be used to calculate the value of other related attributes. These newly created attributes may store information that is already stored in other attributes elsewhere in the data model, and information that users will never need to see. In these cases, you can create virtual attributes.

The system calculates values for virtual attributes only when these values are required to determine the values of other attributes, storing the values temporarily in cache, but not saving them to the physical database. Virtual attributes allow you to take advantage of derivation rules without denormalizing your data model or storing unnecessary data. For more information about virtual attributes, see “Virtual attributes” on page 104.

Attributes tab of the Transaction Logic Designer

You can add, delete, or modify attributes for data objects on the Attributes tab of the Transaction Logic Designer. You also can define derivation, validation, and presentation rules for attributes on this tab. For information about these rules, see page 189.

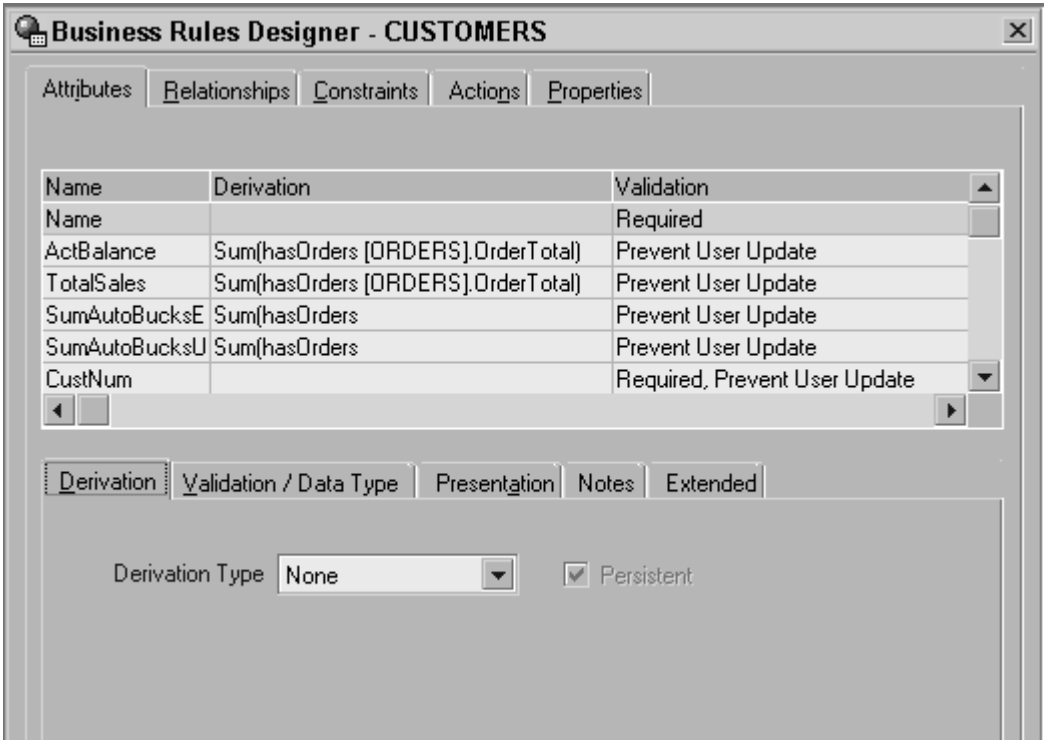

Figure 4 BRD Attributes tab

The Attributes tab of the Transaction Logic Designer has a read-only grid of all the attributes in the selected data object, with their attribute-level rule information. Rules are not input directly into the grid. This tab also contains a tab control with tabs for each type of attribute-level rule, as well as a tab where you can enter notes about the attribute.

When the Attributes tab is selected, the Add Attribute, Delete Attribute, and Rename Attribute options are available from the Edit menu, and buttons on the main toolbar become enabled for Add Attribute and Delete Attribute. Choosing Add Attribute opens the Add Attribute dialog. Choosing Rename Attribute opens the Rename Attribute dialog.

The Extended tab allows you to add attribute properties other than those explicitly specified in the Versata Logic Studio. Extended properties are useful in cases where you plan to add custom Java code for an attribute. Code for these extended properties is generated in the attribute data object's Java implementation file. For each extended property, a static string variable is created. To add an extended property, click the Add button and complete the dialog. Then, enter a property value in the grid. To delete an extended property, place the cursor in the grid row for the property and click the Delete button.

Add Attribute dialog

To add an attribute, complete the following fields in this dialog:

- **Name**. Observe the following conventions.
  - The attribute name can be up to 64 characters and can include alphanumerics and underscores.
  - The first 19 characters are used for code generation so these should be unique.
  - Spaces are permitted in attribute names, but not recommended. If your attribute names have spaces, you must use quoted identifiers when you deploy to the server, and many third party tools do not work with quoted identifiers.
- **Type**. Choose from the following data types.
  - Text
  - Memo
  - Number
  - Date/Time
  - Yes/No
  - Currency
  - LongBinary
  - AutoNumber
- **Size**. This field appears only if you select a Text data type. Enter the number of characters permitted for the attribute value in the Size field. Up to 255 characters are permitted.

- **Sub Type**. This field appears if you select a Text, Number, or Date/Time data type. Choices vary according to which type you selected.
  - If you selected a Text data type, choices are:
    - Variable length (the default)
    - Fixed length
  - If you selected a Number data type, choices are:
    - Byte
    - Integer
    - Long Integer
    - Double
    - Single
    - Decimal

      If you select Decimal, you need to enter a precision and a scale. Precision is the total number of digits stored for an attribute. Scale is the total number of decimal places stored for an attribute.
  - If you selected a Date/Time data type, choices are:
    - Date and Time (the default)
    - Date
    - Time

Note: For attributes with formula rules, data type, subtype, and length information is not used, except to determine the archetype for presentation formatting.

Note about binary data types

Currently, the Transaction Logic Designer does not support binary data types other than LongBinary. By default, values for attributes of this data type are not retrieved at run time during query execution, due to performance optimization. To work around this issue, do the following:

- In the `beforeQuery` event for the data object containing a binary attribute, add the following code:

```
query.setColumnProjectionLevel(DataConst.ALLTYPES);
```

  This code enables queries of all binary data types, including binary, varbinary, and longvarbinary.

- In addition, you may need change the following lines in the `<install_directory\RuntimeJava\Archetypes \javaMetaQueryColumnCtorArchetype.tpl` file:

```
<<if value(VSVBColumn!DataType) = "LongBinary">>
     c = new VSMetaColumn("<<Name>>",
DataConst.LONGVARBINARY);
```

  You need to change `DataConst.LONGVARBINARY` to `DataConst.VARBINARY` or `DataConst.BINARY` to map to the corresponding data type in the database.

Adding attributes to data objects

To add an attribute to a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects and Data Objects folders, then select the data object to which you want to add an attribute.
2. Choose Edit → Add Attribute or click the Add Attribute toolbar button. The Transaction Logic Designer and Add Attribute dialog open.
3. In the Add Attribute dialog, enter a name for the attribute. From the Type drop-down list box, choose a data type. For a Text attribute, enter a size. For a Text, Number, or Date/ Time attribute, select a sub-type. Click the OK button.
4. In the Transaction Logic Designer, define derivation, validation, and/or presentation rules for the attribute. For information, see "Understanding the Transaction Logic Designer" on page 220.
5. Choose File → Save Transaction Logic.

Note: If you are adding a derived attribute to store calculations used in rules processing but you do not want to physically store the attribute in the data model, you can disable the Persistent option on the Derivations tab of the Transaction Logic Designer.

If you are adding an Autonumber type attribute, be sure to enable the Prevent User Updates check box on the Validation/Data Type tab.

Also, indexed attributes that you plan to deploy to an Informix database must have a length (size) of less than 255. You cannot deploy to Informix if any attributes have indexed attributes greater than or equal to 255.

Deleting attributes from data objects

To delete an attribute from a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folders, and the data object where you want to delete an attribute. Select the attribute you want to delete.
2. Choose Edit → Delete Attribute, click the Delete Attribute toolbar button, or right-click the attribute and choose Delete Attribute. The Transaction Logic Designer opens and a dialog appears asking you to confirm the deletion.
3. Click the Yes button.

Note: You cannot delete an attribute that is used in a relationship unless you delete the relationship first.

Renaming attributes

To rename an attribute:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and the data object where you want to rename an attribute. Select the attribute to rename.
2. Choose Edit → Rename Attribute, click the Rename Attribute toolbar button, or right-click the attribute and choose Rename Attribute. The Transaction Logic Designer opens.
3. If the attribute is used in a relationship, a dialog appears informing you that you cannot rename the attribute. Click the OK button.
4. If the attribute is not used in a relationship, a dialog appears asking you to confirm the renaming. Click the Yes button.
5. In the Rename Attribute dialog, enter a new name and click the OK button.

Note: When you rename an attribute, references to the attribute in applications, constraints, and query objects may not be updated. You may need to update these references manually.

Changing an attribute's data type

Use the Transaction Logic Designer to change the data type for an attribute.

For information about data type mappings between Versata Logic Suite and supported RDBMSs, see page 40.

To change an attribute's data type:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Click the Attributes: Validation / Data Type tab.
3. Select the attribute.
4. Select a new data type from the Data Type drop-down list. For a Text data type, enter the number of characters permitted. For a Text, Number, or Date/Time data type, select the sub type.
5. Choose File → Save Transaction Logic.
6. You may want to change the presentation format for the attribute, to fit with the new data type. You can do this on the Attributes:Presentation tab of the Transaction Logic Designer For information, see "Presentation tab" on page 225.

Note: If you attempt to alter the presentation format after changing from a Data Time data type and before saving the change, format choices are not correct. Save the data type change, then retry.

Virtual attributes

Virtual attributes are available in all Versata Logic Studio designers and can be placed on application forms or pages, but they are not deployed to the database server. You can project virtual attributes into query objects. Virtual attributes may be referenced by name in rules, including derivations, constraints, and action rules. Virtual attributes also may be used in `Where` clauses for other derived attributes.

When you design your data model, you need to make decisions about which attributes should be stored and which should be virtual. You need to be conscious of the balance between the benefits of virtual attributes and the performance impact of in-memory calculations performed to obtain virtual attributes' values. Your data model should have a mix of stored and virtual derived attributes.

To define an attribute as virtual, you must define a derivation rule for it, then you must disable the Persistent option on the Derivations tab of the Transaction Logic Designer.

Virtual attributes are recommended for the following:

- Most attributes with formula rules.
- Parent replicates of stored attributes.
- Other derived attributes that are not displayed on application forms or pages.

There are four absolute restrictions on the use of virtual attributes:

- Virtual attributes may not be used as primary or foreign keys.
- Attributes that are unmaintained replicates may not be virtual.

- Attributes with default rules may not be virtual.
- No `:old` value is available for virtual attributes.

In addition to the above restrictions, observe the following limitations:

- Do not use virtual attributes as search criteria on a form or page that uses the Grid Select RecordSource archetype.
- You cannot sort on a virtual attribute in a grid.
- You cannot apply functions to virtual attributes.

Because a virtual attribute is not stored, it is comparable to a function. So as you are designing rules and determining whether attributes used in rule calculations should be stored or virtual, you should keep in mind where it is appropriate to reference a function in rules. You can apply similar principles to determining where to use virtual attributes.

A virtual attribute will need to be recalculated each time it is used, and this recalculation can slow performance in certain cases. Consider the following factors:

- Whether the derivation of the attribute is complex.
- Whether the derivation of the attribute requires the calculation of one or more virtual attributes.
- Whether the attribute is used in many other rules.
- How frequently the value of the attribute must be recalculated.
- Whether the attribute is displayed on application forms or pages, particularly in grids.
- Whether other rules require access to the previous value of the attribute.

If the derivation of an attribute is complex or already involves the input of one or more virtual attributes, it is probably best to store the attribute, to avoid the time necessary for repeated recalculations. Also, if the attribute needs to be recalculated frequently, either to serve as input for other rules, or to be displayed on forms or pages, it may be best to store it, particularly if the attribute is displayed in one or more grids. Conversely, simple calculations that do not need to be performed frequently are good candidates to be virtual attributes.

- Do not create virtual attributes that are sums of other virtual sums. This is called cross-object aggregation and can have severe performance implications.
- If attributes will be displayed on forms or pages, do not make them virtual. When you place data objects or query objects on a form or page, by default the virtual attributes in the objects are generated on the form or page. To avoid the placement of virtual attributes on forms or pages, you can create query objects that filter out the virtual attributes in data objects.
- Avoid using virtual attributes in the `Where` clauses of sums and counts in which the value of the virtual attribute must be recalculated for each record in the data object to evaluate the condition.
- Avoid using virtual attributes that are sums or counts in `Where` clauses or formula expressions.

Example - virtual attributes in sum and count rules

An account balance attribute for a customer, which is used to ensure that the customer does not exceed its credit limit, is a good example of an attribute which normally should remain persistent because it needs to be recomputed every time an order is added or modified or paid. For example, if you write a credit card payment tracking application, you would definitely want this attribute to be persistent, since the alternative is to review the entire transaction history.

If you are writing software for a car dealership, however, where there might be an average of two transactions per customer in a calendar year, making the attribute non-persistent might be an excellent design choice because the need to recompute it is low. Even then, if the account balances must be displayed on a grid of customers, then it would be wise to store it.

Defining an attribute as virtual

Generally, you define an attribute to be virtual when you define a derivation rule for that attribute.

To define an attribute as virtual during rules definition:

1. In the Versata Logic Studio Explorer, double-click a data object to open the Transaction Logic Designer.
2. On the Attributes tab of the Transaction Logic Designer, select the attribute in the grid.
3. Select a type of derivation rule from the drop-down list.
4. Enter data object, attribute, and/or expression as appropriate.
5. Click the Persistent check box to remove the check.

For more information about defining derivation rules, see page 232.

Note: Computed attributes in query objects are different from virtual attributes. The values for computed attribute records are calculated by the database server where the underlying data object is stored, while the values for virtual attributes are calculated by the Versata Logic Server. For information about computed attributes, see “Computed Attribute Details frame” on page 165.

Working with relationships

It is a good idea to define relationships as completely as possible before opening a data model in the Versata Logic Studio. As you define transaction logic and design application user interfaces, you may discover that you need to make changes to relationships. Sum, count, and replicate derivation rules are based on relationships among data objects. Some archetypes for display of data on forms/pages and navigations among forms also are based on relationships. If you have already opened the data model in the Versata Logic Studio, use the Transaction Logic Designer to add, delete, or change keys for the relationships in your data model.

After definition in the Versata Logic Studio, each relationship's definition is represented in the repository as an `.xml` file (`REL_<data_object__name><data_object__name>.xml`). The format of this file conforms to the `Relation.dtd` file included with the product, located in the product installation directory. The `.dtd` file lists all of the nested elements and attributes that define the characteristics of each relationship. Each relationship `.xml` file includes values for these nested elements and attributes. For more information about Versata Logic Suite `.dtd` and `.xml` files, see the *Reference Guide*.

Note: If you need to make changes to the relationships for query objects, use the Query Object Designer. Relationships for a query object are based on the relationships of its childmost data object. For information, see page 160.

The Versata Logic Studio incorporates relationship information into the scripts and files it generates to deploy data objects in locations available to run-time applications. For instance, when you deploy related data objects to the database server, relationship metadata is deployed at the same time. Data object Java files that are deployed to the Versata Logic Server also include information about data object relationships.

Types of relationships supported

For the purpose of database server enforcement, the Versata Logic Studio presumes that all relationships are equi-joins. You may choose to define join options other than equi-joins if your client application requires it, but be aware that the relationship rules for these joins will be enforced as equi-joins on the database server.

If you use outer joins to ensure that child rows with null foreign keys are retrieved by queries, users will be able to search on blank attributes.

The most common kind of relationship in Versata data models is one-to-many (1:N). For example, one customer can have many orders, one order can have many order items. Versata data models also support many-to-many (N:M) relationships. In addition, the Versata Logic Suite supports Super/Sub relationships, where a super type of object contains subtypes that inherit the behavior of the supertype and extend it.

Many-to-many relationships

A many-to-many relationship (N:M) is a relationship where many rows in one data object can be related to many rows in another. For example, Parts and Suppliers data objects would generally have a many-to-many relationship. Many parts may be provided by many suppliers and many suppliers may provide many parts.

Many-to-many relationships typically are implemented indirectly, through a third data object called a junction data object or intersection data object. Both primary data objects have a direct one-to-many relationship with the junction data object. They do not have a direct, many-to-many relationship with each other. For example, a PartsSuppliers junction data object would provide an indirect many-to-many relationship.

Because of this implementation, you may want to use query objects to build forms or pages based on many-to-many relationships.

For example, to build a form or page that displays one supplier and all the parts it sells, start with a form that displays one supplier from the Suppliers data object. Then define a query object that joins the Parts data object and the PartsSuppliers junction table. The query object selects all parts sold by the current supplier.

Add the query object to the form or page. If you simply add the Parts data object, it will display all parts, not the parts of a selected supplier.

Type hierarchies

In addition to one-to-many and many-to-many relationships, Versata Logic Suite data models also support Super/Sub relationships, where a supertype of an object contains subtypes that inherit the behavior of the supertype and extend it. A Super/Sub relationship, commonly referred to as a type hierarchy, is not a relationship by the traditional relational definition, because it does not join two data objects.

The following is an example of a type hierarchy: the supertype is Employee and the subtypes are Salaried, Hourly, and Commissioned. A supertype can be concrete or abstract. A concrete supertype can contain records that are not members of any subtype. For example, if Employee is concrete, an employee that is neither Salaried, Hourly, nor Commissioned can exist. If Employee is abstract, all employees must be one of the subtypes.

You can define type hierarchies in your data model in one of the following ways:

- **Store with Super.** All data remain in one data object. A type attribute identifies the subtype of each row. For example, the data model could include an Employees data object with a Type attribute, containing a value of S for salaried employees, H for hourly employees, and C for commissioned employees. By definition, some of the other attributes in the row would be NULL. For example, the salary attribute would be NULL for hourly employees because they receive wages, while the HourlyWage attribute would be NULL for the other employees, because they receive salaries.
- **Store Alone.** Common data are stored in a supertype data object and the data specific to each subtype are stored in separate subtype data objects. For example, the data model contains an Employees data object, and also Salaried, Hourly, and Commissioned data objects. No attributes are NULL by definition. This way works well for type hierarchies with concrete supertypes.
- **Store Separate.** A variation of Store Alone where no supertype data object exists. Separate subtype data objects duplicate the supertype definition, and also contain type specific attributes. No attributes are NULL by definition. This way works well for abstract supertypes.

We generally recommend that you implement type hierarchies in Store with Super data objects, but guidelines vary according to circumstances.

Implementing type hierarchies

The generally recommended way to implement type hierarchies in Versata data models is Store with Super. In most cases, Store with Super produces the simplest and best performing design. For example, the sample database includes an EMPLOYEES data object with an EmpType attribute. In this example, an EMPLOYEE is an abstract supertype.

If you use Store Alone rather than Store with Super, you could not easily refer to attributes in the supertype data object except through replication. You would need to create foreign keys that point to multiple data objects, and reused foreign keys can cause errors. Also, form or page generation is more complicated, because you might need to place multiple data objects instead of one on a form or page. Further, using Store Alone requires the join input/output required for data retrieval.

In the following circumstances, you may need to use Store Alone:

- If you want to use indexes to enforce the uniqueness of an attribute that is null for some subtypes. A workaround is to use event code to enforce uniqueness and use Store with Super.
- If your data source has limits on the number of attributes you can define in a row or the total number of bytes a row can contain. In this case, Store with Super can cause your design to exceed these limits.
- If the supertype is abstract and has few attributes or relationships and you rarely need to display subtypes together on forms or pages. In this case, you could use the Store Separate way to create separate data objects for the subtypes, duplicating the few supertype attributes and relationships in each subtype data object.

Guidelines for Store with Super type hierarchies

Review the following guidelines before you implement a type hierarchy as Store with Super in a Versata data model. The Versata Logic Suite sample repository contains a Store with Super type hierarchy in the EMPLOYEES data object.

- Define a type indicator attribute for the supertype data object. Typically you should limit the values for this attribute to those in a Coded Values List. For instructions, see “Defining a coded values list validation rule” on page 234.
- Define subtype specific attributes to allow NULL values. If you want to make an attribute required for one subtype, define a constraint rule.
- To enforce a relationship that is specific to a subtype, define a relationship to the supertype data object, and define a replicate or a sum to limit the relationship to records with the appropriate subtype. You can replicate the type indicator attribute in the related child data object, and add a constraint to this data object that rejects inapplicable types.
- To enforce a constraint that is specific to a subtype, include a check of the type indicator attribute in the constraint definition.

Relationships tab of Transaction Logic Designer

You can add, delete, or modify relationships between data objects on the Relationships tab of the Transaction Logic Designer. You also can define referential integrity rules and relationship-level presentation properties on this tab. For information about these rules, see page 197 and page 194.

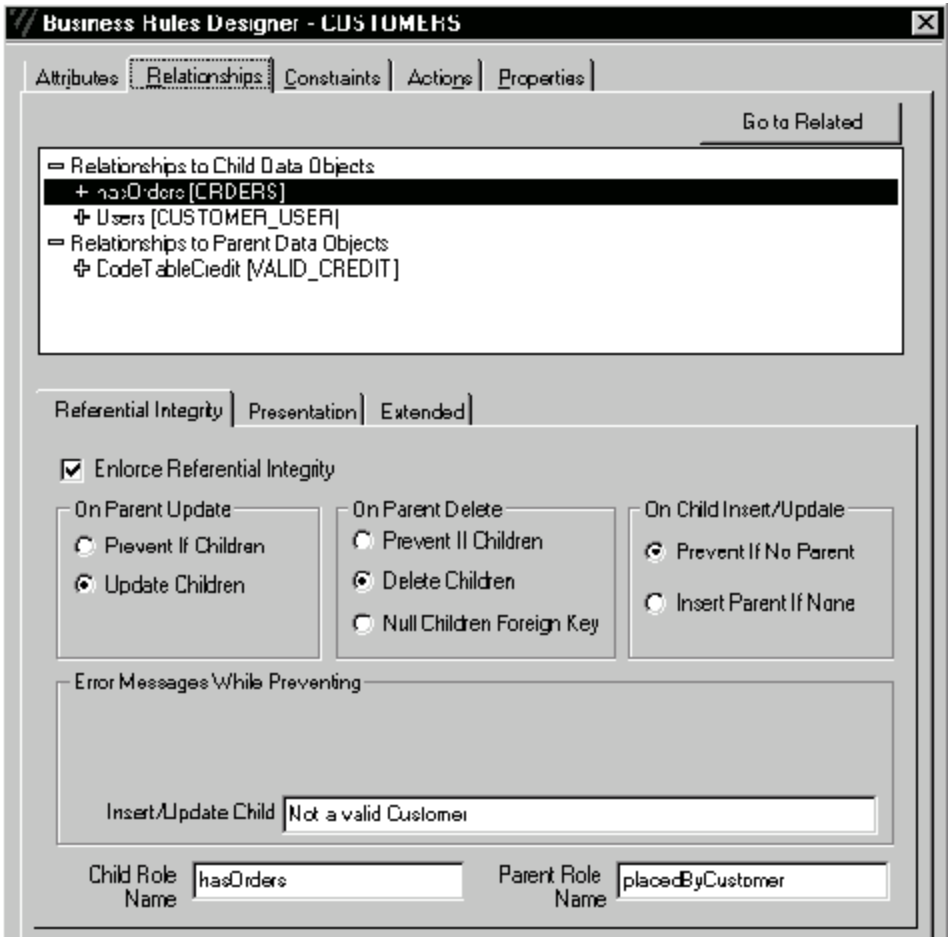

Figure 5 **Transaction Logic Designer Relationships tab**

When the Relationships tab is selected, Add Relationship, Modify Relationship, and Delete Relationship are available from the Edit menu and from the right-click menu for the relationships outline.

The relationships outline lists the parent and child relationships for the selected data object, and lists the primary and foreign keys for each relationship. Click the + sign next to a relationship to view its key(s). Select a relationship from this outline to modify it.

The Extended tab allows you to add relationship properties other than those explicitly specified in the Versata Logic Studio. Extended properties are useful in cases where you plan to add custom Java code for a relationship. Code for these extended properties is generated in the related data objects' Java implementation files. For each extended property, a static string variable is created. To add an extended property, click the Add button and complete the dialog. Then, enter a property value in the grid. To delete an extended property, place the cursor in the grid row for the property and click the Delete button.

Note: All changes to relationships are saved immediately, so there is no need to explicitly save these changes.

The childmost data object is not updated automatically when the relationship between underlying data objects changes. After such a change, review the childmost data object for any affected query objects and modify it as necessary.

A relationship is deleted automatically if both the parent and child data objects are deleted.

Relationship Editor

The Relationship Editor appears when you select a relationship and choose Edit → Modify Relationship, and after you complete the Create Relationship dialog when adding a relationship.

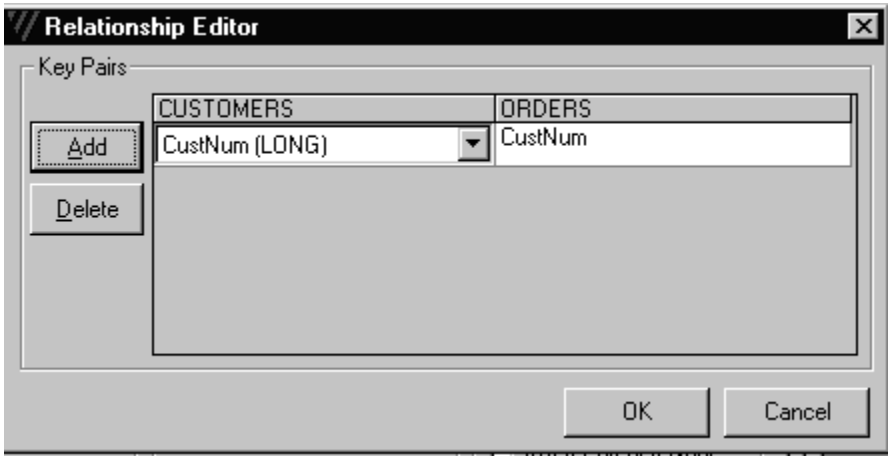

Figure 6 **Relationship Editor**

In the Relationship Editor, you can make the following changes:

- To change keys in an existing key pair, select attributes from the drop-down lists for data objects.
- To add a key pair, click the Add button and select attributes from the drop-down lists.
- To delete a key pair, select a pair and click the Delete button.

Click the OK button to confirm the changes and close the Relationship Editor.

Adding relationships

You can add parent and child relationships for a data object.

To add a relationship for a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and double-click the data object where you want to add a relationship. The Transaction Logic Designer opens.
2. In the Transaction Logic Designer, click the Relationships tab and choose Edit→Add Relationship.
3. In the Create Relationship dialog, select an option button for Parent or Child relationship. In the Related Data Object list box, select the other data object for the relationship. Click the OK button.
4. In the Relationship Editor, click the Add button. From the drop-down lists, select a key attribute for each data object. Repeat to add more key pairs as desired. Click the OK button.
5. On the Relationships tab, enter referential integrity rules and presentation rules for the new relationship.
6. Choose File→Save Transaction Logic.

Note: If you have chosen to enforce referential integrity, the data types for each key pair must be identical. If the data type is Number with a Size of Decimal, then Precision and Scale also must match. The following data types are not supported for keys: Yes/No, Memo, and LongBinary.

You may encounter errors if one key has a Text data type with fixed length sub-type, and the other key has a Text data type with variable length sub-type.

You may encounter data type mismatch errors if you attempt to create a relationship between a reengineered data object and a data object imported with Repository Exchange Manager, as a result of data type remapping that occurs during deployment and reengineering. For example, a Currency attribute that is deployed to DB2 is mapped to Decimal, then reengineered as Decimal.

Adding a relationship from XML

You can directly import an object created outside of the Versata Logic Suite as a relationship, if it can be represented in an `.xml` file. The `.xml` file for the external object must conform to the `Relation.dtd` file. For information about this file, see the *Reference Guide*.

To add a relationship from XML:

1. Review the `Relation.dtd` and the `.xml` file for the external object, to ensure that the `.xml` file contains all values for all elements and attributes required by the .dtd.
2. Revise the `.xml` file as necessary to conform to the `.dtd`. If the `.xml` file includes elements not contained in the `.dtd`, you can make them into Hidden Property elements so they can be maintained in the `.xml` file for the relationship.

3. In the Objects view of the Versata Logic Studio Explorer, right-click the Business Logic folder, or one of its subgroup folders, and choose Add Existing.
4. In the dialog that appears, select the `.xml` file to be added as a relationship.

Deleting relationships

To delete a relationship for a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and double-click the data object where you want to delete a relationship. The Transaction Logic Designer opens.
2. In the Transaction Logic Designer, click the Relationships tab and choose Edit→Delete Relationship.
3. In the Action Choice dialog, click the Yes button to confirm the deletion.

Note: To delete an attribute that is used in a relationship, you must delete the relationship first.

Changing keys for relationships

You can modify a relationship between data objects by changing the attributes used as keys for the relationship.

For information about primary keys, see page 117.

To change keys for a relationship:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and double-click the data object where you want to change keys. The Transaction Logic Designer opens.
2. In the Transaction Logic Designer, click the Relationships tab and choose Edit→Modify Relationship.
3. In the Relationship Editor, you can change keys:
   - To change keys in an existing key pair, select attributes from the drop-down lists for data objects.
   - To add a key pair, click the Add button and select attributes from the drop-down list boxes.
   - To delete a key pair, select a pair and click the Delete button.
4. Click the OK button to confirm the changes and close the Relationship Editor.

Note: Every relationship must have at least one key pair. Do not delete the last key pair for a relationship. Create a new key pair first, or delete the relationship instead. If you have chosen to enforce referential integrity, the data types for each key pair must be identical. If the data type is Number with a Size of Decimal, then Precision and Scale also must match. The following data types are not supported for keys: Yes/No, Memo, and LongBinary.

Working with indexes and primary keys

It is a good idea to define data object indexes as completely as possible before you open your data model in the Versata Logic Studio. As you define declarative business rules, you may discover that you need to make changes to indexes. If you have already opened the data model in the Versata Logic Studio, use the Transaction Logic Designer to add, delete, or make changes to indexes.

Note: As of release 5.5, object naming conventions are enforced for index names. However, invalid index names may exist in repositories created before this release and these names are not validated when data objects are loaded into the repository. For information about naming conventions, see “Naming conventions for data objects and attributes” on page 38.

Primary keys

The Versata Logic Suite does not support data objects without primary keys. Such data objects are treated as "read only" and cannot be updated. For Microsoft SQL Server, the primary key attributes must be NOT NULL. In addition, observe the following guidelines when defining keys:

- Do not use floating point numbers as primary keys. Using floating point numbers for primary keys or foreign keys may cause unpredictable results, depending upon the Java Virtual Machine being used.
- Do not define non-unique indexes on primary keys. When you create indexes, do create them on foreign keys. Note that foreign key indexes are created automatically during the relationship enforcement process.
- Do define a primary key for every data object in the data model. Note that indexes for primary keys are not defined automatically; you must explicitly define each index.
- Do determine the correct data type for attributes that are keys early in your development process. You cannot change the data type of a key in the Versata Logic Suite. If you need to change the data type for an attribute that is a key, you need to drop the key, change the data type, then recreate the key and its index. You also need to review any relationships involving that key and any rules dependent on those relationships, and recreate them if necessary.
- It may be necessary to review primary key indexes’ names and modify them for uniqueness across a database, particularly if they are likely to be truncated. For information about Versata’s truncation rules, see “Naming conventions for data objects and attributes” on page 38.

Index Editor

The Index Editor appears when you click the Add or Modify buttons on the Properties tab, or the Keys/Indexes tab in the Transaction Logic Designer.

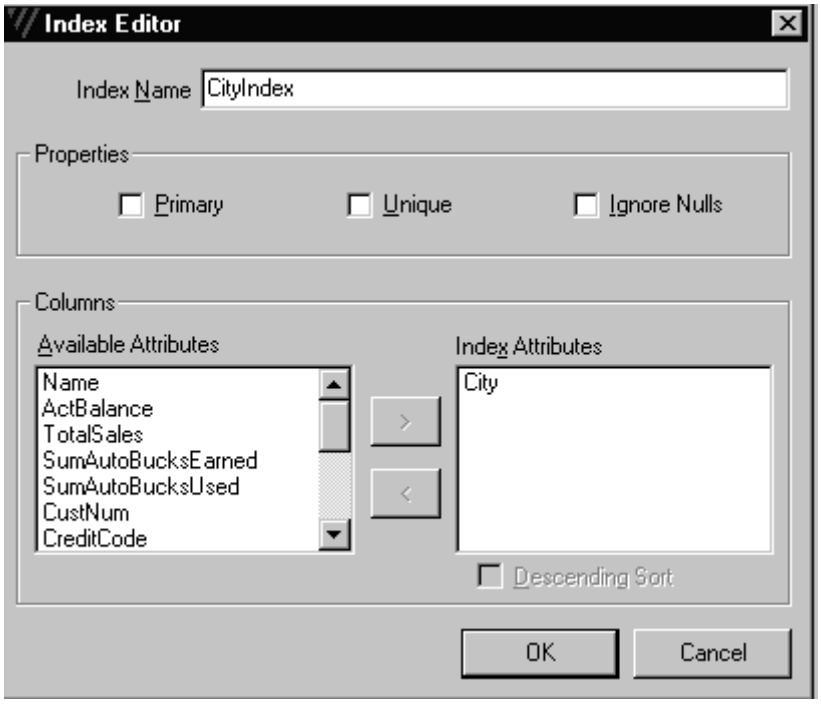

Figure 7 Index Editor

In the Index Editor, you can:

- Enter a name for the index.
- Indicate whether the index is primary, unique, and/or ignores nulls.
- Indicate how the index will be sorted by selecting one or more attributes in the Available Attributes list box and clicking the unfold button to move them to Index Attributes list box.
- Enable the Descending Sort check box. (By default, the sort is ascending.)

Click the OK button to save the additions or modifications.

Note: All changes to indexes are saved immediately, so there is no need to explicitly save these changes.

If a data object has more than ten indexes, it is not currently possible to modify the data object's indexes in the Index Editor. In this case, it is necessary to modify the data object's `.xml` file directly.

Adding indexes

To add an index to a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and double-click the data object where you want to add an index. The Transaction Logic Designer opens.
2. In the Transaction Logic Designer, click the Properties tab and the Keys/Indexes tab.
3. Scroll down in the Transaction Logic Designer window and click the Add button.
4. In the Index Editor, enter a name for the index.
5. Indicate whether the index is primary, unique, and ignores nulls.
6. Indicate how the index will be sorted by selecting one or more attributes in the Available Attributes list box and clicking the unfold button to move them to Index Attributes list box.
7. By default, the sort is ascending. If you would like a descending sort, enable the check box.
8. Click the OK button.

Note: Indexed attributes that you plan to deploy to an Informix database must have a length (size) of less than 255. You cannot deploy to Informix if any indexed attribute has a length greater than or equal to 255.

DB2 UDB does not allow the creation of a unique index on a nullable attribute.

As you create an index, you may encounter errors with unclear, confusing messages. These are Versata internal errors and are not fatal.

Deleting indexes

To delete an index from a data object:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and double-click the data object where you want to delete an index. The Transaction Logic Designer opens.
2. In the Transaction Logic Designer, click the Properties tab and the Keys/Indexes tab.
3. Scroll down in the Transaction Logic Designer window, select an index, and click the Delete button.
4. In the Action Choice dialog, verify that you have selected the correct index to delete, then click the Yes button to continue.

5. If the index is being used to enforce a relationship, you receive a message that it cannot be deleted. Click the OK button to close the dialog.

Changing index definitions

You can modify an index on a data object by changing its name, changing the attributes it uses for sorting, changing the type of sort, indicating whether the index should ignore null values, and indicating whether the index is primary or unique.

To make changes to an index:

1. In the Versata Logic Studio Explorer, expand the Business Objects folder, the Data Objects folder, and double-click the data object where you want to modify an index. The Transaction Logic Designer opens.
2. In the Transaction Logic Designer, click the Properties tab and the Keys/Indexes tab.
3. In the table on the Keys/Indexes tab, double-click an Index record. The Index Editor opens.
4. Make changes to the index. You can:
   - Enter a new name.
   - Enable or disable the Primary, Unique, and Ignore Nulls options.
   - Change the attributes the index uses to sort data object records by selecting attributes and clicking the unfold buttons to move them between the Available Attributes and Index Attributes lists.
   - Indicate that the index should sort records in descending order by selecting an attribute in the Index Attributes box and enabling the Descending Sort option.
5. Click the OK button.

CHAPTER 4

Deploying Data Models

Chapter overview

Read this chapter to understand how to complete tasks to deploy a data model from the Versata Logic Studio to a supported RDBMS.

This chapter includes the following:

- "Deployment overview" on page 123, describes the deployment process.
- "Setting up a system DSN" on page 124, explains how to set a data source name for the RDBMS database(s) where Versata data objects will be deployed.
- "Deploying a data model to a database server" on page 126, provides step-by-step instructions for copying Versata data model information to one or more RDBMS databases. This chapter explains how to deploy directly from the Server Manager wizard, as well as how to generate and use deployment scripts.

Deployment overview

Deployment is the process of setting up the components of an application so that the application can be run by users. The files that compose each of the parts must be placed in locations available to users.

Versata Logic Studio-generated applications are built for a three-tier environment. In this environment, the data model is deployed to a database server, such as Microsoft SQL Server™, Oracle®, Sybase®, Informix®, or DB2 UDB.

To deploy your data model, use the Server Manager wizard to install the repository's data model and optionally, test data, onto a database server. Also, you can generate script files instead of deploying to the database server, and then later run the scripts to install the data model on the database server.

Generally, you should deploy the data model and transfer test data to the database server before you deploy business objects containing transaction logic to the Versata Logic Server. You must deploy the data model and the business objects in order to test an application and review its user interface.

If you need to retarget your application(s) to run against a different type of database server, the Versata Logic Studio automates the retargeting process. You can deploy the repository data model to the other database server, then redeploy transaction logic to the Versata Logic Server. Also, you can check connection properties for the redeployed data objects in the Versata Logic Server Console. Once connection properties are set to the correct database server, you simply run the application.

The Versata Logic Studio also enables you to deploy individual data objects to different database servers so that applications can run against multiple data sources simultaneously. For information about this type of situation, see "Deploying to multiple databases" on page 142.

Note: This chapter includes information about all RDBMSs supported by the Versata Logic Suite. Every release of the Versata Logic Suite may not support every RDBMS discussed in this chapter. For information about the RDBMSs supported by this release, see the *Getting Started Guide*.

Setting up a system DSN

Before you can deploy a data model to a database server, you need to set up an ODBC data source name (DSN) for the database server. The Server Manager uses the DSN to connect to the database server. Versata Logic Studio-generated applications and the Versata Logic Server may also use the DSN for database server connectivity.

A DSN stores information about how to connect to a specified data provider. A user DSN is visible only to the user who sets it up and can be used only on the current machine. A system DSN is visible to all users on the machine, including Windows NT services.

It is a good idea to give a meaningful name to each DSN. For example, you could set up a DSN called `Sample` for the database server where you deploy the Versata Logic Suite sample data model.

To set up a system DSN:

1. Choose Start → Settings → Control Panel.
2. In Windows NT, double-click ODBC.
3. In the ODBC Data Source Administrator dialog, click the System DSN tab.
4. On the tab, click the Add button.
5. In the Create New Data Source dialog, select a supported driver for the type of database server you are using, then click the Finish button. A dialog for the selected driver appears.
   - **Microsoft SQL Server or Sybase**. Enter a name for the DSN (such as `Sample`) and the server name of the server for the data model deployment. The server name is usually the name of the machine as well. Review fields on subsequent tabs. You can leave the defaults for all of them. You may want to specify a default database other than master if you want to set up the DSN for that specific database.
   - **Oracle**. Enter a DSN and the `server name:`SQL*Net connect string—for example, `t:database_server:orcl`.
     - **SQL*Net version 1**. The SQL*Net connect string has the format: `<protocol>:<host name>:<Oracle SID>`.
     - **SQL*Net version 2**. The SQL*Net connect string has the format: `<name of Oracle service>`.

- **Informix®**. Enter a DSN, the name of the host machine on which the Informix server is running, the name of the TCP service identifying the port on which the server is listening (usually turbo, but it can be set differently with the `setnet32.exe` utility), the name of the Informix server as defined with the `setnet32.exe` utility, and the name of the protocol type. Also, enable the Insert Cursors option.
- **DB2 UDB**. You can select an alias for a database from the drop-down list or click the Add Database button to use a different database as the target of the DSN. If you choose Add Database, you need to complete the fields in the Add Database Smart Guide. The Smart Guide allows you to use a database access profile provided by an administrator, search the network for a database, or manually configure a connection to the database. On the Smart Guide tabs, you need to select a target database, enter an alias for the database, and register the database as an ODBC data source. If you are manually configuring the connection, you need to enter additional information, including the communications protocol to be used for the connection and the type of operating system on the machine where the database server is located.

6. Click the OK button to create the DSN.
7. After you have completed the dialogs to add the DSN, review the list of DSNs to verify that a new DSN has been created, and click the OK button.

Note: If the Create New Data Source dialog does not list a supported driver for your database server, you may need to install a new driver.

Deploying a data model to a database server

To deploy a data model to a database server, use the Server Manager part of the Deployment Manager. For information about Server Manager dialogs and options, see "Working with the Server Manager" on page 128.

Note: During deployment, you are asked to enter a user name and password for the database where the data model is to be deployed. If you are deploying to DB2, you may encounter an issue where you are asked to enter this information for every object. This problem has not been encountered with DB2 7.1 Fixpack 3 and later.

To deploy a data model to a database server:

1. Start the Deployment Manager (choose Managers → Deployment Manager, click the Deployment Manager toolbar button, or press F8).
2. In the Choose Deployment Target dialog, select Database Server deployment.
3. Choose the type of database server where the data model will be deployed, and click Next.
4. Choose whether you would like the Server Manager to automatically select changed objects for deployment. If this is your first data model deployment, do not enable this option. If you have previously deployed the data model, it is a good idea to enable this option.
   - If you do not enable Auto-Select, click the Next button.
   - If you enable Auto-Select, a dialog appears where you need to confirm this choice, enter database connection information, and click OK.

     Another dialog appears where you need to choose a DSN for the server. You may need to set up a new DSN. For instructions, see "Setting up a system DSN" on page 124. Choose a DSN and click the OK button. A login dialog for the database server may appear. Enter required information, and click the OK button.

     The Server Manager connects to the database server and compares the tables in the database with the data objects in the repository data model, selecting any repository data objects that are different.

   Note: If a previous data model deployment resulted in problems or was not complete, the Auto-Select option will not work properly.
5. If you enabled Auto-Select, review the selected objects for deployment and make changes as desired. If you did not enable Auto-Select, move objects that you want to deploy from the All Objects list box to Selected Objects. Then, click the Next button.
6. Select the Deploy to the Server option, and click Next. (For information about creating scripts rather than deploying directly to the server, see "Generating deployment scripts instead of deploying to server" on page 135.)

7. Choose whether to deploy test data to the server. Any repository data object containing data has a `.csv` file. You should transfer test data if you want to transfer stored and displayed values for coded values lists.
8. Choose whether to enforce referential integrity on the database server.

   Generally you do not need to enforce referential integrity on the database server. The Versata Logic Server always enforces referential integrity. In some cases, if you are expecting direct updates to the database, you may enable this option in order to ensure that referential integrity is enforced for these direct updates.
9. Choose whether to grant all permissions to public. In a development environment, you can enable this option to save time. For more information about this option, see “Granting permissions manually” on page 138. Click Next.
10. Choose whether to drop and recreate the data model on the database server or make incremental updates to it. The Synchronize option requires that you connect to the database server. If a connection is not possible, select the Drop and Recreate option. Click Next.
11. Choose whether to generate quoted identifiers. For more information about this option, see “Generating quoted identifiers” on page 139. Click Next.
12. Review the choices displayed in the final dialog, and click Finish.
13. If necessary, select a DSN. You may need to set up a new DSN. For instructions, see “Setting up a system DSN” on page 124.
14. If necessary, log on to the database server.
15. Review the contents in the Server Deployment Preview dialog and continue the deployment.

Caution

For deployments to Informix®, the user performing deployment must have `connect` and `resource` permissions. If you plan to synchronize the repository with the existing database, the user performing deployment also must have `dba` permission on the database. In addition, it is a good idea to use the `setnet32.exe` utility to set the `DELIMIDENT` environment variable to “y” before you begin your deployment. You must set this variable if you plan to generate quoted identifiers.

For deployments to DB2 UDB, the user performing deployment must have a login with no more than eight characters and must have the following DB2 administrator privileges: connect database, create tables, create schemas implicitly, and database administrator authority.

Note: If an error is encountered during data model deployment, the Server Manager attempts to roll back any transactions committed to the database server up to that point. However, the rollback may not be complete. If the deployment encounters errors, your best recourse is to fix any problems in the repository and redeploy with the Drop and Recreate option.

Indexed attributes that you plan to deploy to an Informix database must have a length (size) of less than 255. You cannot deploy to Informix if any attributes have indexed attributes greater than or equal to 255.

During data model deployment, data objects that are not abstracted from the RDBMS are ignored.

In redeployments to Microsoft SQL Server, changes of attributes' Value Required properties may not be reflected accurately. The Server Manager attempts to alter the table to add a not null constraint, which is different from the behavior that occurs when a column is changed from null to not null in the SQL Server Enterprise Manager.

Working with the Server Manager

Use the Server Manager to deploy the application's data model to a database server, such as Microsoft SQL Server, ORACLE, Sybase, Informix, or DB2 Universal Database.

To start the Server Manager, start the Deployment Manager by choosing Managers → Deployment Manager, clicking the Deployment Manager toolbar button, or pressing F8. In the Choose Deployment Target dialog, choose Database Server.

The following sections describe the dialogs that appear to lead you through the data model deployment process.

Server Manager Introduction dialog

In this dialog, choose the type of RDBMS (relational database management system) where the data model will be deployed. Versata System-generated applications currently run against the following RDBMSs:

- Microsoft SQL Server
- Oracle
- Sybase
- Informix
- DB2 Universal Database

Note: Every release of the Versata Logic Suite may not support every RDBMS discussed in this section. For information about the RDBMSs supported by this release, see the *Getting Started Guide*.

Connect for Auto Selection dialog

If you have previously deployed the data model, you can enable the Server Manager to connect to the database server, and to select automatically any data objects that have changed since the last deployment. The selected objects are to be deployed this time.

If you have not previously deployed the data model, or if you currently cannot connect with the database server, do not enable Auto Select.

When you enable Auto Select, a dialog appears for you to log in to the database server. The dialog varies, depending on the type of RDBMS to which you are deploying. You also must specify the data source name (DSN) for the server to which you are connecting.

Auto-select Changed Data Objects

When you choose Auto-select in the Connect for Auto Selection dialog, an Auto-select Data Objects dialog appears in which you must log in to the appropriate database server. The dialog has different formats depending upon the target database server for deployment. Complete this dialog, then click the Yes button to confirm that you want auto selection to proceed. Next, you will need to choose a DSN.

Auto-select Changed Data Objects for Oracle dialog

If you selected Oracle as your database server type, the dialog has the following fields:

Server Type	Oracle should be selected from this drop-down list.
Schema/User	Enter the user's schema/user name to use to deploy the objects.

Auto-select Changed Data Objects for SQL Server or Sybase dialog

If you selected SQL Server or Sybase as your database server type, the dialog has the following fields:

Server Type	SQL Server or Sybase should be selected from this drop-down list.
Login Name	Enter the user login name to use to deploy the objects.
Database	The database name. The designation Default means that you will connect to the default database.

Auto-select Changed Data Objects for Informix dialog

If you selected Informix as your database server type, the dialog has the following fields:

Server Type	Informix should be selected from this drop-down list.
Schema/User	Enter the user's schema/user name to use to deploy the objects.

Auto-select Changed Data Objects for DB2 UDB dialog

If you selected DB2 Universal Database as your database server type, the dialog has the following fields:

Server Type	DB2UDB should be selected from this drop-down list.
Schema/User	Enter the user's schema/user name to use to deploy the objects.

Select Data Objects dialog

In this dialog, you can select data objects to be deployed.

- If you have enabled Auto Select in the previous dialog, any data objects that have changed since the last deployment automatically are placed in the Selected Objects list box.
- To move a single data object between the two list boxes, select the data object and click the > or < button. (Use the SHIFT or CTRL key to select multiple objects.)
- To move all data objects in one list box to the other list box, click the >> or << button.

Note: To prevent database anomalies, the Versata Logic Studio automatically orders the data objects in the Selected Objects list box. Parent data objects are listed before their children data objects, to prevent referential integrity errors if you transfer test data after you deploy the rules.

Deploy to Server or Scripts dialog

In this dialog, choose whether to deploy selected data objects directly to the database server or to generate deployment scripts.

- **Deploy To The Server**. Copies information about the selected data objects from the repository directly to the database server by creating deployment scripts and running them against the server.
- **Generate Scripts for DDL**. Creates deployment script files and places them in a `<repository name>_<database_server_name>_Scripts` subdirectory in the directory where the repository is located. You can run these scripts against the database server using a tool such as `isql`.
  - Use this option to review and debug deployment scripts or if the database server currently is unavailable.
  - When you use this option, the script includes `insert` statements that create data rows for any coded values lists in your data model.

What to Deploy dialog

In this dialog, you can make decisions about what is deployed to the database server.

- **Transfer Test Data from Repository to Server**. Copies any existing data (`.csv`) files in the repository to the database server. This option is available only if you selected the Deploy to the Server option in the previous dialog. Use this option for first-time deployments and test deployments. Also, you should elect to transfer test data if you want to transfer stored and display values for coded values lists.
- **Enforce Referential Integrity on DBMS**. Copies referential integrity rules to the database server. Other rules are deployed to the Versata Logic Server, but referential integrity rules are considered part of the data model so that you have the option of deploying them to the database server. (You can review these rules on the Relationships tab of the Transaction Logic Designer.)

  Generally you do not need to enforce referential integrity in the database. The Versata Logic Server always enforces referential integrity. In some cases, if you are expecting direct updates to the database, you may enable this option in order to ensure that referential integrity is enforced for these direct updates.

  DB2 Universal Database supports different referential integrity options from RI options available in the Versata Logic Suite. If you used Reengineering Manager to import a DB2 UDB data model into the Versata Logic Suite, the DB2 UDB referential integrity information is stored in metadata and can be deployed back to the database server. However, the Versata Logic Suite does not distinguish between `No Action` and `Restricted`, and uses the default `No Action` for deployment.

Note: You may encounter problems if you attempt to deploy referential integrity information to Oracle. Cascade constraints are not being generated correctly.

- **Grant All Permissions to Public**. Allows full access to all objects on the database server to all users. Use this option for test deployments to avoid spending a lot of time assigning permissions to specific users. For more information, see “Granting permissions manually” on page 138.

Data Model Deploy Options dialog

In this dialog, choose how the deployment will alter the data model on the database server.

- **Synchronize the Repository with the Server**. Checks for changes that have been made to the data model in the repository and updates the corresponding data objects on the database server.

  Use the Synchronize option when you do not want to replace the entire data model but just want to add to or edit one or more data objects in the data model. For example, if you added attributes to a data object in the repository, choose Synchronize to add those same attributes to the data object on the server.

  Do not use Synchronize if you are transferring data.

- **Drop and Recreate**. Creates a new data model on the database server by deleting the existing data objects on the server and replacing them with the data objects from the repository.

  Use this option when you first deploy or if you need to replace the data model on the server. Use this option with caution if you are unsure, as it overwrites the data model currently on the server.

  The Drop and Recreate option applies only to data objects located on the server. Drop and Recreate does not apply to extended data objects, which cannot be deployed to the server.

Note: For deployments to Informix where you plan to synchronize the repository with the existing database, the user performing deployment must have `dba` permissions on the database.

If you are deploying to SQL Server or Sybase after adding any Autonumber attributes to data objects, you may encounter errors like the following: `"Incorrect syntax near the keyword 'Identity'"`. For more information about issues with the Identity property, see "Identity Columns" on page 54.

Be careful if you synchronize repositories that have quoted identifiers enabled. Newly created tables on the database resulting from the synchronizing may not be created properly with quoted identifiers. Also, an error has been observed where the synchronize fails after addition of a primary key on DB2 7.2.

Configuration Options dialog

In this dialog, choose whether to enable the Generate Quoted Identifiers option.

When you enable this option, double quotation marks are placed around identifiers in deployment scripts before the scripts are run.

- **Oracle**. Quoted identifiers indicate that the names of database objects are case-sensitive. For more information, see "Quoted identifiers for Oracle" on page 139.

- **Microsoft SQL Server or Sybase**. Quoted identifiers allow you to avoid the special handling of reserved words so that database objects may be created with those words as names. You also can use spaces and underbars in names if you use quoted identifiers. For more information, "Quoted identifiers for Microsoft SQL Server and Sybase" on page 139.
- **Informix**. If you enable this option, you must use the `setnet32.exe` utility to set the `DELIMIDENT` environment variable to `y` before you begin your deployment. In general, it is a good idea to set this variable to `y` even if you do not plan to generate quoted identifiers. For more information, see "Quoted identifiers for Informix" on page 140.
- **DB2 Universal Database**. Quoted identifiers indicate that the names of database objects are case-sensitive. For more information, see "Quoted identifiers for DB2 Universal Database" on page 140.

Note: Only use quoted identifiers if they are necessary. Once you start using quoted identifiers in the Versata Logic Suite, you must always use them; and some front-end database server tools do not support quoted identifiers.

Ready to Deploy dialog

This dialog lists the choices that you have made in previous Server Manager dialogs. Review these choices to ensure that you have selected the right deployment options, then click the Finish button to begin the data model deployment.

Server Deployment Preview dialog

The Server Deployment Preview dialog displays the contents of the Deployment Log file (`ServerDeploy.log`). Review your deployment choices before connecting to the server and deploying. If there are errors in the log file, cancel the deployment and correct the problems. If problems are encountered after you deploy, fix the repository and redeploy with the Drop and Recreate option.

Data model deployment files

Files generated by the Server Manager are placed under the directory where the Versata Logic Suite repository is located, in a subdirectory named `<repository>_<database_server_type>_SCRIPTS\`, where `<repository>` is the name of your repository and `<database_server_type>` is `SQLSERVER`, `ORACLE`, `SYBASE11`, `INFORMIX`, or `DB2UDB`.

Script files provided with the Versata Logic Studio are located in the root installation directory.

The following table describes data model deployment script files.

File	Creation	Description
<SERVER>_server_setup.sql where <SERVER> is sql, oracle, db2, or informix	Provided with the Versata Logic Studio	There is one copy for Oracle, one for Informix, one for Microsoft SQL Server or Sybase, and one for DB2 Universal Database. Run this script once on each database where you plan to generate scripts for deployment, and then run the scripts—rather than using the Server Manager. This script must be run before running any deployment-generated scripts.
ServerDeploy.log	Generated	Contains logged information about the deployment. For more information, see "Deployment log file" on page 134.
DDL.sql	Generated	Contains data object definitions. If you deploy to SQL scripts, this file contains the complete DDL for the deployed data objects. If you deploy directly to a server, it contains only the DDL for the last data object deployed.
RULEDDL.sql	Generated	If you choose to enforce referential integrity in the database server, contains check constraints.

Deployment log file

When you deploy data objects, warnings and informational messages generated by the system are displayed in the Server Deployment Preview dialog. Then they are saved as the Server Manager generates a deployment log file (ServerDeploy.log) in the \<Repository_name>\<repository>_<database_server_type>_SCRIPTS\ subdirectory.

The log lists all warnings and informational messages generated by the system, based on a comparison of data objects in the repository and data objects on server. Use a text editor such as Microsoft Notepad or WordPad to view the files.

The contents of the deployment log file initially appear during the deployment.

Generating deployment scripts instead of deploying to server

When you deploy a data model using the Server Manager, you can choose to deploy to scripts instead of deploying to the server. Use this option if you cannot connect to the database server, if you want to review deployment information before it is implemented or if you do not have required permissions for database updates.

To generate data model deployment scripts:

1. Start the Deployment Manager (choose Managers → Deployment Manager, or click the Deployment Manager toolbar button, or press F8).
2. In the Choose Deployment Target dialog, select Database Server deployment.
3. Choose the type of database server where the data model will be deployed, and click Next.
4. Choose whether you would like the Server Manager to automatically select changed objects for deployment. If this is your first data model deployment, do not enable this option. If you have previously deployed the data model, it is a good idea to enable this option.
   - If you do not enable Auto-Select, click Next.
   - If you enable Auto-Select, a dialog appears where you need to confirm this choice, enter database connection information, and click OK.

     Another dialog appears where you need to choose a DSN for the server. You may need to set up a new DSN. For instructions, see "Setting up a system DSN" on page 124. Choose a DSN and click OK. A logon dialog for the database server may appear. Enter required information, and click OK.

     The Server Manager connects to the database server and compares the tables in the database with the data objects in the repository data model, selecting any repository data objects that are different.
5. If you enabled Auto-Select, review the selected objects for deployment and make changes as desired. If you did not enable Auto-Select, move objects from the All Objects list box to Selected Objects. Then, click Next.
6. Select the Generate Scripts for DDL option, and click Next.
7. Choose whether to enforce referential integrity on the database server. (Generally, you do not need to enforce referential integrity on the database server. The Versata Logic Server always enforces referential integrity. In some cases, if you are expecting direct updates to the database, you may enable this option in order to ensure that referential integrity is enforced for these direct updates.)
8. Choose whether to grant all permissions to public. In a development environment, you can enable this option to save time. For more information about this option, see "Granting permissions manually" on page 138. Click Next.

9. Choose whether to drop and recreate the data model on the database server or make incremental updates to it. The Synchronize option requires you to connect to the database server. If a connection is not possible, select the Drop and Recreate option. Click Next.
10. Choose whether to generate quoted identifiers. For more information about this option, see "Generating quoted identifiers" on page 139. Click Next.
11. Review the choices displayed in the final dialog, and click Finish.
12. Review information in the Status Preview for Script Generation dialog, and click the Deploy button to continue.
13. Review the generated script files. For information about these files, see "Data model deployment files" on page 133.
14. If you want to use the scripts to deploy to the database server, see "Running deployment scripts" on page 136.

Running deployment scripts

Once you have used the Server Manager to generate deployment scripts, you can run the scripts against your database server, using a tool such as `isql` or SQL*Plus. The scripts must be run in the correct order.

Note that generated files are found in the scripts subdirectory and files provided with the Versata Logic Studio are found in the root directory where the product is installed.

The procedures vary according to the type of database server you are running.

Running deployment scripts against Oracle

When you are running the scripts against Oracle, run them in the following order:

- If you have not run them previously or if you are running the scripts against a server to which you have never deployed, run:
  `oracle_server_setup.sql`
- Run the following scripts each time you deploy:
  - `ddl.sql`
  - `ruleddl.sql`

Running deployment scripts against Microsoft SQL Server or Sybase

When you are running the scripts against Microsoft SQL Server or Sybase, run them in the following order:

- If you have not run them previously or if you are running the scripts against a server to which you have never deployed, run:
  `sql_server_setup.sql`
- Run the following scripts each time you deploy:
  - `ddl.sql`
  - `ruleddl.sql`

Note: If you are using quoted identifiers, either add the following line to the `ddl.sql` file before executing the ddl:

```
set quoted_identifier on
```

or set the quoted identifier flag on in the Query Analyzer (sqlplus) tool.

Running deployment scripts against Informix

When you are running the scripts against an Informix database, run them in the following order:

- If you have not run them previously or if you are running the scripts against a server to which you have never deployed, run:
  `informix_server_setup.sql`
- Run the following scripts each time you deploy:
  - `ddl.sql`
  - `ruleddl.sql`

Running deployment scripts against DB2 Universal Database

When you are running the scripts against a DB2 UDB database, run them in the following order:

- If you have not run them previously or if you are running the scripts against a server to which you have never deployed, run:
  `db2_server_setup.sql`
- Run the following scripts each time you deploy:
  - `ddl.sql`
  - `ruleddl.sql`

Granting permissions manually

If you want to grant permissions on data objects created during deployment, you can either use the Grant ALL Permissions to Public option in the What to Deploy dialog, or you can create a script to manually grant specific permissions to groups defined in your database. When the Grant ALL Permissions to Public check box is enabled, all users included in the database server group Public are granted full permissions to the data objects deployed to the database server. Statements are generated to grant all access permissions to the group public. Use this option with care, especially in a production environment.

If you want to manually grant permissions to certain groups instead of using this option, you should review the following guidelines for your database server before you write the script to grant permissions.

Permissions for Microsoft SQL Server and Sybase

For Microsoft SQL Server and Sybase, you must grant permissions on tables and stored procedures. Use `isql` or a similar tool to query your `SYSOBJECTS` table for the tables and stored procedures owned by the user ID that you used to deploy, and grant permissions on them to the appropriate users.

Note that you can create a group that includes all the appropriate users, and then grant permission to that group. For more information, see the documentation provided with your database server.

Permissions for Oracle

In Oracle, you must grant permissions on tables and packages. Use SQL*Plus or a similar tool to query your data dictionary for objects owned by the user ID that you used to deploy and include them in a grant statement.

Generating quoted identifiers

You can produce quoted identifiers by enabling the Generate Quoted Identifiers option during data model deployment. When you generate quoted identifiers during deployment to a database server, the Versata Logic Studio inserts quotes around the names of attributes and data objects.

The effect of quoted identifiers varies according to the type of database server.

Caution: Once you start using quoted identifiers, you have to use them throughout the Versata Logic Studio to refer to the named data objects; and all applications and tools which reference the data objects must use quoted identifiers.

Note: If objects to be deployed contain custom code that dynamically builds `Where` clauses, and you are changing whether quoted identifiers are generated during deployments, you will need to modify this custom code between deployments. Once you have deployed using quoted identifiers, it is best to continue deploying in this manner, in order to avoid problems running applications.

The following sections explain the effects of quoted identifiers on different types of servers.

Quoted identifiers for Oracle

By default, Oracle generates all identifiers in upper case. To preserve mixed case identifiers, you must use quoted identifiers. Quoted identifiers also allow the use of spaces and underbars in identifiers. Note that some database server tools do not support quoted identifiers.

When you deploy to the database server and generate quoted identifiers, the Versata Logic Studio inserts quotes around the names of attributes and data objects. If you enable the Generate Quoted Identifiers option when deploying to Oracle, occurrences of identifiers in generated SQL statements are enclosed in double quotation marks before the data objects and rules are deployed. The quotation marks ensure that the case of the identifiers is preserved. For example, the data object named Customers will be named “Customers” after deployment.

If you disable the Generate Quoted Identifiers option in Oracle, occurrences of identifiers in SQL statements are automatically generated in upper case upon deployment.

Quoted identifiers for Microsoft SQL Server and Sybase

For SQL Server, generate quoted identifiers to bypass special handling of reserved words and to allow spaces and underbars. Data objects created with this option can have reserved words in their names. When you deploy to the server and generate quoted identifiers, the Versata Logic Studio inserts quotes around the names of attributes and data objects.

For example, you could name a data object Table (a reserved word), then enable the Generate Quoted Identifiers option, and since occurrences of identifiers are enclosed in double quotation marks, the name “Table” is allowed as a data object name. If you name a data object Table without enabling the Generate Quoted Identifiers option, you receive an error indicating that the data object name is not allowed. If you generate with quoted identifiers, all references using any tool or application must use quotes.

For SQL Server and Sybase, quoted identifiers have nothing to do with case sensitivity. During installation, the server is configured to be either case sensitive or insensitive. Object names are deployed with the case as it is typed in the repository. If the server is configured to be case sensitive, all queries must use the correct case. If the server is configured as case insensitive, case is ignored for all queries.

Refer to your database server documentation for a list of reserved words.

Quoted identifiers for Informix

By default, Informix generates all identifiers in lower case. To preserve mixed case identifiers, you must use quoted identifiers. Quoted identifiers also allow the use of spaces and underbars in identifiers. Some database server tools do not support quoted identifiers. When you deploy to the database server and generate quoted identifiers, the Versata Logic Studio inserts quotes around the names of attributes and data objects.

If you enable the Generate Quoted Identifiers option when deploying to Informix, occurrences of identifiers in generated SQL statements are enclosed in double quotation marks before the data objects and rules are deployed. The quotation marks ensure that the case of the identifiers is preserved. For example, the data object named Customers will be named “Customers” after deployment.

If you disable the Generate Quoted Identifiers option in Informix, occurrences of identifiers in SQL statements are automatically generated in lower case upon deployment.

Note: If you want to enable this option for Informix, you must use the `setnet32.exe` utility to set the `DELIMIDENT` environment variable to `y` before you begin your deployment. In general, it is a good idea to set this variable to `y` even if you do not plan to generate quoted identifiers.

Quoted identifiers for DB2 Universal Database

By default, DB2 UDB generates all identifiers in upper case. To preserve mixed cased identifiers, you must use quoted identifiers. Quoted identifiers also allow the use of spaces and underbars in identifiers. Some database server tools may not support quoted identifiers. When you deploy to the database server and generate quoted identifiers, the Versata Logic Studio inserts quotes around the names of attributes and data objects.

If you enable the Generate Quoted Identifiers option when deploying to DB2 UDB, occurrences of identifiers in generated SQL statements are enclosed in double quotation marks before the data objects and rules are deployed. The quotation marks ensure that the case of the identifiers is preserved. For example, the data object named Customers will be named "Customers" after deployment.

If you disable the Generate Quoted Identifiers option in DB2 UDB, occurrences of identifiers in SQL statements are automatically generated in upper case upon deployment.

Testing the repository for quoted identifiers

We recommend that you test your repository for quoted identifiers before you start building applications and remove the quoted identifiers, if possible. To test the repository, deploy it to the database and look for naming errors. Where they appear, rename the objects with non-quoted names. Or deploy to the target database server without using quoted identifiers. Any invalid names return errors. You can change the invalid identifiers in the repository, and redeploy with the Drop and Recreate option.

If you decide to change whether your repository uses quoted identifiers, redeploy enabling the Drop and Recreate option and changing the Quoted Identifiers option. In addition, you must rebuild any applications in the repository to ensure that they function properly. If you have included additional `Where` clause information in properties sheets for the application, check that the SQL includes the proper identifiers (either quoted identifiers or unquoted identifiers).

Example of quoted identifiers

This portion of a sample `DDL.sql` file displays an example of SQL used to generate quoted identifiers during deployment:

```
CREATE TABLE "WizardDriverHelpLink"(
"TopicID" NUMBER(10,0) NULL ,
"TopicHelpID" NUMBER(10,0) NULL ,
"TopicReady" FLOAT NULL )
```

Data model deployment errors

Errors may occur when you deploy. Some errors can be corrected by redeploying, while others must be fixed manually, either in the repository or on the database server.

If you receive syntax errors in Microsoft SQL Server, redeploy your data model. This frequently fixes the problem.

You may encounter errors stemming from attempts to make changes that cannot be made using the ALTER statement. ALTER has different functionality in Oracle and Informix than it does in SQL Server and Sybase, though there is some overlap:

- ALTER can always add a column.
- ALTER can never drop a column.
- ALTER can never change the name of a column.

In Oracle and Informix, ALTER is quite flexible, and can be used to increase or decrease the width of a character column, increase or decrease the number of digits in a number column, and increase or decrease the number of decimal places in a number column. Note that you can change the data type of a column or decrease its width only if all values in the column are null.

In Microsoft SQL Server and Sybase, ALTER cannot

- Change column name
- Change column length
- Change column data type
- Change column nullability

If there are problems in your data model that cannot be fixed by the ALTER command, you must make the changes manually. Depending on the type of change you want to make, you might need to drop and recreate the table or select all the data out of the table and make any changes. See your database server documentation for information on commands used to change the structure of your tables.

Deploying to multiple databases

You can deploy data objects from a single repository to multiple databases, or schemas. For instance, you might store all your customer information in one database and your employee information in another. You need to run the Server Manager multiple times to deploy to each database separately, logging in to the correct database server each time.

You should deploy to all databases before you deploy business objects to the Versata Logic Server. The deployment database is stored separately for each data object, and is available as a connection property for Versata Logic Server deployment.

Note: If multiple data objects are used to build a single query object, all these data objects must be deployed to the same schema or database.

Example of multiple schema deployment

The following example discusses a fictional Sybase server containing a CUSTOMERS database and an EMPLOYEES database.

The order of deployment would be:

1. Use the Server Manager to deploy the CUSTOMERS data object to the CUSTOMERS database on the database server. If you deploy as JSMITH, the CUSTOMERS data object is created with the fully-qualified name CUSTOMERS.JSMITH.CUSTOMERS.
2. Use the Server Manager to deploy the EMPLOYEES data object to the EMPLOYEES database on the database server. If you deploy as JSMITH, the EMPLOYEES data object is created with the fully qualified name EMPLOYEES.JSMITH.EMPLOYEES.
3. Use the Versata Logic Server Deployment wizard to deploy the CUSTOMERS and EMPLOYEES business objects to the Versata Logic Server.

   The rules are generated to reference CUSTOMERS.JSMITH.CUSTOMERS and EMPLOYEES.JSMITH.EMPLOYEES.

It is a good idea to check connection properties for data objects in the Versata Logic Server Console after you have completed data model and transaction logic deployment. These connection properties indicate the database server where each data object is physically stored. For information about viewing and modifying database server mappings, see the *Administrator Guide*.

CHAPTER 5

Working with Query Objects

Chapter overview

Read this chapter to get an introduction to how query objects are used by Versata Logic Suite applications and to understand how to create and modify query objects in the Versata Logic Studio.

This chapter includes the following:

- "Query object overview" on page 147, provides an introduction to query objects, including how to use them in applications, the definition of childmost data objects, and design guidelines.
- "Adding query objects" on page 152, explains how to use the New Query Object wizard to create query objects.
- "Modifying query objects" on page 159, explains how to use the Query Object Designer to review and edit query object properties and SQL text.

Query object overview

Query objects are reusable presentation objects that you can use as data sources on multiple applications' forms or pages. Query objects are available to all applications in the repository. During run time, query objects are instantiated as needed by the Versata Logic Server. The Versata Logic Server retrieves data from one or more data objects. A query object is based on a SQL `Select` statement. This statement defines the attribute values retrieved to create the query object.

Query objects provide flexibility in the manner you choose to display data on application forms or pages. Query objects are used as data sources more often than data objects. You can use query objects to select a set of attributes from a data object, to specify the order in which the attributes of a data object appear, or to select attributes from multiple data objects so that they behave as a single data source. As your application models are dependent on queries, you may need to go through an iterative process of defining query objects as you design your applications.

Query object definition

The Versata Logic Studio provides a graphical wizard that allows you to create query objects without being a SQL expert. The New Query Object wizard generates SQL text based on your responses. For information, see "Adding query objects" on page 152. After you have used the wizard to create query objects, you can use the Query Object Designer to modify them further. For information, see page 160.

After definition, each query object is represented in the repository as an `.xml` file (`<query_object_name>.xml`). The format of this file conforms to the `QueryObject.dtd` file included with the product, located in the product installation directory. The `.dtd` file lists all of the nested elements and attributes that define the characteristics of each query object. Each query object `.xml` file includes values for these nested elements and attributes. For more information about the Versata Logic Suite `.dtd` and `.xml` files, see the *Reference Guide*.

Query object deployment

To display query object data at run-time in applications, you need to build query objects into usable files that can be copied to the application servers. (Query object information does not need to be copied to the database server because query objects are not physically stored.)

The Versata Logic Studio also provides menu options to build and compile each query object definition into files that run on the application server(s). The next step is to deploy these files to a development Versata Logic Server on IBM WebSphere Application Server Single Server Edition for testing purposes. The Versata Logic Studio includes a Versata Logic Server Deployment wizard that handles this deployment. You set a deployment property in the Query Object Designer to indicate whether to deploy query objects as Enterprise JavaBeans (EJBs) or simply as Java class files. After they have been tested in the development environment, you can copy files to a production Versata Logic Server on IBM WebSphere Application Server Advanced Edition. For more information about building and deploying data objects, see "Building and Deploying Business Objects" on page 255.

When to use query objects in applications

There are several reasons you normally would use query objects in building applications:

- **Displaying parent data**. For example, the Versata Logic Suite sample repository contains an OrderItemJoinPart query object. When this query object is selected to be a data source on a form or page, data from two data objects (ORDERITEM and PART) are displayed in one location. Along with ORDERITEM attributes, attributes of the parent data object PART, such as the PART Name, can also be displayed.
- **Optimizing data transfer**. If a data object has 200 attributes and the end user only needs to see 30, using a query object to project only those 30 attributes results in smaller messages to get data and improves performance.
- **Optimizing virtual attribute performance**. Imagine that you build a form or page on a data object, and then add several non-persistent attributes for transaction logic. Applications built on this data object require the Versata Logic Server to instantiate these attributes when the applications runs, even though the client has no need for the data. So, an application can suddenly degrade in performance. However, if you use a query object to declare the specific set of attributes your application needs, then the Versata Logic Server does not have to instantiate the non-persistent attributes. So, performance remains fast, even after new non-persistent attributes are added.

If you build applications on data objects, then later realize you need to use query objects for reasons of displaying parent data or optimization, the Versata Logic Studio provides a property you can use to retarget a form or page RecordSource from a data object to a query object (with presentation design only).

Childmost data object

The Versata Logic Studio uses the childmost data object in a query object to determine the key features of the query object, including:

- Relationships
- Properties
- Whether you can insert, update, or delete rows in data objects returned by the query object

For a query object with attributes from only one data object, that data object is the childmost data object. For a query object with attributes from one or more related data objects, the data object that is lowest in the relationship hierarchy or has no children in the query object is the childmost data object. The data from the childmost data object are unique in every row of the query object's result set. Generally, any insert to a query object results in an insert to its childmost data object.

For example, the sample query object OrderItemJoinPart includes attributes from ORDERITEM and PART data objects. ORDERITEM is a child of PART, which makes it the childmost data object. Child objects are on the many side in a one-to-many relationship.

If the data objects in the query object are not joined along relationships, there may be no childmost data object. If no childmost data object has been set, users can update existing rows from the query object but cannot insert new rows.

Design the query objects so that their childmost data objects have the relationships that you want the query objects to have. Query objects inherit relationships from their childmost data objects.

You can review and modify the childmost data object on the Properties tab of the Query Object Designer. For information, see page 171.

Note: You can set an extended property for a query object in order to enable inserts to an underlying data object that is not the childmost data object. For information, see "Enabling inserts to a parent data object" on page 179.

Query object relationships

The relationships of a query object are inherited from the keys in its childmost data object. If the key for a childmost data object relationship is also in the query object, the query object also has the relationship. If the key is not in the query object, the childmost data object has the relationship but the query object does not.

For example, the OrderItemJoinPart query object in the `SampDB1.xml` repository is a child of the ORDERS data object because the childmost data object ORDERITEMS is a child of ORDERS; and the query object includes the key for the parent-child relationship between ORDERITEM and ORDERS (OrderNum). OrderItemJoinPart is also a child of the PART data object, for the same reason.

Query objects can have relationships with other query objects as well as with data objects. The type of relationship between two query objects must be consistent with the type of relationship between their childmost data objects.

Note: Do not define relationships for query objects in another data modeling tool. The Versata Logic Studio deduces all possible relationships between query objects from the relationships between the base data objects on which the query objects are built. You can review a query object's relationships on the Joins tab of the Query Object Designer. For more information, see page 167.

Query object design guidelines

Follow these recommendations when you define query objects:

- If you are using an existing relational database as your data model, define the query objects after you convert the database to a Versata Logic Suite repository.
- Define only `Select` query objects that do not use any parameters. Query objects with unions and Cross-Tabs do not appear in the Versata Logic Studio Explorer.
- Do not use queries based on queries. If you want to use these, you must open the query object in the Query Object Designer, delete the SQL text that appears, and re-enter the subquery in the SQL Text attribute.
- Use only the following functions: `MIN`, `MAX`, `AVG`, `COUNT`, and `SUM`. If a query uses other functions, the SQL text is left empty and you must enter it manually in the Query Object Designer.
- When you define aliases for attributes and data objects, ensure they are less than 30 characters in length. Use square brackets to enclose problematic text or characters. Do not use double quotes.
- The Versata Logic Studio generates default ODBC SQL text for each query object. You may view and edit the text on the SQL tab of the Query Object Designer.
- In general, use outer joins to build the queries for query objects. The use of outer joins ensures that child rows for optional parent rows can be retrieved. That is, they return all rows from the childmost data object regardless of whether the rows meet the join condition, thereby enabling users to search on `NULL` attributes. If you do not use outer joins, queries will not return rows that do not meet the join condition.

  For example, suppose users are allowed to define orders with no customer. Query objects defined with outer joins would return all orders, whether or not a customer was associated with the order. Query objects defined without outer joins would return only orders with customers.
- Microsoft SQL Server does not support outer joins greater than two levels. A workaround is to use replicates instead.

- Project all required attributes for the data object into which you want to insert records on the application forms or pages. Typically, you only insert rows into the childmost data object. The Versata Logic Studio warns you if you select a childmost data object that does not have all required attributes projected into the query object. In this case, you cannot insert rows into that data object.

  Required attributes may not have `NULL` values. Therefore, make attributes required only if you are sure that users will provide values for them in all queries. A safer alternative is to define default values for required attributes. You can define default derivation rules on the Attributes: Derivation tab of the Transaction Logic Designer. For instructions, see "Defining a derivation rule" on page 232.
- Project foreign keys into query objects so that picks function properly.
- You can replace a data object with a query object as a RecordSource in your application. If the form or page is already customized when you make the replacement, attributes from the data object reappear, but are made Unbound (`DataField = None`). You must delete such attributes from the form or page, or you will be warned at run time that the `DataField` is set to `None`.
- You may encounter errors if you use non-persistent attributes in `Order By`, `Where`, or `Having` clauses forquery objects. Currently, the Query Object Designer does not stop you from using these.
- If a query object contains a non-persistent attribute, then the query object also must contain the primary key attribute(s) for the data objects containing the attributes necessary to calculate the non-persistent attribute. The Query Object Designer currently does not enforce this requirement. If a query object definition fails to meet this requirement, all query object records display the same value in run-time applications.

System validation of query objects

When changes to underlying data objects or their groups occur, query objects may need to be updated. These changes include conversion of pre-5.5 repositories, rebuilds of data objects, moves of data objects to new groups, and certain changes to underlying data objects, their relationships, or their groups. When query objects are loaded after these types of changes, the Versata Logic Studio displays a dialog listing all of the affected query objects, asking whether they should be updated to incorporate the most recent changes. Generally, you should click OK to update the query objects and continue. However, if you have manually customized query text in the Query Object Designer, you may want to click Cancel in order to preserve the current text. In this case, you can make changes manually to reflect changes in underlying data objects without overwriting other query text.

Adding query objects

Use the New Query Object wizard to create new query objects in your repository. It is best to define query objects before developing applications, but you may define query objects at any time in the development cycle. You most likely will discover the need for new query objects as you define your applications.

To create a new query object:

1. In the Versata Logic Studio Explorer, select the Query Objects folder. Right-click and choose New Query Object.
2. Complete the dialogs in the New Query Object wizard.
3. Indicate whether you are importing a query object from another repository or creating a new query object. For details about this dialog, see page 154.
4. If you are creating a new query object, complete the Choose Data Objects for the New Query Object dialog. In this dialog, you indicate the data objects that supply attributes to the query object, define the join condition that limits the records to be retrieved for the query object, and, optionally, define aliases for the included data objects. For more details about this dialog, see page 154.
5. Complete the Choose Attributes for the Query Object dialog. In this dialog, you select attributes from the included data objects to be in the query object. You also have the option of defining expressions for computed attributes in the query object. Computed attributes are not physically stored in the database; their values are computed by the database from the values of other attributes that are physically stored. You also can define aliases for included attributes and functions used to aggregate values for attributes. For more details about this dialog, see page 156.
6. Complete the Specify Where/Order By Clause for the Query Object dialog. In this dialog, you can enter criteria to restrict or sort records returned for the query object. If you do not want to enter these criteria, click the Next button to leave the dialog blank and continue. For more details about this dialog, see page 157.
7. If you defined an aggregate for one or more attributes included in the query object, complete the Specify Having/Group By Clause for the Query Object dialog. The `Having` clause, which defines criteria to restrict records, is optional. The `Group By` clause groups records into sets according to attribute values. You have the option of reordering the attributes used for grouping. For more details about this dialog, see page 157.
8. Complete the Finished dialog. In this dialog, you select the childmost data object and add any description or comment information. You also have options to add the `Distinct` keyword to your query and to use a custom superclass to build the query object. For more details about this dialog, see page 158.

New Query Object wizard

The New Query Object wizard leads you through the process of adding a query object to a repository. You point and click in its dialogs and it generates the SQL text for the query object so that you avoid making syntactical errors.

Figure 8 **New Query Object wizard**

The New Query Object wizard includes the following dialogs:

- Welcome to the Query Object Wizard
- Choose Data Objects for the New Query Object
- Choose Attributes for the Query Object

- Specify Where/Order By Clause for the Query Object
- Specify Having/Group By Clause for the Query Object
- Finished

Welcome to the New Query Object Wizard

In this dialog, you indicate how you want to create a new query object in the repository. Choose one of the following:

- **Create**. Continue with the wizard to define characteristics of a new query object. When you choose this option and click the Next button, the Choose Data Objects for the New Query Object dialog opens, where you can begin defining the query object.
- **Import**. Copy an existing query object from another repository into this one. When you choose this option and click the Next button, an Import From dialog opens, where you can browse for the query object's `.xml` file so it can be copied to the repository.

Choose Data Objects for the New Query Object

In this dialog, you indicate the data objects that will supply attribute values for the new query object.

- **Show Data Objects frame**. This frame allows you to indicate whether to display all data objects in the Available Data Objects list box or only those related to the data object(s) in the Selected Data Objects list box. The Related option becomes available after you have moved a data object to the Selected list box.
- **Available and Selected Data Objects List Boxes**. To choose a data object, select it in the Available Data Objects list box and click the > button to move it to the Selected Data Objects list box. You can then choose the Related option in the Show Data Objects frame if desired.

  To choose another data object, select it in the Available Data Objects list box ,and click the > button to move it to the Selected Data Objects list box. The Select Joins dialog opens. Complete this dialog.
- **Alias**. If desired, add an alias for the data object for more description or conciseness. To enable entry of an alias, select a data object in the Selected Data Objects list box.
- **Edit Joins**. Click this button if you want to make changes to the join(s) for the data objects. The Define Joins for the Selected Data Objects dialog opens.

Select Joins dialog

This dialog opens when you choose more than one data object to supply attributes to a query object. In this dialog, you select the attributes whose values will be compared to join records from the data objects. Joins for query objects are based on attributes that are keys for data object relationships; therefore, this dialog lists the relationships between the chosen data objects and the key pair for each relationship.

In many cases, only one relationship exists between the chosen data objects. In these cases, this relationship is checked and you simply click the OK button to accept it. In cases where multiple relationships exist between the chosen data objects, select a relationship from the list so that a check appears next to it, then click the OK button.

Define Joins for the Selected Data Objects

Use this dialog to add, modify, or delete a join condition for a query object.

- To add a join in the top drop-down lists, choose data objects to be joined. For each data object, choose an attribute to be included in the join condition. Then, choose an option button to define the type of join, indicating the records to be retrieved from the referenced data objects to populate the query object.
  - **Equal**. An equal join joins on matching values for the specified attributes, returning only records that satisfy the join condition.
  - **Left**. A left join is a type of outer join. This type of join returns all records from both data objects that satisfy the join condition plus all records from the first-named data object.
  - **Right**. A right join is also a type of outer join. It returns all records from both data objects that satisfy the join condition plus all records from the second-named data object.

  Generally, it is a good idea to define an outer join that includes all records from the childmost data object. For information about the concept of childmost data object, see page 149.

  Click the Add button to add the join condition. It appears in the Joins text box in the lower part of the dialog.
- To modify a join, select it in the Joins text box. You can change the attributes and/or the join type. Then click the Modify button. The modified condition appears in the Joins text box.
- To delete a join, select it in the Joins text box and click the OK button.

When you have completed your changes to joins, click the Next button.

Choose Attributes for the Query Object

Use this dialog to indicate the attributes from each selected data object to be included in the query object.

- **Data Object**. From the drop-down list, select one of the data objects designated to supply attributes to the query object. Its attributes are listed in the Available Attributes list box.
- **Attributes**. To choose an existing attribute to be retrieved, select it in the Available Attributes list box, and click the > button to move it to the Selected Attributes list box. You can use the SHIFT or CTRL keys to select multiple attributes. You can click the >> button to select all available attributes.

  After you have moved an attribute to the Selected Attributes list box, you can select it and enter additional characteristics for it including:

  - **Alias**. If desired, add an alias for the attribute, for more description or conciseness.
  - **Aggregates**. You can elect to use an aggregate to retrieve summary values for the attribute rather than all values for each individual record. Aggregates are applied to sets of rows and are generally used along with a `Group By` clause. The drop-down list displays aggregate functions that can be used with each attribute, according to the attribute's data type.
    - **`COUNT`**. Reports the number of records with non-null values for the attribute.
    - **`AVG`**. Reports the average value for the attribute.
    - **`MIN`**. Reports the lowest value for the attribute.
    - **`MAX`**. Reports the highest value for the attribute.
    - **`SUM`**. Reports the total of all values for the attribute.
  - **Computed Attributes**. To create a computed attribute to be retrieved for the query object, click the Computed attributes/Expressions button. A computed attribute's values are calculated based on values of attributes stored in referenced data object(s). When you click the button, the Expression Builder opens. Note that when you define the expression for a computed attribute, no error message is displayed when the attribute's data type is incompatible with the specified SQL function.

When you have finished adding attributes, click the Next button.

Specify Where/Order By Clause for the Query Object

Use this dialog to define the `Where` clause and `Order By` clause included in the `Select` statement for the query object, if any. The `Where` clause limits the records retrieved for the query object to those that meet the specified condition (separate from the join condition). The `Order By` clause sorts the records retrieved for the query object.

- **Where clause**. To enter selection criteria for the `Where` clause, click the browse button to open the Expression Builder.
- **Order By clause**. To define a sort attribute, select it in the Available Attributes list box and click > to move it to the Order By attributes list box. You can use the SHIFT or CTRL keys to select multiple attributes. You can click the >> button to select all available attributes. To change the order of sort attributes, select an attribute and click one of the arrows. To indicate a descending or ascending sort, select an attribute and click the Desc or Asc button. By default, sorts are ascending.

Specify Having/Group By Clause for the Query Object

This dialog appears if you have defined any aggregate functions for query object attributes in the Choose Attributes for the Query Object dialog. Use this dialog to define the `Having` clause and `Group By` clause included in the `Select` statement for the query object, if any. The `Group By` clause divides records into sets, while aggregate functions produce summary values for each set. The `Having` clause is a `Where` clause for groups; it defines a condition that limits the groups of records to be retrieved for a query object.

- **Group By attributes**. Lists the order of attributes used to group sets of records. The first-listed attribute provides totals while other listed attributes provide subtotals. To change the order of `Group By` attributes, select an attribute and click one of the arrows.
- **Having clause**. To enter selection criteria for the `Having` clause, click the browse button to open the Expression Builder.

When you have finished, click the Next button.

Note: You cannot selectively group totals for a subset of query object attributes. All attribute values or none must be aggregated.

Finished

Use this dialog to provide general information about the query object. It includes the following fields:

- **Name**. Enter a name for the query object.
- **Distinct Rows Only**. Enable this option to add the `Distinct` keyword to the `Select` statement for your query object. You can use the `Distinct` keyword to retrieve only unique values for the attributes included in the query object, eliminating duplicates. Also, you can use the `Distinct` keyword with aggregate functions to include only unique values in the calculation of summaries.
- **Childmost Data Object**. Of the data objects supplying attributes to the query object, the data object that has a parent by no child. The childmost data object is updated when users modify query object records. Select from the drop-down list.

When you have completed these fields, click the Finish button to create the query object in your repository.

Note: After you have created a query object, use the Query Object Designer to view or modify its properties and SQL text. For information about the Query Object Designer, see page 160.

Modifying query objects

The Versata Logic Studio provides a graphical editor to review and modify properties of query objects. This editor, the Query Object Designer, allows you to make changes without being a SQL expert. The Versata Logic Studio generates changes to SQL text based on your modifications. The Query Object Designer displays SQL text for the query object so that you have the option of editing it directly. The designer also provides a validate functionality. You can use this functionality to test your SQL against the Versata Logic Server and ensure that the query object is instantiated correctly.

To modify a query object:

1. In the Versata Logic Studio Explorer, double-click the query object to open it in the Query Object Designer.
2. Choose the appropriate tab in the designer and make changes. For details about tabs to use for different tasks, see “Query Object Designer” on page 160. For instructions for specific tasks, see page 173.
3. When you have completed the changes, click the Save toolbar button.

After you have saved the changes, you can validate the new SQL text for the query object. For instructions, see page 177.

Query Object Designer

Use the Query Object Designer to modify query objects.

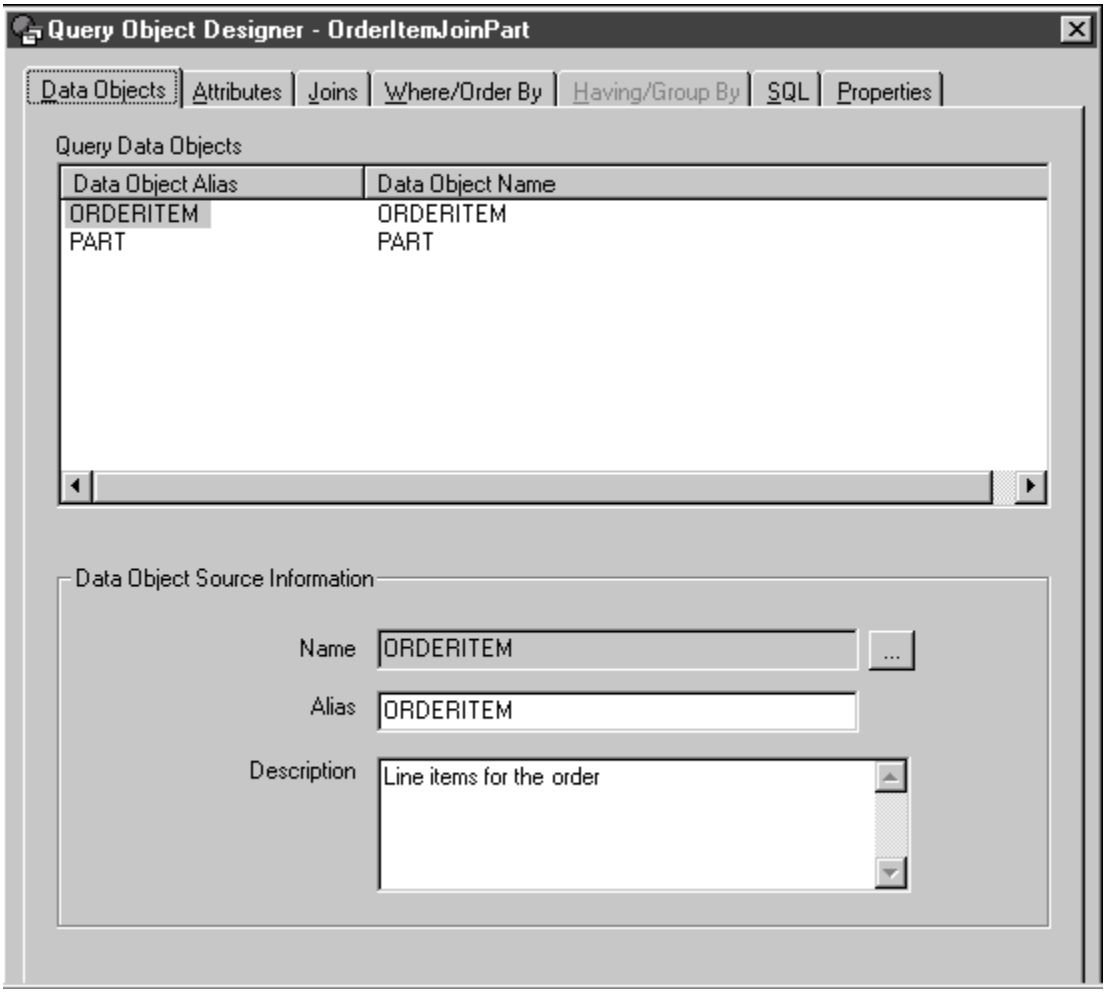

Figure 9 **Query Object Designer**

Note that the Query Object Designer does not provide an implicit save, so any changes you make there are not saved until you choose File → Save, or click the Save button.

- On the Data Objects tab, you can:
  - Add or delete a data object supplying attributes to the query object.
  - Define an alias for a data object.
  - Provide a description of a data object.

  For more information, see "Data Objects tab" on page 162 and "Modifying underlying data objects for a query object" on page 173.
- On the Attributes tab, you can:
  - Add or delete attributes to be included in the query object. These attributes may exist in an underlying data object or may be computed from values of attributes in an underlying data object.
  - Define an alias for an attribute.
  - Provide a description for an attribute.
  - Define a formula expression for a computed attribute.
  - Define a function to aggregate values for an attribute.

  For more information, see "Attributes tab" on page 164 and "Modifying attributes for a query object" on page 174.
- On the Joins tab, you can:
  - Add, delete, or modify the join condition for selecting records to be in the query object.
  - Review relationships between the data objects underlying the query object.

  For more information, see "Joins tab" on page 167 and "Working with joins" on page 175.
- On the Where/Order By tab, you can:
  - Define a selection condition to limit the records included in the query object.
  - Indicate one or more attributes to use to sort query object records.

  For more information, see "Where/Order By tab" on page 168 and "Adding selection and sort criteria for query object records" on page 176.
- On the Having/Group By tab, you can:
  - Define a selection condition to limit the records included in a query object containing aggregates.
  - Indicate the attributes to use to group records for aggregated functions in a query object.

  For more information, see "Having/Group By tab" on page 169 and "Adding selection and sort criteria for query object records" on page 176.

- On the SQL tab, you can:
  - View and modify generated SQL text for the query object. You can change text to conform to a different SQL dialect or to customize in some way. For more information, see "SQL tab" on page 169.
  - Validate the SQL text against the Versata Logic Server and database server to ensure the query object is created as expected. For more information, see "Validating query object syntax" on page 177.
- On the Properties tab, you can:
  - Define a custom superclass for the query object to provide it with specialized methods.
  - Define a childmost data object for the query object.
  - Define the query object to include distinct rows only.
  - Indicate whether to deploy attribute-level security information to the Versata Logic Server.
  - Set the deployment property for the query object indicating whether it should be deployed as an EJB or a Java class file.
  - Provide description or comment information about the query object.

  For more information, see "Properties tab" on page 171.

Data Objects tab

This tab includes information about the data object(s) that supply attributes to the query object. It includes the following:

- **Query Data Objects grid**. This grid lists the alias and name for each data object that supplies attributes to the query object.

  To add or delete a data object to supply attributes, right-click in the grid and choose Add Data Object or Delete Data Object. These options are also available from the Edit menu. If adding, the Add Data Object dialog opens. For information about this dialog, see 163.

  To review or edit information about a data object, select its alias in the Query Data Objects grid. You can then make changes in the Data Object Source Information frame.
- **Data Object Source Information frame**
  - **Name**. The name of the data object. To enter a different data object to supply attributes to the query object, click the browse button, then select a data object from the Choose Data Object dialog. For information about this dialog, see 163.
  - **Alias**. By default, the alias is the same as the data object name. You can change it to be more descriptive or more concise. To change the alias, type in the new alias.
  - **Description**. Optional details about the data object. To add or change, type in the information.

To save the new or renamed data object, click the Save button or choose File → Save Query Object. If this change causes any query attributes to become invalid, the Query Object Validation Log appears. For information about this log, see page163.

When you click the OK button in this dialog, the new data object is listed in the Query Data Objects grid. If you do not want to save the changes to the data object, close the Query Object Validation Log and close the Query Object Designer without saving.

Add Data Object dialog

Use this dialog to add a data object that supplies attributes to a query object.

Choose an option button to display either all data objects or only data objects that are related to the data object(s) currently referenced by the query object.

To add a data object, select it in the Available Data Objects list box and click the > button to move it to the Selected Data Objects list box. The Select Joins dialog opens. Complete this dialog. For information about this dialog, see page 155. Then, if desired, add an alias. Click the OK button to complete the addition of the data object.

To modify information for the newly added data object, select it in the Query Data Objects grid. You can then edit the alias or description.

Choose Data Object dialog

This dialog lists all data objects in the repository. You can choose a data object from this list to supply attribute(s) to a query object.

To choose a data object, select it from the list and click the OK button.

Query Object Validation Log

The Versata Logic Studio includes a Query Object Validator. This validator runs when you attempt to save a modified query object. If the validator encounters invalid attributes, the Query Object Validation Log appears before the modified query object is saved. This dialog displays any attributes included in the query object that do not exist in the referenced data objects. SQL cannot be generated for an invalid query object. To avoid errors, you must either remove these attributes or reference a different data object that includes them.

To save the changes to the query object, click the OK button in this dialog. Be sure to go to the Attributes tab and remove any problematic attributes.

To avoid saving the changes to the query object, close the Query Object Validation Log and close the Query Object Designer without saving.

A copy of the Query Object Validation Log is saved in the `<repository>_JavaFiles\Components` folder as `<query_object_name>.log`.

Choose Attribute dialog

This dialog lists all attributes in the source data object. You can choose an attribute from this list to be included in the query object.

To choose an attribute, select it from the list and click the OK button.

Attributes tab

This tab provides information about the attributes from which data is retrieved for the query object. It includes the following:

- **Query Object Attributes grid**. This grid lists the name for each attribute included in the query object.

  To add or delete an attribute, right-click in the grid, and choose Add Attribute or Delete Attribute. These options are also available from the Edit menu. If adding, the Add Attribute dialog opens. For information about this dialog, see page 166.

  To review or edit information about an attribute, select it in the grid. You can then make changes to any of the following objects:

- **Alias**. By default, the alias is the same as the attribute name. You can change it to be more descriptive or more concise. To change the alias, type in the new alias.
- **Derivation Type**. Indicates whether an attribute is computed or not. A value of `(None)` indicates that the attribute is physically stored in the referenced data object. A value of `Formula` indicates that the attribute's values are calculated by the database server based on values of attributes stored in the referenced data object(s).
- **Aggregation Type**. Indicates whether to use an aggregate to retrieve values for an attribute. Aggregates are functions you can use to get summary values. Aggregates are applied to sets of rows and are generally used along with a `Group By` clause. The drop-down list displays aggregate functions that can be used with each attribute, according to the attribute's data type.
  - **`COUNT`**. Reports the number of records with non-null values for the attribute
  - **`AVG`**. Reports the average value for the attribute.
  - **`MIN`**. Reports the lowest value for the attribute.
  - **`MAX`**. Reports the highest value for the attribute.
  - **`SUM`**. Reports the total of all values for the attribute.

If the selected attribute is physically stored in a referenced data object, more information is displayed in the Query Object Attribute Source Information frame.

If the selected attribute is computed, more information is displayed in the Computed Attribute Details frame.

Query Object Attribute Source Information frame

This frame is included on the Attributes tab of the Query Object Designer when the attribute selected in the grid is physically stored in a data object. You can view and modify the following:

- **Source Data Object**. The name of the data object supplying this attribute. To enter a different data object to supply this attribute to the query object, click the browse button, then select a data object from the Choose Data Object dialog.

  To save the new or renamed data object, click the Save button or choose File Save Query Object. If this change causes any attributes to become invalid, the Query Object Validation Log appears.
- **Source Attribute**. The name of the attribute included in the query object. To choose a different attribute from the selected source data object, click the browse button. The Choose Attribute dialog opens, displaying all attributes in the current source data object. Select an attribute and click the OK button.
- **Description**. Optional details about the attribute. To add or change, type in the information.

Computed Attribute Details frame

This frame is included on the Attributes tab of the Query Object Designer when the attribute selected in the grid is computed rather than physically stored in a referenced data object. You can view and modify the following:

- **Expression**. The formula used to derive a value for the computed attribute, consisting of attributes from the referenced data objects, operators, and/or SQL functions. To review or modify details about this formula, click the browse button to open the Expression Builder. For information about the Expression Builder, see page 166.
- **Data Type**. The nature of the data in the computed attribute, determining how the bits representing the attribute values are stored. A drop-down list provides a list of available data types. Text, Number, and Date/Time types require the definition of a sub type. Text types also require the definition of a size. Defaults are provided. The data type is used to determine presentation properties for the attribute. Note that no error message is displayed when the selected data type is incompatible with the specified SQL function.
- **Description**. Optional details about the computation. To add or change, type in the information.

Add Attribute dialog

Use this dialog to add either an attribute that is physically stored in the referenced data object or an attribute whose values are computed based on values of attributes stored in the referenced data object.

- To add a physically stored attribute, first select a referenced data object from the drop-down list, then select the attribute in the Available Attributes list box and click the > button to move it to the Selected Data Objects list box. Then, if desired, modify the alias.
- To add a computed attribute, click the Computed attributes button. Complete the fields in the Expression Builder.

  Click the OK button to complete the addition. To modify information for the newly added attribute, select it in the Query Attributes grid.

Query Object Expression Builder

Complete this dialog to:

- Define a computed attribute to be included in a query object. A computed attribute's values are calculated by the database server based on values of attributes stored in referenced data object(s).
- Define a `Where` clause or `Having` clause that limits the records retrieved for the query object.

Note: Some fields do not apply for both purposes, and are grayed out when not applicable.

The following fields apply only to computed attribute definitions:

- **Attribute Alias**. Name for the computed attribute.
- **Data Type**. The nature of the data in the computed attribute, determining how the bits representing the attribute values are stored. A drop-down list provides a list of available data types. Text, Number, and Date/Time types require the definition of a sub type. Text types also require the definition of a size. Defaults are provided. The data type is used to determine presentation properties for the attribute.

The following fields apply to computed attribute, `Where` clause, and `Having` clause definitions:

- **(Formula) Expression**. The formula used to derive values for the computed attribute, or the condition used to limit records. Use the following objects to construct an expression:
  - **Operator buttons**. To include an operator in the expression, place the cursor in the Expression text box, and click an operator.
  - **Attributes from the referenced data objects**. Attributes are listed under the data objects in which they are physically stored. Attributes included in the query object are listed as #Query attributes. To include an attribute in the expression, place the cursor in the Expression text box and double-click the attribute.

- **Functions**. To include a function in the expression, place the cursor in the Expression text box and double-click the function. Note that only SQL functions can be used in formula expressions for computed attributes.

You can click the Undo button to remove the last entry in the Expression field.

To save, click the OK button. For computed attributes only, if you want to add another computed attribute, click the New button, save the defined computed attribute, then complete the Expression Builder again to define another one.

Note: No validation is performed for `Where` clauses, `Having` clauses, computed attribute formula expressions, or data type compatibility with SQL function.

Joins tab

This tab includes information about the join condition used to retrieve records from two referenced data objects to populate a query object.

The join condition lets you retrieve and manipulate data from more than one data object in a single `Select` statement. You define the join condition by specifying an attribute from each data object whose values can be compared. The Versata Logic Suite query objects' joins generally are based on relationship key pairs and are equijoins—selecting records where values for the two join attributes match. The join condition is in the `Where` clause of the `Select` statement.

The Joins tab includes the following information:

- **Query Object Joins**. This panel diagrams the joined data objects. Click the + sign to view the attributes used in the join condition.

  To add a join, right-click in the grid and choose Add Join. The Add Join dialog opens. For information, see page 167. To modify or delete a join, select a join condition and right-click. If modifying, the Modify Join dialog opens. For information, see page 168. These options are also available from the Edit menu.
- **Relationships**. This panel lists all of the relationships between the referenced data objects and other data objects in the repository. Expand the relationship to view its key pair.

Add Join dialog

Use this dialog to add a join condition for a query object.

- **Data objects**. In the top drop-down lists, choose data objects to be joined.
- **Attributes**. For each data object, choose an attribute to be included in the join condition.

- **Type of join**. Choose an option button to define the type of join, indicating the records to be retrieved from the referenced data objects to populate the query object.
  - **Equal**. An equal join joins on matching values for the specified attributes, returning only records that satisfy the join condition.
  - **Left**. A left join is a type of outer join. This type of join returns all records from both data objects that satisfy the join condition plus all records from the first-named data object.
  - **Right**. A right join is also a type of outer join. It returns all records from both data objects that satisfy the join condition plus all records from the second-named data object.

Generally, it is a good idea to define an outer join that includes all records from the childmost data object.

Click the OK button to add the join condition.

Modify Join dialog

Use this dialog to modify a join condition for a query object.

- **Data objects**. The top drop-down lists display the data objects to be joined.
- **Attributes**. For each data object, you can change the attribute to be included in the join condition.
- **Type of join**. You can change the type of join.
  - **Equal**. An equal join joins on matching values for the specified attributes, returning only records that satisfy the join condition.
  - **Left**. A left join is a type of outer join. This type of join returns all records from both data objects that satisfy the join condition plus all records from the first-named data object.
  - **Right**. A right join is also a type of outer join. It returns all records from both data objects that satisfy the join condition plus all records from the second-named data object.

Click the OK button to confirm the modification of the join condition.

Where/Order By tab

This tab provides information about the `Where` clause and `Order By` clause included in the `Select` statement for the query object, if any. The `Where` clause limits records retrieved for the query object to those that meet the specified condition (separate from the join condition). The `Order By` clause sorts records retrieved for the query object.

This tab includes the following frames:

- **Selection Condition**. Displays the condition that must be met by retrieved records. To add or modify this expression, click the browse button to open the Expression Builder.
- **Sort Order**. Lists the attribute whose values are used to sort records. To define a sort attribute, select it in the Query Attributes list box and click > to move it to the Order By attributes list box. You can define multiple attributes to use in the sort. To change the order of sort attributes, select an attribute, and click one of the arrows. To indicate a descending or ascending sort, select an attribute, and click the Desc or Asc button. By default, sorts are ascending.

Having/Group By tab

This tab provides information about the `Having` clause and `Group By` clause included in the `Select` statement for the query object, if any. The `Group By` clause is used with aggregate functions. The `Group By` clause divides records into sets, while aggregate functions produce summary values for each set. The `Having` clause is a `Where` clause for groups; it defines a condition that limits the groups of records to be retrieved for a query object.

This tab includes the following frames:

- **Having Condition**. Displays the condition that must be met by retrieved groups of records. To add or modify this expression, click the browse button to open the Expression Builder.
- **Group By Order**. Lists the order of attributes used to group sets of records. The first-listed attribute provides totals while other listed attributes provide subtotals. To change the order of group by attributes, select an attribute, and click one of the arrows.

Note: You can define aggregate functions on the Attributes tab of the Query Object Designer. For information, see page 164.

SQL tab

This tab displays the SQL text used to instantiate the query object for your run-time applications, and allows you to test whether the SQL correctly generates the expected query object. This tab includes the following:

- **Run-time SQL**. Displays the SQL text to be generated based on your choices in the New Query Object wizard or the Query Object Designer. You have the option of manually editing the text here to customize it. Click the Reset button to return to the default generated SQL. Custom SQL should be the last change ever made to a query object, as changes that regenerate the query object will cause the custom SQL to be overwritten with generated SQL. For information about generated SQL text, see “Database and schema references in SQL text” on page 178.
  - **SQL Dialect**. Select from the drop-down list to change the syntax to be database-specific. The dialect is not important unless the query object includes outer joins. For examples of outer join syntax, see page 170.

Note: If you change the driver at run time, SQL text is not modified accordingly. You need to manually change this text here, then redeploy the query object to the Versata Logic Server, in order to modify run-time SQL text. If you do not do this, SQL errors occur in the run-time application.

SQL generation for Informix and ODBC dialects is not supported for queries containing mixed inner and outer joins.

- **Reset**. This button is enabled if you make any changes to the Run-time SQL text. Click it to return the text to the original, generated text.
- **Show Default**. This button is enabled if you make any changes to the Run-time SQL text. Click it to open the Default SQL dialog, which allows you to view the default query object SQL text for different dialects.

- **Test/Validate**. You can validate the query object by executing the SQL against a Versata Logic Server and a physical data source to verify that the query object is instantiated as expected. Enter the following:
  - **Username**. Login for the Versata Logic Server.
  - **Password**. Password for the Versata Logic Server.
  - **VLS Server**. Name of the Versata Logic Server.
  - **Data Server**. The name of the data server in the Versata Logic Server Console that represents the database server to which you are connecting.
  - **Max Rows**. Maximum number of records to be returned for the query object.
  - **Test SQL**. Click this button to execute the test.

Example SQL dialects for outer joins in a query

The following examples of query text illustrate the differences between SQL dialects for outer joins. You can select the correct dialect in the drop-down list on the SQL tab of the Query Object Designer. The dialect is not important unless the query includes outer joins. If the query contains outer joins and you do not select the correct dialect (for example, Oracle Native if you are using Oracle Thin JDBC driver, Sybase Native if you are using Sybase JConnect JDBC driver), syntax errors occur.

ODBC

```
SELECT DEPARTMENT.Name AS "DEPARTMENT.Name", EMPLOYEES.Name AS
"EMPLOYEES.Name", ORDERS.OrderNumber AS OrderNumber FROM {oj
<dbschema>.DEPARTMENT DEPARTMENT RIGHT OUTER JOIN
<dbschema>.EMPLOYEES EMPLOYEES LEFT OUTER JOIN <dbschema>.ORDERS
ORDERS ON EMPLOYEES.EmpID = ORDERS.SalesRepID ON
DEPARTMENT.DeptNum = EMPLOYEES.WorksForDeptNum}   WHERE
((EMPLOYEES.Name) Like '%A%')   ORDER BY DEPARTMENT.DeptNum ASC
```

Oracle Native

```
SELECT DEPARTMENT.Name AS "DEPARTMENT.Name", EMPLOYEES.Name AS
"EMPLOYEES.Name", ORDERS.OrderNumber AS OrderNumber FROM
<dbschema>.DEPARTMENT DEPARTMENT, <dbschema>.EMPLOYEES EMPLOYEES,
<dbschema>.ORDERS ORDERS WHERE EMPLOYEES.EmpID =
ORDERS.SalesRepID (+) AND DEPARTMENT.DeptNum (+) =
EMPLOYEES.WorksForDeptNum AND ((EMPLOYEES.Name) Like '%A%')
ORDER BY DEPARTMENT.DeptNum ASC
```

Sybase Native

```
SELECT DEPARTMENT.Name AS "DEPARTMENT.Name", EMPLOYEES.Name AS
"EMPLOYEES.Name", ORDERS.OrderNumber AS OrderNumber FROM
<dbschema>.DEPARTMENT DEPARTMENT, <dbschema>.EMPLOYEES EMPLOYEES,
<dbschema>.ORDERS ORDERS WHERE EMPLOYEES.EmpID *=
ORDERS.SalesRepID AND DEPARTMENT.DeptNum =*
EMPLOYEES.WorksForDeptNum AND ((EMPLOYEES.Name) Like '%A%') ORDER
BY DEPARTMENT.DeptNum ASC
```

Properties tab

The Properties tab has two subtabs: General and Extended.

General Properties tab

The Properties:General tab provides general information about the query object. It includes the following fields:

- **Superclass for the Java Component**. For each query object, the Versata Logic Studio creates a Java class file, which inherits from a general query object superclass. By default, this superclass is `versata.vls.QueryObject`. If you want to provide additional methods functionality in your query object, you can define another subclass of the default superclass, add methods to it, and enter the name of this new subclass as the query object's superclass in the field on the Properties:General tab of the Query Object Designer.
- **Childmost Data Object**. Of the data objects supplying attributes to the query object, the data object that has a parent but no child. The childmost data object is updated when users modify query object records. Select from the drop-down list. For more information about the concept of childmost data object, see page 149.

Note: The childmost data object is not updated automatically when the relationship between underlying data objects changes. After such a change, review the childmost data object for any affected query objects and modify it as necessary.

- **Distinct Rows Only**. You can use the `Distinct` keyword to retrieve only unique values for the attributes included in the query object, eliminating duplicates. Also, you can use the `Distinct` keyword with aggregate functions to include only unique values in the calculation of summaries. Note that aggregate functions are defined on the Attributes tab. For information, see page 164. Enable this option to add the `Distinct` keyword to the `Select` statement for your query object.
- **Deploy Attribute Security Data**. Enable this option to copy attribute names to the Versata Logic Server in order to enable assignment of privileges at the attribute level. Deployment of this information requires more time. Enable this option only for query objects where you plan to implement attribute-level security.
- **Deploy as EJB Session Bean**. Enable this option to implement the query object as an Enterprise JavaBean (EJB) rather than as a Java class. A query object should be deployed as an EJB when its methods need to be remotely accessible. Deployment as an EJB requires more time. Enable this option only as necessary for remote access.
- **Description**. Optional details about the query object. To add or change, type in the information.
- **Comment**. Optional details about the query object. To add or change, type in the information.

Extended Properties tab

The Properties:Extended tab allows you to add query object properties other than those explicitly specified in the Versata Logic Studio. These extended properties are useful in cases where you plan to add custom Java code to a query object. Code for extended properties is generated in the data object's Java implementation file. For each extended property, a static string variable is created inside the query object's constructor code.

The query object's extended properties perform a similar function to the extended properties for controls or elements on forms or pages in Versata Logic Studio-generated applications: the properties provide additional behavior to query objects. You can add Java code that refers to the value for the variable (extended property), where each different value causes different behavior at run time.

- To add an extended property, click the Add button and complete the dialog. Then, enter a property value in the grid.
- To delete an extended property, place the cursor in the grid row for the property and click the Delete button.

Modifying underlying data objects for a query object

Use the Data Objects tab of the Query Object Designer to make changes to the data objects that supply attributes to a query object.

To modify data objects, in the Versata Logic Studio Explorer, double-click the query object to open the Query Object Designer.

Adding a data object

To add a data object:

1. Right-click in the Query Data Objects grid, and choose Add Data Object.
2. Select a data object in the Available list box and click > to move it to the Selected list box.
3. Select from the list of attribute key pairs to indicate which should be compared when retrieving records for the query objects, and click the OK button.
4. If desired, enter an alias for the new data object.
5. Click the OK button. The data object appears in the grid.

Deleting a data object

To delete a data object:

1. Right-click in the Query Data Objects grid and choose Delete Data Object.
2. Click the Yes button to confirm the deletion.

   Note that any attributes included from the deleted data object are deleted from the query object.

Changing a data object

A common reason for changing the data object to supply attributes to the query object is that the data object has been renamed.

To change a data object:

1. In the Data Object Source Information frame, click the browse button next to the Name field.
2. Select a data object, and click the OK button.
3. Save the change.

4. Review the Query Object Validation Log that lists invalid attributes. Click the OK button to dismiss the log. For information about the Query Object Validation Log, see page 163.
5. Indicate whether you want to make changes to invalid attributes before saving, and make changes to attributes as necessary.

Modifying attributes for a query object

Use the Attributes tab of the Query Object Designer to make changes to the attributes for a query object.

To modify attributes, in the Versata Logic Studio Explorer, double-click the query object to open the Query Object Designer. Click the Attributes tab.

Adding an attribute

You can add an attribute that exists in an underlying data object or add an attribute that is not physically stored, whose value is computed based on values of attributes in an underlying data object.

To add attributes from an underlying data object:

1. Right-click in the Query Object Attributes grid and choose Add Attribute.
2. Select a data object from the drop-down list.
3. Choose one or more attributes from the Available list box and click > to move the attribute(s) to the Selected list box.
4. If desired, enter an alias for the attribute.
5. Click the OK button. The attribute appears in the grid, and its details appear in the Query Object Attribute Source Information frame.
6. If you want to display summary values, select a function from the Aggregation Type drop-down list. For more information, see page 164.

To add a computed attribute:

1. Right-click in the Query Object Attributes grid and choose Add Attribute.
2. Select a data object from the drop-down list.
3. Click the Computed Attributes button.

4. Enter a name for the attribute in the Attribute Alias field. Choose a data type from the drop-down list. Click operator buttons and double-click attributes to define the formula expression used to calculate values for the new attribute. (Note that you can click the + signs to expand data objects and display their attributes.) For more information about the Expression Builder, see page 166.
5. Click the OK button to close the Expression Builder. Then, click the OK button to close the Add Attribute dialog. The attribute appears in the grid and its details appear in the Computed Attribute Details frame.

Note: The Versata Logic Studio may crash after you modify an attribute alias. This problem occurs infrequently. If it occurs, restart the Studio, then retry the change.

Deleting an attribute

To delete an attribute:

1. Right-click in the Query Object Attributes grid, and choose Delete Attribute.
2. Click the OK button to confirm the deletion.

Note: If you delete a query object attribute that no longer exists in an underlying data object, the Versata Logic Studio crashes.

Working with joins

You can view and modify the details of the join conditions on the Joins tab of the Query Object Designer.

To work with joins in the Versata Logic Studio Explorer, double-click the query object to open the Query Object Designer. Click the Joins tab.

Adding a join condition

To add a join condition:

1. Right-click in the Query Object Joins frame, and choose Add Join.
2. Select from the list of attribute key pairs to indicate which should be compared when retrieving records for the query objects, and click the OK button.

Deleting a join condition

To delete a join condition:

1. In the Query Object Joins frame, highlight a join, right-click, and choose Delete Join.
2. Click the Yes button to confirm the deletion.

Caution: If you delete the last join condition in a Query Object drawing from multiple data objects, the resulting query object could generate a "Cartesian resultset"—producing an extremely large result set containing every possible permutation of the joined data objects.

Modifying a join condition

You can change a join condition to be a different type of join (equal, right, or left) or change the attributes whose values are compared to retrieve records. For information about the different types of joins, refer to "Add Join dialog" on page 167, or view the context sensitive help in the wizard.

To modify a join condition:

1. In the Query Object Joins frame, select a join, right-click and choose Modify Join.
2. Make changes in the Modify Join dialog as desired. For more information, see page 168.

Adding selection and sort criteria for query object records

You can view and modify selection and sort criteria for query object records on the Where/Order By and Having/Group By tabs of the Query Object Designer.

- `Where` clauses are expressions that limit the records retrieved for a query object.
- `Order By` clauses are expressions that indicate attribute values to use to sort query object records.
- `Having` clauses are expressions that limit the groups of records retrieved for a query object that displays summary values. These clauses are used when you define aggregates for one or more attributes in a query object.
- `Group By` clauses indicate how to group summary values for query objects with aggregated attributes by designating the order of included attributes.

To add selection or sort criteria:

1. In the Versata Logic Studio Explorer, double-click the query object to open the Query Object Designer. Click the Where/Order By tab or the Having/Group By tab.
2. To add selection criteria, click the browse button near the Selection Condition or Having Condition field and complete the Expression Builder. Click operator buttons and double-click attributes to define the formula expression used to calculate values for the new attribute. (Note that you can click the + signs to expand data objects and display their attributes.) For more information about the Expression Builder, see page 166.
3. Click the OK button to close the Expression Builder.

4. To add sort criteria for individual records, on the Where/Order By tab, select one or more attributes in the Query Attributes list box and click > to move the attribute(s) to the Order By list box. You can click the up or down arrow buttons to change the order of sort attributes, and click the Asc or Desc buttons to indicate the type of sort.
5. To define how to group records to calculate summary values, on the Having/Group By tab, click the up or down arrow buttons to change the order of included attributes.

Validating query object syntax

On the SQL tab of the Query Object Designer, you can review and modify the SQL text generated to instantiate a query object. For information about this tab, see page 169. This tab provides a test button that you can use to retrieve records from the database server for the query object. Use this test function to ensure that no syntax errors occur and that the data you expect is returned for the query object. When you execute the test, the records for the query object appear in a grid for your review.

For information about syntax for references in query object SQL text, see “Database and schema references in SQL text” on page 178.

Note: If you use the New Query Object wizard to define a query object, the wizard generates the SQL text and there should not be any syntax errors. However, you may still need to check to ensure query object data matches your requirements. If you modify SQL text yourself and/or use outer joins in the query object, you need to check for possible syntax errors.

To validate query object syntax:

1. Ensure that the data model has been deployed to a database server. For instructions, see page 126.
2. Ensure that the business objects have been deployed to a Versata Logic Server. For instructions, see page 268.
3. In the Versata Logic Studio Explorer, double-click the query object to open the Query Object Designer. Click the SQL tab.
4. Complete the fields in the Test/Validate frame, entering the following:
   - Your user name for the Versata Logic Server.
   - Your password for the Versata Logic Server.
   - The Versata Logic Server name.
   - The name of the data server in the Versata Logic Server Console that represents the database server to which you are connecting. For information about setting up data servers in the Versata Logic Server Console, see the *Administrator Guide*.
   - The maximum number of query object rows to return from the database server in the test result set. By default this value is set to 100.

5. Click the Test SQL button.

Note: If there is no data in the database when you the TestSQL is executed, a null pointer exception occurs. If this error occurs, verify that the targeted data source contains data, review settings for the data server in the Versata Logic Server Console, and verify that a test connection from the Versata Logic Server Console works properly.

Database and schema references in SQL text

In the SQL text for each query object, the Versata Logic Studio provides `<dbschema>` tags in each data object reference. Each tag is replaced at run time with the database and/or schema to which the data object is currently deployed. This convention provides flexibility, allowing applications to execute queries against varying data sources.

Following is an example of query object SQL text:

```
Select SKILL.SkillName AS SkillName, EMPLOYEESKILL.Rating AS
Rating, EMPLOYEESKILL.SkillNum AS SkillNum, EMPLOYEESKILL.EmpID
AS EmpID FROM <dbschema>.SKILL SKILL, <dbschema>.EMPLOYEESKILL
EMPLOYEESKILL WHERE SKILL.SkillNum = EMPLOYEESKILL.SkillNum
```

The `<dbschema>` is always replaced with the properties supplied for the database and/or schema in the data object's data server in the Versata Logic Server Console.

For example, if a CustomerOrders query object selects from CUSTOMERS and ORDERS data objects that are attached to a data server named MySQL, and the MySQL data server has the properties `db = test` and `schema = orderentry`, then `<dbschema>.CUSTOMERS` and `<dbchema>.ORDERS` in the SQL text would become `test.orderentry.CUSTOMERS` and `test.orderentry.ORDERS`.

The `<dbschema>` tag is detected the first time the query object is executed in a session. The tag values are replaced with the appropriate values once per session as follows:

- The `<dbschema>` tag is replaced by the database value for the data object's data server. Then, this is concatenated with the schema value for the data object's data server.
- If the Database field is blank, no database qualifier is produced for the data object. If the Schema field is also blank, `<dbschema>` is removed from the SQL text and there is no qualifier for the data object name.
- In generated text for query objects, the Versata Logic Studio prefixes all data object references, except those in subqueries, with `<dbschema>`. For any customized SQL, you can use the same tags in front of any data object name references to get the same run-time behavior. You can choose to edit the SQL to either remove the `<dbschema>` prefixes or to change them to hardcoded database or schema names that will not be overridden by Versata Logic Server's run time.

Defining a custom superclass for a query object

By default all query objects are subclassed from `versata.vls.QueryObject`. If you want to provide additional methods to your query object, you can define a subclass of the default superclass, add methods to it, and enter the name of this new subclass as the query object's superclass in the field on the Properties:General tab of the Query Object Designer.

Note: You need to add the Java source file for the custom superclass to the repository. For information, see "Adding files to a repository" on page 308.

Enabling deployment of attribute-level security data for a query object

To define permissions at the attribute-level rather than only at the object level, enable the check box on the Properties:General tab of the Query Object Designer. This setting copies the query object attribute names to the Versata Logic Server when you deploy business objects. Enable this option only when you plan to use attribute-level permissions, as it slows deployment.

Enabling inserts to a parent data object

Generally, an insert to a query object results in an insert to its childmost data object, and inserts to other underlying data objects are not allowed. However, you can set an extended property for a query object in order to enable query object inserts that result in inserts to an underlying data object other than the childmost data object. This underlying data object can be referred to as the parent data object. For example, you can set this property on the sample query object, OrderItemJoinPart, in order to allow users modifying the OrderItemJoinPart RecordSource in an application to cause an insert to the PART data object.

This extended property is called `ParentInsertable`. You can add it to a query object and set its value to `true` on the Query Object Designer's Properties:Extended tab. Formerly, this property was available as an extended security property in the Versata Logic Server Console, that has now been deprecated.

Note: You also have the option of writing custom code to set the `ParentInsertable` property to `true`. The `ParentInsertable(boolean flag)` method of the `VSMetaQuery` interface is available for this purpose. For information, see the Versata Class Libraries Help (`vstudio.hlp`).

When a query object includes a setting of `true` for the `ParentInsertable` property, behavior that is different from the default occurs in the query object's RecordSources in run-time applications. The system inserts a new parent data object record whenever a user modifies any parent fields in a query object RecordSource, unless these changes occur as a result of a pick selection. The user's selection from a pick object modifies the foreign key field and all other parent fields, but as it is based on selection of an existing parent data object record, no insert is required.

Setting the ParentInsertable property in the Query Object Designer

To enable inserts to a query object that insert to a parent data object:

1. In the Versata Logic Studio Explorer, double-click the query object to open it in the Query Object Designer.
2. Click the Properties tab, then the Extended tab.
3. On the Extended tab, click the Add button.
4. In the dialog that appears, enter `ParentInsertable` and click OK. This entry appears in the Property Name column of the extended properties table.
5. In the Property Value column of the table, enter `true`.
6. Click the Save toolbar button.

Notes about the ParentInsertable property

The previous implementation of `ParentInsertable` through a Versata Logic Server Console property provided unconditional parent inserts that potentially caused problems for pick support. The new implementation of `ParentInsertable` provides a conditional logic that allows for full and proper support of picks. This implementation allows end users to either enter parent data fields, in which case a new parent record is inserted, or to select a parent from a pick object, in which case no parent insert occurs and the operation succeeds if rules such as referential integrity are not violated.

The following pseudo code illustrates the conditional logic supported by the current implementation:

```
If pkey for for parent column is not null
   try to insert parent
else {
   if(foreignkey is not null)
     skip parent
   else {
     if(parent pkey is autonumber)
        try to insert parent
     else
        skip parent
}
}
```

The population of the child's foreign key is automatic when the parent's primary key is one autonumber column; the value of this column is copied to the child's foreign key column. If the parent's primary key includes multiple columns, you must add a method to the query object to indicate which column of the child's foreign key is the target for replication of the parent's autonumber column value. The following code provides an example method:

```
   protected boolean isForeignKeyColumn(String tblName,
VSMetaColumn col) {
      boolean result;
      if ( col.getName().equalsIgnoreCase("PTYM_ID_K") )
         result = true;
      else
         result = false;
      System.err.println("isForeignKey for " + col.getName() + ",
rtn: " + result);
      return (result);
   }
```

This method is called for each child attribute. The `col` parameter represents the metadata for the child column. The method returns `true` for only one column, the child column matching that parent autonumber column.

Note: If you still require additional control over updates and inserts resulting from user modifications to query objects, you can override the `save()` method for the query object. You can examine the collection of updates to be started on underlying data objects and add custom code to control them, for example, by removing updates or setting fields.

Disabling resynchronization with a persistent data source

The extended property, `refreshAfterUpdate`, indicates whether or not to resynchronize data between a business object and its persistent data source after a transaction is committed. If this property is set, it overrides the default values for business objects. The possible values are `true` and `false`. By default, query objects are specified as `true`. To override this default, you can add the extended property to a query object and set its value to `false`.

To disable resynchronization of a query object with its persistent data source:

1. In the Versata Logic Studio Explorer, double-click the query object to open it in the Query Object Designer.
2. Click the Properties tab, then the Extended tab.
3. On the Extended tab, click the Add button.
4. In the dialog that appears, enter `refreshAfterUpdate` and click OK. This entry appears in the Property Name column of the extended properties table.
5. In the Property Value column of the table, enter `false`.
6. Click the Save toolbar button.

CHAPTER 6

Understanding Transaction Logic

Chapter overview

This chapter provides an introduction to the business rules that implement transaction logic in the Versata Logic Server. Read this chapter to get an understanding of what transaction logic is and how business rules represent transaction logic. This chapter includes the following:

- "Transaction logic overview" on page 185, introduces declarative business rules, describing the benefits they provide.
- "Types of business rules" on page 189, details the different types of business rules you can define.
- "Transaction logic processing" on page 200, outlines the order of processing for transaction logic at run time.
- "Analyzing business requirements" on page 206, discusses the mapping of business requirements to rules.

This chapter is intended as background to read before you begin defining transaction logic. For procedures for logic definition, see the following:

- For instructions on defining business rules in the Transaction Logic Designer, see "Procedures for defining business rules" on page 232.
- For information about building and compiling Java files for business objects that contain transaction logic, and deploying the objects to the Versata Logic Server and the IBM WebSphere Application Server, see "Building and Deploying Business Objects" on page 255.
- For information about generated business object files and the code they include, see "Understanding Business Object Files" on page 285.
- For information about extending and customizing transaction logic code, see "Extending Business Object Code" on page 321.

Transaction logic overview

Versata Logic Studio allows you to define transaction logic for the objects in your data model. This transaction logic applies to objects across applications. You define this logic in terms of declarative business rules.

What are declarative business rules?

Declarative business rules are simple, unambiguous statements that define the derivation, validation, referential integrity enforcement, and presentation of data (with presentation design only). Business rules defined in the Versata Logic Studio are declarative rather than procedural because you design applications in terms of what the application needs to do, not how it does it.

For example, in designing the appearance of an application user interface, you can specify the data to appear on each form or page, and the navigations, without having to code how the data gets displayed or how users move from one form or page to another. In the same manner, when you define business rules, you design transaction logic in terms of what data values should or should not be, based on calculations or restrictions. You do not need to write code to arrive at these data values; the Versata Logic Studio generates this code based on your declarative definitions.

You define declarative business rules for the data objects in your data model, using graphical tools and simple language in the Transaction Logic Designer. Rules are properties of data, and fire only when data changes state. Business rules are stored in the repository, along with the data model.

The Versata Logic Studio allows you to extend and customize declarative rules in a variety of ways. The Transaction Logic Designer allows you to define action rules that call system-supplied or user-defined Java methods. You also can reference methods in rule expressions. You can use the Code Editor to edit the code generated for rules. Your edits can range from event-handling code to modify the default handling provided for the Versata Logic Suite's exposed server events, to subclassing the Versata Logic Server Classes that provide the building blocks for rules and other business object code.

The biggest issues to solve when you are defining a rule are:

- Identifying which data should own the rule.
- Gathering the data into the right object.
- Figuring out how to change data to fire the rule.

The basic operations provided by declarative business rules are:

- Replication
- Formulation

- Aggregation
- Constraints

For details about the different types of declarative business rules, see "Types of business rules" on page 189.

The Versata Logic Studio provides menu options that you can use to build and compile business rules into the Java files that are generated for each data object. You have the option of deploying each data object and query object to the middle tier as a Java class or as an Enterprise JavaBean (EJB). Every business object file includes marked sections where you can add custom code that is preserved when object files are rebuilt and recompiled.

Why use declarative business rules?

The Versata Logic Studio automates the procedural implementation of transaction logic that is defined declaratively in business rules. When you deploy your business rules to the Versata Logic Server, the system automatically generates and compiles Java component files that contain the required logic. Each rule fires automatically whenever an application action affects the data element (attribute, relationship, or data object) to which the rule is attached. The firing of one rule can affect related data elements and cause the firing of the rules attached to these other data elements.

Every transaction in a generated application automatically reuses all business rules applicable to the data elements affected by the transaction, regardless of the user action that initiates the transaction. Because the Versata Logic Studio automates all of the required processing, business rules can be shared among multiple business transactions, or functions, and even among multiple applications running against the same database. If the requirements for transaction logic change, declarative business rules can be altered and redeployed without consideration of the processing implications.

Business rules provide integrity enforced by the Versata Logic Server. The implementation of business rules as a middle tier also reduces network traffic to make performance scalable. Rules processing can be distributed across multiple Versata Logic Servers on different machines to improve performance further.

Defining declarative business rules generates a large amount of code that would otherwise need to be handwritten. This automatic generation of code shortens initial development, reduces debugging, and simplifies maintenance. You can easily extend declarative business rules by incorporating calls to your own methods within rule expressions, adding your own event-handling code, or subclassing system-supplied classes used to build rule component files. The Versata Logic Suite's J2EE-enabled architecture allows your custom additions to be preserved as you modify rules and regenerate rule component files.

Declarative business rules enable application development with a business orientation that focuses on the big picture. Developers and end users alike can devote more attention to the desired behaviors for an application, rather than on the details of how to implement the behaviors.

The modularity and reusability of business rules encourage a flexible approach to business requirements definition and implementation. One standard approach is to work with end users to develop an inventory of application business functions, analyze each function to determine transaction logic requirements, and define rules that enforce each requirement. You can integrate your existing requirements, definition process, and modeling tools into the Versata Logic Suite development process.

When you change a rule, you do not have to model all of the potential effects on all related requirements and functions. The Versata Logic Suite's automation of rules processing addresses the dependencies between business rules. As a result, modifying the transaction logic is a fairly short and simple procedure.

To sum up, declarative business rules offer the following advantages:

- You can concentrate on requirements rather than on implementation.
- Business rules are easy to communicate to management and users so that you can work more effectively with them.
- Testing and maintenance are simplified because procedural implementation is automated. You need to maintain only the statement of the rule rather than all the procedural code. Rules are order independent and automatically applied to all relevant transactions in the generated code.

Business rules functionality compared to spreadsheet functionality

One way to understand business rules is to compare their functionality to that of spreadsheets. The following table provides this analogy.

Spreadsheet functionality	Declarative business rules functionality
The formula to determine a spreadsheet cell value may refer to many other spreadsheet cell values, each determined by its own formula.	The rule to determine an attribute value may refer to many other attribute values, each determined by its own rule.

Spreadsheet functionality	Declarative business rules functionality
A change to the value or the formula for a spreadsheet cell, or insertion or deletion of spreadsheet rows, may cause automatic changes to many other cell values that refer to the changed cell value in their own formulas.	A change to the value or rule for an attribute, or insertion or deletion of records, may cause automatic changes to many other attribute values that refer to the changed attribute value in their own rules. Each declarative business rule is defined on a single data object. Because cascading rules automatically cause other rules to fire, business rules can be combined to implement update processing across multiple data objects.
A change to a single cell in a set of linked spreadsheets may cause changes to cells in many other spreadsheets.	A change to one business rule may affect many business processes, which are implemented through that rule.

Types of business rules

The Versata Logic Studio allows you to define the following types of business rules:

- Derivation rules
- Validation rules
- Presentation rules, including captions (available for applications designed in Versata Logic Studio only.
- Referential integrity rules
- Constraints
- Actions

The business rules that may be applied to an element in a data model depend on the element's object type, as shown in the following table.

Type of business rule	Applicable data element
Derivation, Validation, and Presentation	Attribute
Constraints, Actions, and Presentation	Data object
Referential Integrity and Presentation	Relationship

Derivation rules

Derivation rules define how an attribute's value is computed when a database update occurs. Derivations can be aggregations of child record values (sums or counts), replicates of parent record values, or formulas based on values of other attributes in the same record. You define derivation rules on the Attributes: Derivation tab of the Transaction Logic Designer.

You can create derived attributes that are used for calculation of other attributes' values but are not stored in the database. These attributes are called virtual attributes. A virtual attribute is calculated on the Versata Logic Server as needed rather than being physically stored in the data source. The Persistent check box on the Derivations tab indicates whether an attribute is virtual or stored. The decision of whether to make a derivation virtual or stored has significant implications for optimization of your applications. For information about defining and using virtual attributes, see "Virtual attributes" on page 104.

Whenever an attribute has a derivation rule, the Versata Logic Studio by default generates a validation rule to prevent user updates. This validation rule can be changed, if necessary.

Each derivation rule can cause other derivation rules to fire. This cascading of rules enables complex update processing across multiple data objects. For an example of this process, see "Multiple data object updates through cascading rules" on page 192.

The following are types of derivation rules.

Rule type	Explanation
Sum	A sum rule derives a parent attribute value by adding values of a specified attribute in a child data object. A sum rule optionally can include a qualification expression that restricts child records included in the sum. The generated component code for a sum rule: • Initializes the parent sum attribute value to be 0 on insert of parent record. • By default, raises an error when a user tries to insert or update the parent sum attribute directly. (This default can be overridden by changing the Prevent User Updates validation rule.) • Adjusts the sum attribute value by subtracting values of deleted child rows, adding values of inserted child rows, and subtracting or adding the changes to values of updated rows. (Note that most processing occurs in the child data object's component.) Defining sums as described here results in superior performance because it does not result in aggregate queries every time the summed value needs to be changed.
Count	A count rule derives a parent attribute value by counting the number of records in a specified child data object. A count rule, optionally, can include a qualification expression that restricts the child records included in the count. The generated component code for a count rule: • Initializes the parent count attribute value to 0 on insert. • By default, raises an error when a user tries to insert or update the parent count attribute directly. The default can be overridden by changing the Prevent User Updates validation rule. • Adjusts the count attribute value: decreases the value by one for each deleted child row and each updated child row that no longer meets the specified condition, and increases the value by one for each inserted child row and each updated child row that newly meets the specified condition. (Note that most processing occurs in the child data object's component.) Defining counts as described here results in superior performance because it does not result in aggregate queries every time the counted value needs to be changed.

Rule type	Explanation
Parent Replicate	A parent replicate rule derives a child attribute value by copying a value from an attribute in a related parent data object. A replicate occurs when a new child record is inserted or when a new parent is assigned to an existing child. Additionally, a Maintained option is available to specify whether updates to the replicated attribute in the parent should be cascaded to related children. By default, replicates are not maintained. Replicates are useful in reducing joins, and in making parent data values available for use in other business rules. The generated component code for a parent replicate rule: • Copies the parent attribute value to the child replicate attribute on insert. • Copies the new parent attribute value to the child replicate attribute if a foreign key is changed in the child. • Changes the child replicate attribute value to NULL if a foreign key is nullified. • Cascades updates to parent attribute to child replicate attribute, if replicate is maintained. (Note that most processing occurs in the parent data object's component.) • By default, raises an error when a user tries to insert or update the child replicate attribute directly. The default can be overridden by changing the Prevent User Updates validation rule.
Formula	A formula rule derives an attribute value by evaluating an expression on other attribute values from the same record. Formula rules can reference data modification operations (`Inserting, Updating, Deleting`), include system-supplied or developer-defined methods that return a value, and include `if-then-else` conditional structures. The generated code for a formula rule calculates attribute value on insert or update.
Default	A default rule specifies the value of an attribute when a user does not enter a value on insert. This specification can be a literal number value, a quoted string, or a method that returns a value. Subsequent user updates can change a default value. The generated code for a default rule: • On insert, checks if an attribute is NULL. • If an attribute is NULL, inserts the default attribute value.

Note: Derivation rules other than defaults are not supported on primary or foreign keys.

Multiple data object updates through cascading rules

A derivation rule fires when a user action changes the value of an attribute used in the rule. The firing of one rule often results in changes to other attributes, causing the firing of rules that use those attributes' values. Because of this cascading of rules, one user action can result in updates to multiple data objects, including changes that start in a parent and cascade to its children. The cascading can begin with a number of user actions, including a change in the value of a primary key, the deletion of a parent record, and the deletion of a child record.

Child to parent cascade

A change to the value of a child record that is used to provide a sum, used as a qualification condition in a validation rule, or used as a foreign key, can cause an adjustment to a parent record. The change in the parent record may trigger other rules in turn. For example, in the sample applications included with the Versata Logic Suite (with presentation design only), when a user changes the value of the QtyOrdered attribute for an ORDERITEM, many other attribute values change.

- PART.QtyUnshipped is adjusted, because its derivation rule uses ORDERITEM.QtyOrdered as a summed attribute.
- ORDERITEM.Amount is recalculated, because its derivation rule uses ORDERITEM.QtyOrdered.
- ORDERS.AmountItems is recalculated, because its derivation rule uses ORDERITEM.Amount as a summed field.
- ORDERS.OrderTotal is adjusted, because its derivation rule uses ORDERS.AmountItems in a formula.
- CUSTOMERS.ActBalance is recalculated, because its derivation rules uses ORDERS.OrderTotal as a summed field.
- A constraint on the CUSTOMERS data object is fired by this recalculation. The change to PART.QtyUnshipped also causes the firing of other rules for the PART data object.

Parent to child cascade

A change to a parent record that is used in a maintained replicate rule may cause an update to child records to receive the new value. For example, when a user changes the value of the ShippedFlag attribute for an ORDER, the value of the ShippedFlag changes for each ORDERITEM in the ORDER. The change to ShippedFlag values for ORDERITEM records adjusts the values of the QtyShipped and QtyUnshipped attributes for related PART records.

Parent-to-child cascades also may result from referential integrity updates.

Note: The sample database includes examples of how different types of derivation rules can cascade to implement updates across multiple data objects.

Attribute validation rules

Validation rules define limitations for attribute values. These limitations can be based on a developer-defined condition or on a coded values list. The Versata Logic Studio also allows you to restrict attributes' updatability and nullability through validation rules.

You define validation rules on the Attributes: Validation / Data Type tab of the Transaction Logic Designer. On this same tab, you can review and change attributes' data types. For details about working with attribute data types, see "Changing an attribute's data type" on page 103.

Note: Data object validation rules are defined as "constraints" in the Transaction Logic Designer. For information about these, see "Constraints" on page 198.

The following are types of validation rules.

Validation rule type	Explanation
Condition	A condition validation rule limits values in an attribute to those that meet a defined conditional expression. For each condition validation rule, the Transaction Logic Designer allows you to enter a custom error message that appears when a violation occurs. Condition validation rules can: • reference data modification operations (`Inserting, Updating, Deleting`); • use the `:Old` function to reference attribute values prior to update; • include system-supplied methods and developer-defined methods (that are registered and listed in the Enterprise Object Browser.) The generated code for a validation condition rule raises an error if the condition is not met.
Coded values list	A coded values list validation rule limits attribute values to the values in a coded values list. A coded values list consists of pairs of corresponding values. Each pair has a stored value and a display value. The stored values are stored on the database server. The display values are shown in a combo box to the user. The generated code for a coded values list validation rule validates attribute data against stored values. In addition, a coded values list rule on an attribute drives the client application to build either a combo box (by default) or option buttons for the attribute (with presentation design only). For more information about coded values lists, see "Working with coded values lists" on page 95.

Validation rule type	Explanation
Nullability	The Transaction Logic Designer provides a Value Required validation rule with a system-supplied error message. You can specify in this rule that an attribute value cannot be NULL. The generated code for a Value Required rule defines the attribute as not NULL, and nullability is checked in the client application during update processing.
Updatability	The Transaction Logic Designer provides a Prevent User Updates validation rule with a system-supplied error message. You can specify in this rule that an attribute value is not updatable by users. The Versata Logic Studio automatically defines a Prevent User Updates validation rule for an attribute in the Transaction Logic Designer if the attribute has a derivation rule. You can override this default, even for non-persistent, derived attributes. The generated code for a Prevent User Updates validation rule raises an error when a user tries to update the attribute directly. (A user can enter an attribute value for a newly inserted record. If the attribute is derived, the newly entered value is overridden by the derivation when the record is saved.) In Java applications, non-updatable fields are built as disabled on forms (with presentation design only). In HTML applications, non-updatable fields display as empty at design time, while at run time, text displays in the field but is not editable (with presentation design only).
Data type	The Transaction Logic Designer displays the data type of each attribute, and allows you to change data types for attributes displayed in scalar fields. For Text, Number, and DateTime types, you also indicate a sub type. In the case of Text attributes, you also can change the size. The Transaction Logic Designer checks data type changes, prohibiting changes between mismatched types, changes to indexed attributes and key attributes, and changes to AutoNumber when data already exists in an attribute. Data type checking is performed on both the client and the server.

Presentation rules

Presentation rules define certain aspects of Versata Logic Studio-generated application user interfaces. Archetypes and application diagrams define other aspects. Presentation rules are not implemented through business object code and do not affect server processing.

The following table lists the types of presentation rules.

Rule type	Explanation
Attribute presentation	Define attribute presentation properties on the Attributes: Presentation tab of Transaction Logic Designer. These properties include: • A caption that appears as a label for the field on generated forms or pages. • A format for text in the field. • A status bar message that appears in the window status bar when the attribute receives focus (for Java applications only). • An archetype to be associated with the attribute, that determines the control or element to be used for the field on generated forms or pages. Definition of these presentation properties is optional. The Versata Logic Studio provides a default caption and archetype for each attribute.
Data object presentation	Define data object presentation properties on the Properties: Presentation tab of Transaction Logic Designer. The Versata Logic Studio provides default presentation properties for data objects. You may change the defaults. These properties include: • Singular and plural captions that appear on generated forms or pages where the data object is a root RecordSource. In Java applications, these captions are used for controls and appear as references in the status bar. In HTML applications, these captions are used for page elements. • An image to be added to the data object. The image appears on toolbar buttons in Java applications.
Relationship presentation	Define relationship presentation properties on the Relationships: Presentation tab of Transaction Logic Designer. The Versata Logic Studio provides default presentation properties for relationships. You may change the defaults. These properties include: • A caption that appears as a label for data in transitions from parent to child forms or pages. • A caption that appears as a label for data in transitions from child to parent forms or pages. • A child role name used in APIs that retrieve child data for the parent data object in the relationship. • A parent role used in APIs that retrieves parent data for the child data object in the relationship.

Captions

The Versata Logic Studio uses captions to generate RecordSource labels and command button labels on the applications it builds. Generated captions are based on defaults for data objects and relationships. You can override most of the defaults in either the Transaction Logic Designer and/or the forms or pages.

There are three kinds of captions:

- Attribute captions
- Data object captions
- Relationship captions

Note: If you change captions in the Transaction Logic Designer after you have generated an application, modified captions do not appear on forms or pages until you rebuild the form or page layout.

Attribute captions

Define attribute captions on the Attributes:Presentation tab. The attribute names are the default.

In forms and pages, attribute captions are the default label captions for fields and attributes.

Data object captions

Define singular and plural data object captions on the Properties:Presentation tab. The singular caption defaults to the data object name and the plural caption defaults to the data object name with an appended “(s)”. For query objects that are used as RecordSources, the childmost data object in the query is used to derive the default caption.

The singular data object caption is the default label for display forms or pages where the data object is a root RecordSource. The plural data object caption is the default label for grid forms or grids on pages where the data object is a root RecordSource. Plural data object captions also are used as RecordSource references in the status bar.

Relationship captions

Define relationship captions on the Relationships:Presentation tab. Target data object captions are used by default. The parent-to-child relationship caption defaults to the child’s plural data object caption. The child-to-parent relationship caption defaults to the parent’s singular data object caption.

On forms in Java applications, relationship captions are the default captions for command buttons that execute form transitions.

Referential integrity rules

Referential integrity rules preserve relationships between data objects when data manipulation language (DML) updates occur. You define referential integrity rules on the Relationships tab of the Transaction Logic Designer. For each relationship in your data model, you can indicate whether you want to enforce referential integrity; however, you must enforce it to use the relationship in rules.

You can define separate rules for parent updates, parent deletes, and child inserts/updates. The Versata Logic Suite supports standard referential integrity rules with additional provisions for Cascade Update, Cascade Delete, and Cascade Nullify.

The generated code for a default referential integrity rule rolls back the entire user update request if a referential integrity violation occurs. The rollback reverses all changes to data objects that were caused by the update request. By default, the Versata Logic Server enforces referential integrity. You have the option of enforcing referential integrity in the database server as well; you select this option when you deploy the data model to the database server.

The following table lists the types of referential integrity rules available in the Versata Logic Studio.

Rule type	Explanation
Restrict	There are three Restrict rules: 1) Prevent parent update if children are present. 2) Prevent parent delete if children are present. 3) Prevent child insert/update if parent is not present. Each of these rules can be defined separately. You may edit the rules and the system-supplied error messages.
Cascade Update	A cascade update rule, or Update Children on Parent Update, indicates that foreign key values for child records should be updated to match an updated parent key value.
Cascade Delete	A cascade delete rule, or Delete Children on Parent Delete, indicates that related child records should be deleted when a parent is deleted.
Cascade Nullify	A cascade nullify rule, or Null Children Foreign Key on Parent Delete, indicates that foreign key fields for related child records should be nullified when a parent key is deleted.
Cascade Insert	A cascade insert rule, or Insert Parent If None on Child Insert/Update, indicates that if an inserted or updated child record has no parent record, a record should be inserted into the parent data object with a primary key that matches the child foreign key. Note that this option is available only if the parent object does not have required fields other than the primary key fields.

Constraints

Constraints are object-level rules you can use to enforce value changes to attributes. Constraints consist of a condition (such as whether the Account Balance exceeds the Credit Limit), an action (either to accept or reject the pending change), and an error message. When the condition evaluates to true, then the action is taken, and a specified error mesage is returned, along with a specified field to be highlighted. The condition is evaluated whenever any attribute specified in the condition is going to change.

While this is similar to a validation, an attribute can have multiple constraints but only a single validation.

Constraints can:

- Reference data modification operations (`Inserting, Updating, Deleting`).
- Use the `:Old` function to reference attribute values prior to update.
- Include system-supplied and developer-defined methods. (These methods must be registered and listed in the Enterprise Object Browser.)
- Govern derived attributes.

All attributes, derived or otherwise, in the conditional expression of a constraint must be located in the same data object.

You define constraints on the Constraints tab of the Transaction Logic Designer. You may define multiple constraints on a data object. All constraints are evaluated when an update to a data object occurs. The generated code for a constraint rolls back the entire user update request if a constraint violation occurs during the update. The rollback reverses all changes to data objects that were caused by the update request. A transaction is rolled back when the conditional expression for a Reject When constraint evaluates to true or when the conditional expression for an Accept When constraint evaluates to false.

The Versata Logic Studio provides a default error message, but you may enter a customized message to display when the constraint prevents a user transaction from committing. You also may define the attribute where the cursor is placed after the error is raised. The error attribute is client-side information only.

Note: Constraints are inherently unordered. We do not recommend that you attempt to reorder them.

Business rule actions

Business rule actions extend the transaction logic processing capability of the Versata Logic Suite by incorporating your procedural code into the declarative model. Actions enable you to customize the Versata Logic Studio-generated business objects through calls to developer-defined methods. Action customizations are preserved automatically when business rule components are regenerated.

An action rule executes a call to an external method when certain conditions are met. You can pass parameters from the attribute values of the current row.

You must register an object for it to be available to a method call from an action rule. To register an object, click Add in the Enterprise Object Browser, or choose Tools → Add Object to Registry in the main menu.

Transaction logic processing

After you declare business rules for each data object, you can build and compile the rules logic into a Java business object for each data object, choosing implementation as either a Java class or an EJB for each one. In Versata Logic Studio you define derivations and constraints on data, not transactions, because the system automatically generates transaction-specific transaction logic when you build and compile business rules into the business objects. Code generated to implement business rules is added into the business objects that execute on the Versata Logic Server and IBM WebSphere Application Server. The Versata Logic Server provides logic execution services and acts as an EJB container for any business objects implemented as EJBs, while IBM WebSphere Application Server provides application execution services. The execution of transaction logic code on the middle-tier Versata Logic Server guarantees data integrity and reduced network traffic, enhancing performance.

Also, because the Versata Logic Suite centralizes the transaction logic execution on the Versata Logic Server, each application automatically inherits transaction logic. There is no need to recode or even recompile business rules for each application, whether for rapid iterative development, or simple maintenance. When you redeploy business rules, the Versata Logic Studio automatically analyzes all data dependencies, rebuilding to achieve a correct and optimal processing order. This rebuilding protects against a gradual loss of coding efficiency due to multiple patches.

Most of the processing logic for rules resides in the business object files, in pure Java code. The object files have pluggable data access modules, and separate files called Versata Connectors execute data access. The Versata Logic Suite includes default Connectors for SQL-based data sources, like RDBMSs. You must obtain specialized Connectors separately or write your own Connectors to provide data access to non-SQL data sources.

The four basic activities of rules processing on the Versata Logic Server are:

- **Analyze the update**. The first step is to determine the values that the user has changed.
- **Adjust dependent data**. The dependencies among data are computed when business rules are compiled, so the generated components automatically adjust the correct data when users change values. Dependencies are recomputed each time you alter the rules, so they are always correct, complete, and consistent. Queries and `Where` clauses within rule expressions are implemented in SQL and passed to the data source.
- **Check constraints and data restrictions**. These include referential integrity, coded value lists, attribute constraints, and nullability. Basic referential integrity enforcement for RDBMS objects is supported in the database server in the form of DDL constraints, if this option was selected during data model deployment.
- **Invoke events.** Each data object's Java implementation file exposes server events. You can add code to these events to modify server processing. Your server event code may extend the logic for the client application and/or the Versata Logic Server. The system calls any event handlers you have defined to extend declarative rules processing with your own procedural code.

Order of rule processing operations

To use rules effectively, you should understand the order in which they are executed. As in a spreadsheet, the processing order is implicit, and cannot be modified directly by you, although you can nest rules to model complex behavior among multiple objects. For some information about nesting, see “Nest levels” on page 204.

The Versata Logic Server’s Transaction Logic Engine processes business rules in a defined order. This ordered dependency enables rules to be captured declaratively and implemented procedurally. To achieve this, business objects interact with each other to enforce rules across objects at a predefined stage of the dependency graph.

When a modification (insert, update, or delete) is made to a business object and then saved, a set of operations are carried out in the business object. The following table summarizes these operations in the order in which they occur. Specific information about each operation follows the table.

Note: You can change the sequence that the rules fire in the `javaComponent.tpl` file, which is the template used during building and compiling.

Operation	Scope	Insert	Update	Delete
Before Insert/Update/Delete Event	local	x	x	x
Set Defaults	local	x		
Attribute Alterability Check	local		x	
Parent Check / Fetch Parent Replicate	local	x	x	
Evaluate Formula	local	x	x	
Coded Value Constraint Check	local	x	x	
Attribute Validation Check	local	x	x	x
Business Object Constraint Check	local	x	x	x
Nullability Check	local	x	x	
Conditional Action	local	x	x	x
Child Cascades	child	x	x	x
Parent Adjustment	parent	x	x	x
After Insert/Update/Delete Event	local	x	x	x

Each type of modification (insert, update, or delete) enforces the set of rules indicated by an 'x'. For each modification, all local calculations and validations are done first. These are followed by modifications to related objects in the form of cascades to child objects and adjustments to parent objects. Each of these related modifications then implements the dependency graph in its entirety as well.

Rules processing also includes defined nest levels and modification state flags.

Before insert/update/delete event

Before any generated transaction logic is executed, developer-supplied event code is executed. This is recommended for setting up conditions or capturing data to be used later.

Set defaults

For inserts, default values are inserted into attributes.

Attribute alterability check

For updates, a check is made to verify that any attributes that are modified by the user are modifiable.

Parent check/fetch parent replicate

For inserts and updates, a referential integrity check is performed to verify that child rows have related parent rows. If the "Insert Parent if None" rule is declared, a new parent row is created and inserted using the child foreign key as the parent primary key. Other attributes must have defaults or formulas associated with them, or they must be nullable.

Also, if there are any parent replicates in the child, the value is fetched from the parent.

Evaluate formula

For inserts and updates, all formula values are calculated.

Coded value constraint check

For inserts and updates, if any attributes have Coded Values Lists validations, it is verified that the attribute value is in the Coded Values List.

Attribute validation check

For inserts, updates, and deletes, it is verified that all attribute validation conditions are met.

Business object constraint check

For inserts, updates, and deletes, it is verified that all constraints within the business object are satisfied.

Nullability check

For inserts and updates, it is verified that any attributes with null values are not required.

Conditional action

After all local modifications have been made in the business object, a developer-supplied condition is tested. If the condition is true then a developer-supplied method call is made. This is recommended when there are other declarative rules in the business object or related business objects.

Child cascades

After local calculations and validation checks are performed, child cascades are done. This applies to insert, update, and delete. For each related child business object, one or more of the following may occur:

- **Cascade update foreign key**. If the parent primary key has changed, the change is propagated to each related child.
- **Nullify foreign key**. If this rule is declared and a parent is deleted, the foreign key is set to null in all related children.
- **Cascade delete**. If this rule is declared and a parent is deleted, all related children also are deleted.
- **Cascade update parent replicates (Maintained)**. If the value of a replicate is modified in the parent, the new value is cascaded to the children. Note that the parent business object performs the change on the child and then saves the child. This causes the scope of logic execution to be nested in the child.

Parent adjustments

Parent adjustments are done if a child row participates in an aggregate calculation in one or more related parent objects. When an attribute in the child affects a sum or a count in a parent, the change is passed to the parent and the parent is adjusted by the amount of the change. The following conditions may cause a parent adjustment:

- A changed attribute in the child may be a value that is summed in the parent. In this case, the net value of the change is transmitted to the parent, which adds that to the existing aggregate value.
- A changed attribute in the child may participate in a qualifying `Where` clause for a sum or a count. In this case, the row with the changed attribute may no longer qualify for the aggregate where once it did, or it may now qualify where it did not previously. In either case, the aggregate must be adjusted accordingly.
- A changed attribute in the child may be part of a foreign key that has caused the child to be 're-parented'. If a child is re-parented, both the old and new parents must be adjusted to reflect the change.

Note that the child business object transmits the change to the parent and then saves the parent. This causes the scope of logic execution to be nested in the parent. Each of the parent adjustments is done one at a time from the child. If the resulting change in a parent results in a new parent replicate value for the child, the value will be cascaded (and therefore the child business object code will be re-entered) from within the scope of the parent update.

After insert/update/delete event

After all generated transaction logic is executed, developer-supplied event code is executed. It is recommended that you add event code here for cleanup of other conditions or for capturing audit data.

Nest levels

When changes are propagated across multiple business objects, the scope of execution is nested. Within each nest level, rules are reexecuted in the same order of operations. Business rules can be reentered multiple times across nesting levels. The same object (row) instance is used to prevent lost updates in complex recursive cases. After all of the nest levels are completed, the transaction logic returns to its starting point at the first nest level. The examples used to illustrate nested rules processing are based on the sample repository:

For example, if a user updates an ORDERITEM form or page, local modifications to the ORDERITEM business object take place at nest level 1. This modification causes a parent adjustment to the ORDERS business object (for example, update the ORDERS Total Amount). In this case, the ORDERS transaction logic is processed at nest level 2. This modification, in turn, causes another parent adjustment to the CUSTOMERS business object (for example, update the CUSTOMERS Account Balance). The CUSTOMERS transaction logic is processed at nest level 3.

It is possible for transaction logic to be re-entered. In the example above, suppose that there are volume discounts that apply a 10% discount to all ORDERITEMS under $50 in any ORDER that has in excess of 100 ORDERITEMS. When an ORDERITEM business object (nest level 1) adjusts an ORDER so that the 100 ORDERITEM condition is met, the discount is replicated from the ORDERS object (at nest level 2) back down to the ORDERITEM object (which is re-entered at nest level 3).

After necessary calculations are made to each ORDERITEM under $50 (using a conditional formula), the ORDERS Total Amount would need to be recalculated with another parent adjustment (nest level 4). Finally, the CUSTOMERS Account Balance would be adjusted again (nest level 5). After each nest level is completed, the transaction logic 'unwinds' like function calls being popped off of a stack, until the transaction is completed back at nest level 1.

Modification state flags

Logic code in a business object can determine whether an insert, update, or delete is taking place on it by using the following methods: `isInserted()`, `isUpdated()`, and `isDeleted()`. It is possible for `isInserted()` and `isUpdated()` to return different Boolean (Yes/No) values depending upon the current nest level. The operation on the component differs depending upon the context, as shown in the next example.

For example, if a row is being inserted, and transaction logic execution has entered a related object (thus incrementing the nest level), and then the logic re-enters the original object (incrementing the nest level again), `isInserted()` would return True in the first nest level and `isUpdated()` would return False. However, upon re-entry in the third nest level, `isInserted()` would return False and `isUpdated()` would return True.

In general, `isInserted()` returns True only if the current nest level is equal to the nest level in which the insert actually was initiated. This nest level is not necessarily nest level 1, because an 'Insert Parent if None' rule or custom code could be doing an insert from within some other nest level.

There is a method available called `isChanged()`to determine whether or not an attribute has changed. This method returns True if this attribute has changed, or if other attributes that it depends upon have changed, as in the case of a formula. For example, if a formula says `a + b = c`, and `a` changes then `isChanged("c")` returns True, even if the formula has not yet been reevaluated.

Analyzing business requirements

Before you can begin defining business rules in the Versata Logic Studio, you need to spend some time defining the business functions and requirements related to your applications.

This analysis can occur outside of the Versata Logic Studio development environment. You can use whatever methods and tools you would like for requirements analysis and definition. What is different with the Versata Logic Suite is that after you have analyzed and defined business requirements, you need to translate requirements to declarative business rules. This process does not have an explicit road map, because you can complete it in a variety of ways. This section provides some hints and guidelines for this process. For more information about the Versata Logic Suite development process, see the *Architecture and Project Guide*.

Business function definition

A business function is a business operation to be supported by one or more applications. It corresponds to or includes one or more database transactions. Entering a new order and deleting a customer are examples of business functions.

You may need to break down business functions by business areas. For example, you could define business functions for the area of order processing that would include entering an order, adding an order item, and so on.

Generally, you should express business functions with verbs to describe the operations or actions, the tasks someone would need to perform using the application(s).

Business requirements definition

A business requirement is a condition or statement to be satisfied or enforced in the application. This requirement can be high-level or low-level.

Generally, each requirement indicates what must be done or satisfied in carrying out one or more business functions. When defining a requirement, you should state what your customers or users want in terms that you and they can understand. To continue with the example begun with business functions, you cannot add an order item unless the part on the item is recognized by the application.

Declarative business rules can serve as the specifications that support your business requirements. Some requirements map to a single rule each, while others require multiple rules.

Mapping requirements to rules

This list of tasks provides general suggestions for steps to follow in mapping requirements to rules:

1. List business functions to be addressed by applications.
2. List business requirements for each function. (Some requirements may apply to more than one function.)
3. Break down requirements to their simplest level.
4. Map lowest level requirements to the data model objects and attributes to which they apply. (If necessary, add objects and/or attributes to the data model.)
5. Declare one or more business rules for the applicable objects and attributes.
6. Identify key transactions of the business function for testing and performance analysis purposes.

Top-down approach

The task list above uses a top-down approach to mapping requirements to rules. This approach starts with the big picture and breaks down high-level processes into tasks, defining what must be checked or calculated for each task. An alternative approach is to start from the bottom with individual actions and move upwards to more complexity.

With the top-down approach, you break down requirements into their simplest level. The following are examples of simple requirements that can be translated into rules:

- Customer account balance cannot exceed the customer credit limit.
- Customer account balance is the sum of unpaid order totals.
- Order total is the amount of order items plus freight.
- Amount of order items is the sum of order item amounts.
- Order item amount is price times quantity ordered.

The following are examples of larger requirements that have been broken down to a simpler level:

- Check credit limit for each customer upon order entry.
  - Customer account balance is the sum of unpaid order totals.
  - Customer account balance cannot exceed customer credit limit.
- Compute order total for each order entry.
  - Order total is equal to freight plus tax plus the total amount for order items.
  - Total amount for order items is the sum of the amount for each order item.
  - Each order item amount is equal to the quantity ordered times item price.
  - Order item price is equal to the part price at the time of order.

Selecting rules

After you have stated requirements at their simplest level, you need to select a rule or rules to implement each requirement. When you have decided the type of rule(s) to use to implement a requirement, you are ready to define it in the Transaction Logic Designer. For instructions on defining rules, see "Procedures for defining business rules" on page 232.

You can analyze requirements and use categories to limit the possible choices of rules. The language of the requirement can help you to choose the type of rule. The words "have" or "is in" usually point to a relationship. Definitions often are derivations, sometimes constraints. If-then-else wording is available in formula expressions. Note that parent replicates are not always explicitly stated, but are often implied by another requirement that needs parent data to be available in child records.

Your answers to the following questions can help you to select the rule or rules for implementing a business requirement:

- Is the requirement covered already by the data model and referential integrity?
- Does the requirement have to do with the data itself (uniqueness, nullability, updatability)?
- What data element(s) are needed to satisfy the requirement?
- Does the requirement involve a single object or multiple objects?
  - If the requirement involves a single object, does it involve a single attribute or multiple attributes?
  - If the requirement involves multiple objects, in what direction is the calculation: up from children to parents (sum, count), or down from parent to children (replicate)?

The following table provides some further guidelines about translating requirements into rules:

Object/Attribute Involvement	Type of Rule
Single Object, Single Attribute	Default Validation Condition Coded Values List
Single Object, Multiple Attributes	Formula Constraint
Multiple Objects	Sum (Calculate up from children to parent) Count (Calculate up from children to parent) Parent Replicate (Bring down from parent to child)

Mapping requirements to the data model

As stated at the beginning of this chapter, the biggest issues to solve when you are defining a rule are:

- Identifying which data should own the rule.
- Gathering the data into the right object.
- Figuring out how to change data to fire the rule.

To solve these issues, map requirements to the data model as part of mapping requirements to rules. You should be able to map each term in a requirement to an object or attribute in the logical data model. (This data model is considered logical, because some attributes in it may be non-persistent, meaning they are virtual rather than physically stored.) If the requirement involves more than one object, look for relationships between objects.

You may need to modify the data model if you are unable to map some requirement terms. You can add data objects, attributes, and/or relationships. If you need to define a rule that uses a relationship between two objects, such as a sum, count, or replicate, referential integrity must be enforced for the relationship. For example, if you derive the customer account balance as the sum of order totals, the relationship between customers and orders must be enforced. For each derived attribute you add, you need to decide whether it is stored or virtual. This decision has implications for performance.

- For information about adding data objects, see “Adding data objects” on page 84.
- For information about adding attributes, see “Adding attributes to data objects” on page 102.
- For information about adding relationships, see “Adding relationships” on page 113.
- For information about virtual attributes, see “Virtual attributes” on page 104.

Rules design patterns

In many cases, you may need to combine rules to enforce a business requirement. The following combinations are common design patterns:

- Constraining derived attributes.

  Start with the desired constraint, then define derivations to gather data required for constraint comparison into attributes that can be compared.

- Gathering required data from other attributes in the same data object or from related data objects.
  - **Formulate.** Value for attribute calculated from other attributes in same data object.
  - **Replicate.** Value moves downward, copied from attribute in a parent data object into an attribute on the child data object.
  - **Aggregate.** Value moves upward into parent data object, summed or counted from child data object attribute values.Using count rules as existence checks (determining if count greater than zero).
- Using count rules as existence checks (determining if count greater than zero).
- Comparing old and new values to determine if state transition occurred.

The sample repository included with the Versata Logic Suite includes many examples of these common design patterns as well as others. In addition, this Guide provides examples; see “Transaction Logic Examples” on page 397.

Recognizing non-declarative patterns

As you map requirements to rules, it is important to recognize requirements that cannot be translated to declarative business rules.These types of requirements, called non-declarative patterns, include the following:

- More complex relationships than parent-child such as siblings, cousins.
- Quantity-based discount schedules.
- Batch driver loops.
- Workflow, including time-based and calendar-driven rules enforcement.
- Data retrieval with a user-defined business function.

The Versata Logic Suite provides a variety of ways for you to extend and customize rules in order to meet these non-declarative requirements, including extending rules with method calls, adding event-handling code, adding custom Java methods, and subclassing the Versata Logic Server Classes included with the product. For information about these techniques, see “Extending Business Object Code” on page 321. For examples of these techniques, see the sample repository and “Transaction Logic Examples” on page 397.

CHAPTER 7

Defining Business Rules

Chapter overview

This chapter discusses the process for defining declarative business rules to implement transaction logic. After you read this chapter, you should have a basic understanding of how to use the Transaction Logic Designer to define rules. This chapter includes the following:

- "Overview of business rules definition" on page 213, provides background information about business rules definition tasks, including the following:
  - "Business rules design issues" on page 213, describes issues you need to consider before you define rules.
  - "General process for defining business rules" on page 216, outlines the steps for defining all of your business rules as part of an iterative process.
- "Understanding the Transaction Logic Designer" on page 220, describes the user interface available for rules definition.
- "Procedures for defining business rules" on page 232, provides specific procedures for defining different types of rules.
- "Business rule syntax" on page 244, describes the syntax supported for rules expressions.

Overview of business rules definition

After you have analyzed business requirements for your system and built a data model in a Versata repository, you are ready to begin the business rules definition process. You need to break down requirements into rules, recognizing design patterns for rules combinations. Also you need to recognize non-declarative patterns in your requirements. To satisfy non-declarative requirements, you can extend business rules, modifying generated code for transaction logic. But before you begin customizing with your own code, you should fully define declarative rules in the Versata Logic Studio.

Defining declarative business rules in the Versata Logic Studio is an iterative rather than a rigidly sequential process. You have a lot of flexibility in determining the order in which you complete tasks. You most likely will define rules in stages. As users review the application prototype and you refine and add to requirements, you will need to modify and add to rules definitions. In many cases, rules definition and data modeling tasks will overlap.

In order to review rules as you define them, you need to build and compile the business object files that include code for rules' logic execution, then deploy these to the Versata Logic Server and the IBM WebSphere Application Server. For information about these tasks, see "Building and Deploying Business Objects" on page 255.

After you have iteratively refined declarative rules, you can begin extending generated transaction logic code as necessary to fulfill your requirements. For information about generated code, see "Understanding Business Object Files" on page 285. For information about how to extend this code, see "Extending Business Object Code" on page 321.

Business rules design issues

Consider the following general issues when you are defining business rules for your Versata repository:

- You define business rules on data objects and their attributes, not on query objects. The system enforces rules for both query objects and data objects during run-time execution. Query objects inherit rules from underlying data objects and projected attributes. During logic execution, query objects are decomposed to data objects so rules can be enforced.
- Data model definition and rule definition often overlap. You may discover as you attempt to define rules that you need to refine the data model. You can use the Transaction Logic Designer to add, delete, and rename data objects; add, delete, and modify relationships; add, delete, and modify indexes; and add, delete, rename, and change data types for attributes. For information, see "Developing a Data Model" on page 31.

- After you have deployed the business objects that include rules execution code, rules are executing against the data source(s) so that any data values entered subsequently conform to rules or cannot be saved. However, preexisting data values may not conform to rules. To address this issue, the system provides a `recomputeDerivations()` function that you can execute to modify preexisting data so that it does not violate rules. You can create an administrative application that incorporates this API in its client event coding. For information about recomputing, see "Recomputing derivations" on page 354.
- A key decision as you define derivation rules is whether to make derived attributes persistent (stored) or non-persistent (virtual). For information about virtual attributes, see "Virtual attributes" on page 104. Consider the following guidelines as you define derivation rules:
  - For attributes with sum and count derivation rules, it is usually wise to keep these attributes persistent, since the storage overhead is minimal. The reason is that recreating the value requires reading all child records. There is little harm in making sum and count attributes non-persistent if these attributes are not included in displayed RecordSources and if they tend to have a small number of children. If the attribute is displayed in a grid, or is part of a query used to display a grid, or if its base data object is displayed in a grid, the attribute should definitely remain persistent. It should also remain persistent if the record is updated often, and its value is compared against an attribute that often changes in a constraint.

    If an attribute is non-persistent, you cannot examine its old value. If you have constraints or other expressions anywhere in your business rule repository that need to access the old value of this attribute, for instance, to check if it has just been updated, then do not make the attribute non-persistent.

Be aware that if you are redesigning a production system that a change from persistent storage to non-persistent storage is a change to the data model. Any existing tables will have to be recreated. If production data is stored in those tables, it will have to be converted using a SQL database tool, or by writing conversion programs.

- For attributes with parent replicate rules, there is no easy, general rule to follow. Instead, the decision to store a parent replicate is the same as the decision to denormalize a physical database model, which is based upon performance trade-offs. The first thing to look for, as with sums and counts, is whether the attribute is displayed or retrieved as part of a displayed RecordSource. If it is displayed, it might be best to keep the attributes persistent. Next, you might want to look at where the values originate. If they are stored in the parent, a single read is all that is required to retrieve them, but if they are actually stored in grandparents, or great-grandparents, there must be a read at each level of the hierarchy.
- Attributes with formula rules are usually the best candidates for non-persistent storage, as long as all the inputs into the formula are persistent. Then no additional database reads are required to reconstruct the value, so even displayed attributes can be recalculated quickly. Of course, you may find exceptions to this general principle if your formulas reference methods which are computationally complex. If the source attributes are not persistent, these must be calculated before the formula is calculated, and that can be costly.

  If you need to display formula values but not the derived attributes used as inputs, an interesting design strategy is to make the formula a persistent attribute, but not the inputs. Then no recalculation will be necessary because displayed values are stored.

- Attribute references in rule expressions must be local to the data object on which the rule is being defined. For sum and count qualification expressions, attributes must be local to the child data object that the rule references, rather than the parent data object on which the rule is defined. You can use derivations to reference attributes from related data objects. For more information about syntax for rule expressions, see “Business rule syntax” on page 244.

General process for defining business rules

The following steps provide an overview of the tasks you perform to define declarative business rules in the Versata Logic Studio. Review these steps to get a sense of the order in which you should perform tasks. For more detailed instructions for specific tasks, see "Procedures for defining business rules" on page 232.

Completing the prerequisites for business rule definition

1. Define the business processes to be automated and the business requirements to be enforced in your applications. For information about getting started with business requirements definition, see the *Architecture and Project Guide*.
2. Make sure you understand the different types of business rules available and how they can be used to implement business requirements. Then break down requirements into business rules that can enforce them. For overview information about business rules, see "Understanding Transaction Logic" on page 183.
3. Produce a data model in a Versata repository. You can import a data model from an RDBMS database or create the data model in the Versata Logic Studio. For information, see "Developing a Data Model" on page 31.

Defining basic declarative business rules

1. Double-click a data object to open it in the Transaction Logic Designer.
2. Indicate which data objects are to be used as coded values lists by opening each data object in the Transaction Logic Designer, selecting the Properties:Coded Values Lists tab, enabling the option, and completing dialogs as prompted.
3. Determine how attribute values will be calculated by entering attribute derivation rules for each data object. You can use derivations to reference attributes from related data objects. As you build rule expressions to define derivations, keep track of any new attributes you may need to add in order to automate calculations and other data model changes that may be necessary. Determine which derived attributes you want to physically store and which you want to define as virtual. Also note where you need to reference methods within rules expressions.
4. Determine restrictions for single attribute values by entering attribute validation rules for attributes in each data object. You can build a conditional expression to limit values, or select a coded values list to provide valid values. Define which attributes can be null and which can be updated by users.
5. Define constraints that validate data against conditions involving multiple attributes in a data object. You build a rule expression to define the condition and have the option of defining a custom error message.

6. On the Relationships tab of the Transaction Logic Designer, indicate whether referential integrity will be enforced for each data object's relationships and how it will be enforced for each type of data change.
7. In many cases, rules are automatically saved when you move the cursor within the Transaction Logic Designer or when you close it. To explicitly save rules entries, choose File → Save Transaction Logic.

Defining presentation rules

Presentation rules determine characteristics of the user interface for applications designed in the Versata Logic Studio. The Versata Logic Studio provides default rules, but it is a good idea to define your own rules early in the development process in order to obtain user feedback. Define attribute-level rules on the Attributes:Presentation tab. Define data object-level rules on the Properties:Presentation tab. Define relationship-level presentation rules in the Presentation frame of the Relationships tab.

Testing business rules and obtaining user feedback

1. To review the rules that you have defined, you can generate a business rules report to analyze your rules input at any time. This function is available from an option in the File menu. For information about business rules reports, see "Generating business rules reports" on page 239.
2. Deploy the data model and business object files so rules can execute against data and you can review the results. For instructions, see "Deploying Data Models" on page 121 and "Building and Deploying Business Objects" on page 255.
3. Build a test application that you can run to review business rules execution. This test application should include operations required for the production application to ensure that rules are firing as expected for these operations. For information about defining applications in the Versata Logic Studio, see the sections on designing Java and HTML applications in the *Application Developer Guide*.
4. Run the application to test the rules and illustrate their execution to users.

The Versata Logic Studio provides several ways for you to obtain additional information about rules.

- Once you have built and deployed business object Java files that include rules execution code, you can view and print the component file for a data object by choosing options in the File menu.
- You can fire rules for run-time applications without immediately saving the resultant data changes, by adding a rules test button to your applications. You can run the applications, make changes, and click the button to see which rules fire for which data changes and the results of firing. In this manner, you can ensure rules are firing when expected with the expected results. For more information, see "Computing results without saving" on page 355.
- The Versata Logic Server includes a rule tracing utility that you can use to debug rule errors that may not be apparent in run time. For information, see the *Administrator Guide*.
- You can use a third party debugger to step through business objects' rules code. For information about supported debuggers, see "Debugging business object code" on page 279.

Redefining the data model and rules

User feedback and test results may cause you to change the data model, define additional business rules, or redefine rules. Versata Logic Studio allows you to iterate through the rules definition process. You can add, delete, and rename attributes, change attributes' data types, add, delete, and modify relationships, and add, delete, and modify keys and indexes. You can add data objects, including those that are abstracted from standard relational tables and those that have other types of data sources. You can define additional rules and refine previously defined rules.

After you make changes to the data model and rules, redeploy the data model to the database server, then redeploy transaction logic to the Versata Logic Server. Then you can retest your rules and refine them until they meet your needs.

Defining extensions and customizations for rules

The following steps list tasks you can perform to extend and customize declarative business rules. For more detailed information about these tasks, see “Extending Business Object Code” on page 321.

1. Identify the methods to be referenced or called in rule expressions. These may include those supplied by the Versata Logic Suite and those you write yourself. You can view system-supplied methods in the Enterprise Object Browser. The ones you will use most frequently are those in the `versata.vls.DataObject` class.
2. Write your own methods. You can add them to an existing class or create a new class file, for example, a subclass of `versata.vls.DataObject`. If you create a new class file, you can add the class file to your repository, so its methods are available to be referenced in that repository’s rules. Or you can add the file to the registry, so its methods are available to all repositories system-wide.
3. Define action rules that call methods and build other rule expressions that reference methods. If the method does not exist in the current data object, you must specify its class name. By default, `versata.vls.DataObject` is the superclass for all data objects, so they inherit all its members. If methods are members of a `DataObject` subclass that you created, you can define this subclass as the superclass for a data object on the Properties:Data Access tab of the Transaction Logic Designer. Then the subclass’s methods exist in the data object.
4. Add event-handling code to the server events exposed by the Versata Logic Studio. To view a data object’s events, click the Files tab in the Versata Logic Studio Explorer, double-click an implementation file to open the Code Editor, click the right button to display events, and select an event from the drop-down list. You can review or add to the code in the designated section.
5. Set up data objects so you can define rules on data from sources other than relational tables. After you have added data objects to the data model you can define specialized data access by choosing the Custom option on the Properties:Data Access tab. You then must write custom Versata Connectors, add them to the repository, and provide their name on this tab.

Understanding the Transaction Logic Designer

The Transaction Logic Designer provides a graphical user interface to define transaction logic in the form of declarative business rules and to modify your data model from within the Versata Logic Studio. Data modeling tasks you can complete in the Transaction Logic Designer include adding, deleting, renaming, and changing data types of attributes; adding, deleting, and modifying relationships; and adding, deleting, and modifying indexes and keys. For information about these tasks, see "Developing a Data Model" on page 31.

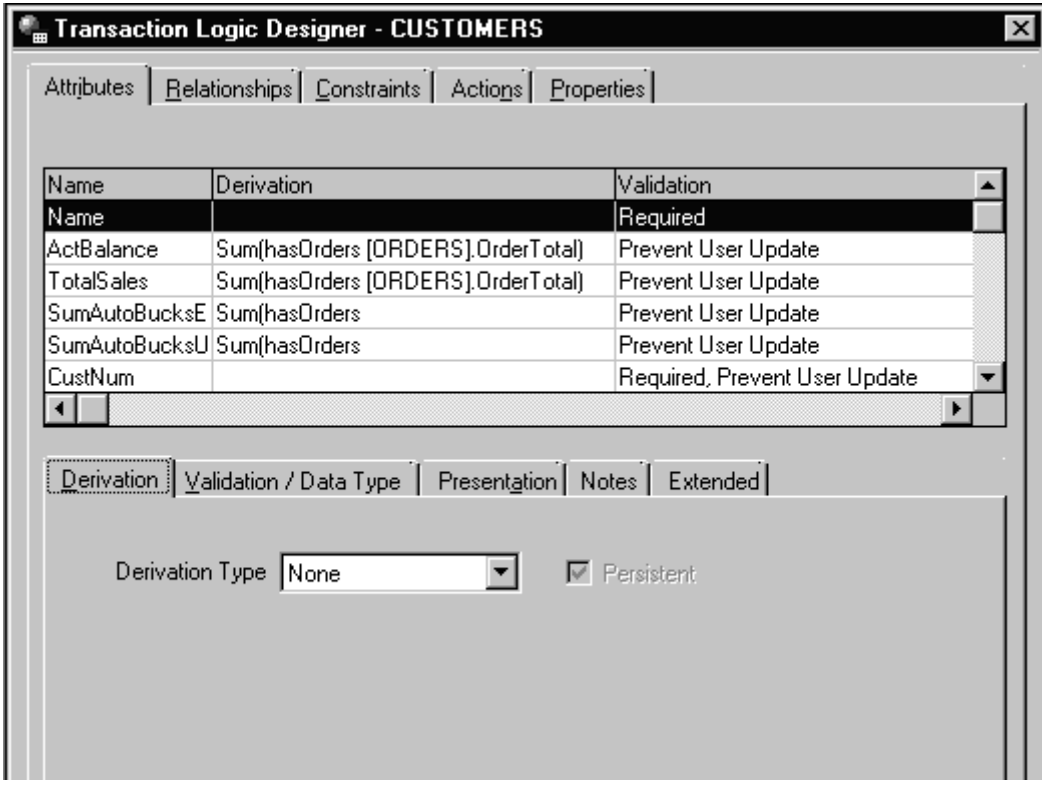

Figure 10 Transaction Logic Designer

The Transaction Logic Designer consists of several overlapping tab sheets where you can define different types of business rules and data object properties, and a Rule Builder, where you can graphically build expressions. Once you have defined business rules, you can print business rules reports by choosing File → Print Reports → Business Rules.

Attributes tab

The Attributes tab of the Transaction Logic Designer has a read-only grid of all the attributes in the selected data object, with their attribute-level rule information. Rules are not input directly into the grid. This tab also contains a tab control with tabs for each type of attribute-level rule. To define an attribute-level rule, select an attribute in the grid, and click one of the following tabs: Derivation, Validation / Data Type, Presentation, or Notes.

Also on this tab you can add, modify, and delete attributes, and define extended properties for them. For information about these tasks, see “Working with coded values lists” on page 95.

Note: The Presentation tab is not available if you have not purchased presentation design capabilities for the Versata Logic Suite.

Derivation tab

The Derivation tab allows you to enter rules that define how an attribute's value is derived when inserts or updates to the data object occur.

Select an option from the Derivation Type box. Types of derivation rules available include sums and counts (which are aggregates of child record values), parent replicates, formulas, and defaults.

Individual combo boxes list valid data objects and/or attributes to build sum, count, or parent replicate rules.

- **Sum Rules**. A combo box lists children of the selected data object. A second combo box lists attributes in the selected child data object.
- **Count Rules**. A combo box lists children of the selected data object.
- **Parent Replicate Rules**. A combo box lists parents of the selected data object. A second combo box lists attributes in the selected parent data object.

For all derivations other than default, a Persistent check box appears next to the derivation type box. The setting of this option determines whether the attribute is stored or virtual. By default, the check box is enabled, indicating the attribute is stored. For information about virtual attributes, see “Virtual attributes” on page 104.

For parent replicates, a Maintained check box appears. The setting of this option determines whether previously calculated replicates are recalculated when the parent data object's attribute is updated. Enable the Maintained option if you want updates to the parent attribute named in the parent replicate rule to cascade to child attributes. Disable this option to prevent cascading updates to children, if you want the parent replicate to occur on the initial value only. This option is disabled by default.

For sums and parent replicates, a dfn (definition) button is available. Click this button to go to the derivation rule for the referenced attribute in the parent data object.

For sums, counts, defaults and formulas, two other buttons are available, a browse button that invokes the Rule Builder, and a syntax checker button that opens the Syntax Checker.

- In the Rule Builder you can point and click to enter a qualification expression for the sum or count, a literal number value or quoted string for the default, or a calculation expression for the formula. For more information, see “Rule Builder” on page 230.
- The Syntax Checker checks whether the expression you entered is syntactically correct. If an error exists, a message is displayed to alert you. The Syntax Checker verifies the internal consistency and correctness of the rule expression. It does not check for inconsistencies or errors between rules, such as cyclical dependencies. It does not verify the compatibility of attribute data types.

Validation/Data Type tab

The Validation / Data Type tab allows you to enter rules that define limitations for attribute values. These limitations can be based on a user-defined condition or on a specified list of values in a Coded Values List, as indicated in the Validation Type box.

- If you select Condition in the Validation Type box, you can enter a conditional expression to limit the valid values for the attribute. Click the browse button to open the Rule Builder, where you can enter the expression. After writing the expression and closing the Rule Builder, you can verify the syntax of your condition is correct by clicking the syntax checker button.

  You also can enter a brief error message to display to users when data they enter causes the specified validation condition to be evaluated as FALSE.You can use the system-supplied error message of `"Rule <condition-text>: Validation violation"` by leaving the Validation Error text box blank
- If you select Coded Values List in the Validation Type box, you can complete the Coded Values List Manager.

Coded Values List Manager

Use the Coded Values List Manager to select a coded values list to provide valid values for an attribute.

- To select an existing coded values list, select it in the Coded Values Lists list box on the right and click OK.
- To select a data object that is not yet designated as a coded values list, select it in the Data Objects in the Repository list box and click the unfold button to move it to the Coded Values Lists list box. Then select it and click OK.

- To create a new coded values list, click the New button. The Create Coded Values List dialog opens. Enter a name and stored value type and click OK. The new data object appears in the Coded Values List list box. Select it and click OK.

Prevent User Updates check box

The Validation/ Data Type tab allows you to specify updatability for attributes. Enable Prevent User Updates to make this attribute non-modifiable by users running the application. This option is the default for derived attributes that the Versata Logic Server calculates and for virtual attributes, but it can be used for any attribute in which you do not want end users to enter values. This option should always be enabled for attributes with an Autonumber data type.

When this check box is enabled, end users running the application will see this non-customizable error message if they try to make an entry in the field: `Attribute <attribute_name> in data object <data_object_name> is not alterable.`

You can disable this option for any attribute that has it enabled by default, including virtual attributes. For an attribute with this option enabled, a user can enter an attribute value for a newly inserted record. If the attribute is derived, the newly entered value is overridden by the derivation when the record is saved.

Value Required check box

The Validation/ Data Type tab allows you to specify nullability for attributes. Enable Value Required to require users to provide a value in any field representing this attribute at run time. Disable this option to permit NULL values to be stored in the server.

When this check box is enabled, end users running the application will see this non-customizable error message if they try to save without making an entry in the field: `'<Field_Name>' Requires Non NULL Value.`

Data Type combo box

The Data Type combo box identifies the data type defined for this attribute in the data model. You can make a selection from the combo box to select a different data type. Available data types are:

- **Text**. For a Text attribute, you need to enter the number of characters permitted for the attribute value in the Size field. Up to 255 characters are permitted. The defined attribute size is used to determine the width of the attribute's controls or elements in run-time applications.

  You also need to enter a sub type. The following sub-types are supported:

  - **Variable Length**. This is the default sub type.
  - **Fixed Length**.

- **Memo**.
- **Number**. For a Number attribute, you need to enter a sub type. The following sub types are supported:
  - **Byte**.
  - **Integer**.
  - **Long Integer**.
  - **Double**.
  - **Single**.
  - **Decimal**. For a Decimal, you need to enter values for precision, the total number of digits stored for the attribute, and for scale, the total number of decimal places stored for the attribute.
- **Date/Time**. For a Date/Time attribute, you need to enter a sub type. The following sub types are supported:
  - **Date and Time**. This is the default sub type.
  - **Date**.
  - **Time**.
- **Yes/No**.
- **Currency**.
- **LongBinary**.
- **AutoNumber**.

The Transaction Logic Designer checks data type changes, prohibiting changes between mismatched types, changes to indexed attributes and key attributes, and changes to AutoNumber when data already exists in attributes.

For information about data type mappings between the Versata Logic Suite and supported RDBMSs, see “Data type mapping between the Versata Logic Suite and RDBMSs” on page 40.

Presentation tab

Note: This tab is not available if you have not purchased presentation design capabilities with the Versata Logic Suite.

The Presentation tab allows you to define presentation properties for an attribute in generated applications, including:

- **Caption**. Appears as a label for the attribute.
- **Format.** Specifies the appearance of numeric and date fields at run time. Specifies a format that determines how this attribute's data is displayed to users. For example, a currency format might be $#,##0.00;($#,##0.00).

  The format you specify here is used by default wherever this attribute is displayed on application forms or pages. For Java applications, Versata Logic Studio also allows you to modify the format through a property sheet for the attribute's graphical control.

  For more information about supported formats and modifying them, see the appendix on localization in the *Application Developer Guide*.

Note: Users can enter values for Date Time attributes in any form. When the cursor leaves the attribute cell, the value is formatted to fit the assigned format, by default the universal form (yyyy-mm-dd hh:mm:ss).

- **Status Bar Message**. Appears in the status bar at the bottom of the application window when the attribute is selected (Java applications only).
- **Archetype Name**. Determines the control or element to be used for the field.
- **Layout by Default**. De-select this check box if you do not want the attribute to appear on forms or pages by default. This is particularly useful for derived attributes that are used in calculations but do not need to be displayed.

Notes tab

The Notes tab allows you to record descriptions and comments for each attribute in the selected data object. This information is especially useful in a team development environment.

Relationships tab

The Relationships tab provides information about parent-child relationships for the selected data object. This tab allows you to modify referential integrity rules, specify customized error messages to appear when referential integrity violations occur, and specify relationship-level presentation properties.

Also on this tab you can add, modify, and delete relationships, and define extended properties for them. For information about these tasks, see "Working with relationships" on page 107.

Referential Integrity tab

The Referential Integrity tab allows you to modify Versata Logic Server's default referential integrity rules to preserve relationships between data objects when updates occur. This tab contains an Enforce referential integrity check box. Separate sets of referential integrity rule option buttons exist for parent updates, parent deletes, and child inserts/updates:

On Parent Update

- Choose Prevent If Children to prevent changing the primary key in a record in the parent data object if there are related records in child data objects.

  For example, you establish a relationship between a Customers (parent) data object and an Orders (child) data object. If a user tries to update the primary key for a customer that has outstanding orders, the update is not permitted.
- Choose Update Children to update the foreign key in all related records in the child data object when the primary key in a parent record changes.

  For example, you establish a relationship between a Customers (parent) data object and an Orders (child) data object. If the primary key in a record in the Customers data object is updated, the foreign keys for all order records for that customer are also updated.

On Parent Delete

- Choose Prevent If Children to prevent deleting the record in the parent data object if there are related records in a child data object.

  For example, you establish a relationship between a Customers (parent) data object and an Orders (child) data object. If a user tries to delete a record for a customer that has outstanding orders, the deletion is not permitted.
- Choose Delete Children to delete related records in a child data object when a record in the parent data object is deleted.

  For example, you establish a relationship between a Customers (parent) data object and an Orders (child) data object. If a user deletes a record in the Customers data object, all order records for that customer are also updated.
- Choose NULL Children Foreign Key to nullify the foreign key in related records in a child data object when a record in the parent data object is deleted. This option deletes the child record's pointer to the parent while preserving child data.

  For example, you establish a relationship between a Department (parent) data object and an Employees (child) data object. If a department is deleted, each employee record that has a foreign key value corresponding to the deleted department's primary key is updated by setting the foreign key (for example, the WorksForDeptNum field) to NULL. Those employees with the WorksForDeptNum field set to NULL can be reassigned to a new department and employee records can be updated with new foreign key values.

On Child Insert/Update

- Choose Prevent If No Parent to prevent inserting a record into a child data object when the appropriate related record does not exist in the parent data object.

  For example, you establish a relationship between a Customers (parent) data object and an Orders (child) data object. If a user tries to insert an order for a customer who does not yet exist in the Customers data object, the insert is not permitted.
- Choose Insert Parent If None to add a record in the parent data object and fill in the foreign key when a user adds a related record in a child data object. This option provides a good way to implement time-based summary data, for example monthly forecasts.

  For example, you establish a relationship between a Daily Orders (parent) data object and an Orders (child) data object. You can automatically maintain summary data in the parent based on order activity by creating a sum rule for a DailyTotal attribute in the Daily Orders data object based on the Amount attribute in the Orders data object, for example, DailyTotal = sum(Orders.Amount).

Error Messages While Preventing frame

The Error Messages While Preventing frame allows you to enter custom messages that appear when a user attempts a parent update, parent delete, or child insert/update that violates a prevent referential integrity rule. A blank text box appears in the frame for each prevent rule you define. If you do not enter a message, Versata Logic Server uses the default error message.

Delete Parent Error Message

If you enable Prevent If Children on Parent Delete, you can enter a brief message to display when a user attempts to perform an invalid operation.

You can use the system-supplied error message of `"Delete Rejected Because There are existing <child-data object-name> found for <parent-data object-name>"` by leaving the Delete Parent error text box blank. For example, if an end user tried to delete a customer with unpaid orders, a system-supplied error message similar to this example would appear:

```
'Delete Rejected Because There are existing ORDERS found for
CUSTOMERS'
```

Update Parent Error Message

If you enable Prevent If Children on Parent Update, you can enter a brief message to display when a user attempts to perform an invalid operation.

You can use the system-supplied error message of `"Update Rejected Because There are existing <child-data object-name> found for old <parent-data object-name>"` by leaving the Update Parent error text box blank.

Insert/Update Child Error Message

If you enabled Prevent If No Parent on Child Insert/Update, enter a brief message to display when a user attempts to perform an invalid operation.

You can use the system-supplied error message of "`<parent-data object-name> not found for <child-data object-name>`" by leaving the "Insert/Update Child" error text box blank.

Presentation tab

The Relationships:Presentation tab allows you to specify customized captions for transitions from parent to children and for transitions from children to parent.

Note: This tab is not available if you have not purchased presentation design capabilities with the Versata Logic Suite.

Extended tab

The Relationships:Extended tab allows you to define extended properties for the relationship. For information about this task, see “Relationships tab of Transaction Logic Designer” on page 111.

Constraints tab

The Constraints tab allows you to define data object-level constraints that enforce multiple attribute conditions for data validation. This tab provides a grid that lists information for all constraints defined for the selected data object. When the Constraints tab is selected, Add Constraint and Delete Constraint are available in the Edit menu.

- The Constraint Name field allows you to specify a unique name for a constraint.
- The Condition field allows you to enter an expression describing the constraint's condition, optionally using the Rule Builder. The condition can be one of two types:
  - An Accept When type indicates that an update to the data object is rolled back if the condition is not true.
  - A Reject When type indicates that an update to the data object is rolled back if the condition is true.
- The Error Message field allows you to specify a customized error message that appears when the constraint is violated. You can use the system-supplied error message of "`Constraint: <constraint-condition> is violated`" by leaving the Error Message text box blank.
- The Error Attribute field allows you to specify the attribute in which the cursor is placed after a constraint is violated and the error message is dismissed.

Actions tab

The Actions tab allows you to incorporate custom code into Versata Logic Studio-generated business rule components. Action rule code calls a specified action (method) to be executed when a defined condition evaluates to true. This tab provides a grid that lists a name and description for each business rule action defined for the selected data object. Action information is not input directly into the grid.

When the Actions tab is selected, Add Action and Delete Action are available in the Edit menu.

- The Action Name field enables you to specify a unique name for the action rule. Code generated by Versata Logic Suite refers to the action rule by this name.
- In the Event Condition field, enter an expression to define the condition that must evaluate to True for a call to be executed to the specified action (method call). The Rule Builder is available to complete this field as necessary. The Syntax Checker button also is available.
- In the Action (Method Call) field, specify the method to be executed when the condition evaluates to True. The Rule Builder's Methods list box and the Enterprise Object Browser are available for you to select a method. The method can be local, inherited, or from an object outside the data object's hierarchy. If the method is from an outside object, you must reference it in this format: `<object name>.method`. Note that such methods must be static.
- The Description field is available for documentation of the action's purpose and implementation. This information is especially useful in a team development environment.

Note: For information about using the Rule Builder to complete the Event Condition and Action (Method Call) fields, see "Rule Builder" on page 230.

Properties tab

The Properties tab has tabs to define data object presentation rules (with presentation design only) and other data object characteristics. For information about using this tab, see “Properties tab of the Transaction Logic Designer” on page 90.

Rule Builder

Use the Rule Builder to create business rule expressions graphically, limit typographical errors, and help to ensure that rule syntax is correct. The Rule Builder has lists and buttons with expression elements. Click a list item to include it in an expression. The Rule Builder’s contents vary according to the type of rule being defined when it is opened.

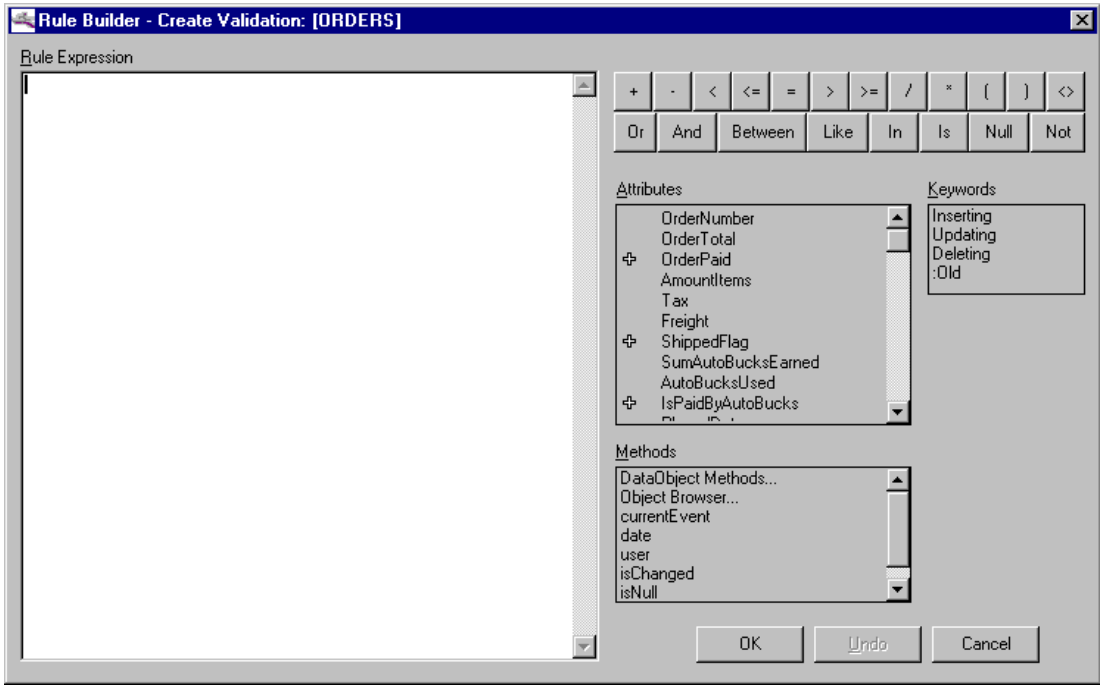

Figure 11 Rule Builder

Rule expression elements may include the following:

- **Data object attributes**. Click the attribute from the selected data object to enter it in the Rule Expression text box. The list of attributes uses standard outline controls. Click the plus (+) symbol to the left of an attribute name to display valid values for an attribute that has a Coded Values List validation rule. You can click a value to enter it in the Rule Expression text box.
  - The list of attributes changes according to the data object selected.
  - If an attribute name contains spaces, it is enclosed in quotes within the rule expression.
- **System-supplied or developer-defined methods**. Listed methods include `currentEvent()`, `date()`, `isChanged`, `isNull`, and `isOldNull`. You also can click listings to select from methods listed in the Enterprise Object Browser. You can open this browser to display the methods from `versata.vls.DataObject`, or you can open the browser to display all methods. The Methods list box also includes many methods that start or modify processes through the Process Logic Add-On. For information about these methods, see the *Logic Integration Guide*.

  Methods can be included when you are entering qualification expressions for sum, count, attribute validation, or constraint rules, formula expressions for formula rules, default expressions for default rules, or event conditions for action rules.

Note: The `isNull` and `isOldNull` methods can be used to indicate whether the value of an attribute is NULL where the argument is of type String and provides the attribute name.

- **Keywords, including `Inserting`, `Updating`, `Deleting`, and `:Old`**. Keywords are available when you are entering formula, sum, or count derivation rules for attributes or data object constraints.

  The `:Old` keyword allows you to differentiate between a changed attribute value and its value before the transaction that caused the change. You can refer to the value that existed before the change with a `:Old` prefix. For example, use `:Old` in the conditions for data object constraints that prevent updates and deletes.
- **If-Then conditions, including `If-Then`, `If-Then-Else`, `If-Then-Elseif-Else`, and `IIF()` constructs**. These constructs are available for formula rules expressions only. Once you have clicked a construct to include it in the expression, to complete the expression, fill in the `<condition>` and `<expr>` parameters.

Note: When opened from the Actions tab of the Transaction Logic Designer, the Rule Builder includes a Process Model Browser button. Use this button only if you have installed the Process Logic Add-On.

For more details about supported syntax for rule expressions, see “Business rule syntax” on page 244. For instructions for building rule expressions, see “Building rules expressions in the Rule Builder” on page 239.

Procedures for defining business rules

This section provides procedures for defining specific types of rules. For an overview of the business rules definition process, see page 213.

Defining a derivation rule

Derivation rules define how an attribute is computed when an update occurs. Types of derivation rules include sums, counts, parent replicates, formulas, and defaults.

For information about the Transaction Logic Designer tab where you define derivation rules, see “Derivation tab” on page 221.

To define a derivation rule:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Click the Attributes:Derivation tab in the Transaction Logic Designer.
3. Select the attribute in the grid.
4. On the Derivation tab, select the type of rule from the Derivation Type combo box. Different text fields appear according to the type selected.
5. Select or make entries to the fields that appear for the rule type you selected:
   - Sum rules
     - Select a parent data object and attribute from the combo boxes.
     - Indicate whether the attribute whose value is defined by the sum rule is stored or virtual. To indicate the attribute is stored, enable the Persistent check box. To indicate the attribute is virtual, disable the check box.
     - (Optional) Enter a qualification expression in the text box. A qualification expression limits the records to be included in the sum to those that meet the specified condition.
   - Count rules
     - Select a parent data object from the combo box.
     - (Optional) Enter a qualification expression in the text box. A qualification expression limits the records to be included in the count to those that meet the specified condition.

- Parent replicate rules
  - Select a child data object and attribute from the combo boxes.
  - (Optional) Click the Maintained check box if you want changes in the parent record to cascade to the child. This option is disabled by default, so parent record changes do not cause the record to change.
- Formula rules

  Enter a formula expression in the text box. This expression should be a calculation of other attribute values from the same record.
- Default rules

  Enter a default expression in the text box. This expression can be a literal number value or a quoted string.

6. If you want the derived attribute to be virtual rather than stored, disable the Persistent check box. For information about virtual attributes, see page 104.
7. For any rule where you enter an expression, click the browse button to write the expression in the Rule Builder.
8. After defining the derivation rule and closing the Rule Builder, you may want to verify the syntax with the syntax checker.
9. Choose File → Save Transaction Logic**.**

Note: All attribute references in a rule expression must be local to the data object on which the rule is being defined. Use derivations to reference attributes from related data objects.

When defining formula rules with divide operations, check to make sure that no divide by zero equations will occur. This type of equation will cause an error.

If you define a formula rule for an attribute, its data type, subtype, and length information is not used, except to determine the archetype for presentation formatting.

When selecting an atttribute to be replicated for a parent replicate rule, be sure that its data type matches the type of the attribute with the rule defined.

If you define a parent replicate rule where the attribute to be replicated has a data type of LongBinary, invalid Java syntax may be generated, resulting in compile errors.

The syntax checker no longer perform attribute validation in formulas in order to allow constants to be used. Any errors are raised at run time rather than design time. Syntax checking still occurs at design time.

Deleting a derivation rule

To delete a derivation rule:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Select the attribute with the rule to delete.
3. On the Attributes: Derivation tab, select None in the Derivation Type combo box.
4. Choose File → Save Transaction Logic.

Defining a condition validation rule

Condition validation rules enable you to enforce single attribute conditions for data validation. For information about the Transaction Logic Designer interface for defining this type of rule, see "Validation/Data Type tab" on page 222.

To define a condition validation:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Click the Attributes:Validation/Data Type tab.
3. Select the attribute in the grid.
4. Select Condition in the Validation Type frame.
5. Enter a qualification expression in the Condition text box to indicate the limitations for valid attribute values. Click the browse button to use the Rule Builder.
6. In the Validation Error field, enter a customized error message or accept the system-supplied message.
7. Choose File → Save Transaction Logic.

Defining a coded values list validation rule

Coded values list validation rules enable you to limit values for an attribute to a defined list of values in another data object. In run-time applications, attributes with coded values list validation rules display as drop-down lists where users can select from a list of values but not enter a different value.

For information about designating a data object as a coded values list and entering valid values, see "Defining a coded values list" on page 96.

To define a rule that uses a coded values list to validate user input:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Select the Attributes:Validation/Data Type tab.
3. Select the attribute to be validated.
4. Select Validation Type:Coded Values List.
5. Click the browse button in the Coded Values List Validation text box to open the Coded Values List Manager. Use this dialog to select the data object to be used as a coded values list.
   - To select a data object that is already designated as a coded values list, select it in the Coded Values Lists list box on the right and click OK.
   - To select a data object that is not yet designated as a coded values list, select it in the Data Objects in the Repository list box and click the > button to move it to the Coded Values Lists list box. Then select it and click OK.
   - To create a new coded values list, click the New button. The Create Coded Values List dialog opens. Enter a name and stored value type and click OK. The new data object appears in the Coded Values List list box. Select it and click OK.
6. Choose File → Save Transaction Logic.

Note: There is only one way to stop using a coded values list. On the Attributes:Validation/Data Type tab, select the data object name in the Coded Values List text box and press Backspace. If you previously have deployed the business rules, redeploy them so that Versata Logic Studio generates new code without the coded values list.

If you choose the Coded Values List option button, but do not specify a data object to use as a coded values list, the validation rule is recorded as a condition validation rule at repository load time.

Defining a constraint

Constraints enforce validation conditions on database updates. They apply to all updates of the data object, rather than to updates of specific attributes.

For information about the Transaction Logic Designer tab where you can view, define, and modify constraints, see “Constraints tab” on page 228.

To define a constraint:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Click the Constraints tab in the Transaction Logic Designer.
3. Choose Edit → Add Constraint.

4. Enter the name of the constraint.
5. Enter the conditional expression in the Condition text box. The expression indicates when the constraint fires.
6. Select Accept When or Reject When to indicate whether to roll back the update when the condition evaluates to False (Accept When) or to roll it back when the condition evaluates to True (Reject When).
7. Enter a customized error message or accept the system-supplied message.
8. Select the error attribute in which the cursor is placed after a constraint evaluates to true and the error message is dismissed.
9. Choose File → Save Transaction Logic.

Note: All attribute references in a constraint's conditional expression must be local to the data object on which the constraint is being defined. Use derivations to reference attributes in related data objects.

You may define multiple constraints on a data object. All constraints are evaluated when an update to a data object occurs. A transaction is rolled back when the conditional expression for a Reject When constraint evaluates to True or when the conditional expression for an Accept When constraint evaluates to False.

If you use the keyword `NULL` in the conditional expression for a constraint, keep in mind that the `:New` value is set to `NULL` on delete and the `:Old` value is set to `NULL` on insert. These settings could cause unexpected errors if you do not take them into account when defining the constraint.

Defining an action rule

Action rules are calls to methods that are executed when data meet certain conditions. They allow you to extend Versata Logic Suite-generated rule components from within the Transaction Logic Designer. Creating audit records and notifying management when a customer has placed a large order are examples of processes that action rules can automate.

For information about the Transaction Logic Designer tab where you can view, define, and modify action rules, see "Actions tab" on page 229.

To define an action rule:

1. Determine how to implement the method to be called by the action rule. You can use a method from a packaged EJB component. You can create your own class and add the method to it. You can subclass the `versata.vls.DataObject` class and add the method to it. You can add the method directly to the data object where you are defining the action rule.
2. Register the class so its methods are available in the Enterprise Object Browser.

3. If the method belongs to a subclass of `versata.vls.DataObject`, add the new subclass to the repository in the Other Files folder.
4. Double-click the data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
5. If the rule will be calling a method from a subclass of `versata.vls.DataObject`, record this subclass as the superclass for the currently selected data object. Enter the superclass on the Properties:Data Access tab.
6. Click the Actions tab.
7. Choose Edit → Add Action**.**
8. Enter a name for the action.
9. Use the Rule Builder to enter a conditional expression in the Event Condition text box. The expression indicates when the action fires.
10. Enter the method to be executed, and any arguments, in the Action text box. Enter the method name as the name of the action. If the method is not a member of the data object itself or of its superclass, you must include the object name, in the following format: `<object name>.<method name>`.

    The Rule Builder lists a few standard utility methods. You can double-click one of these methods to enter it as the action. The Methods list box also includes many methods that start or modify processes through the Process Logic Add-On. For information about these methods, see the *Logic Integration Guide.*
11. You can click the browse button to open the Enterprise Object Browser and select a method.
12. Enter a description of the action for other developers.
13. Choose File → Save Transaction Logic.

Defining a presentation rule to select a non-default archetype for an attribute

Note: This type of rule is not available if you have not purchased presentation design capabilities with the Versata Logic Suite.

Archetypes define the controls or elements generated for attributes in an application. By default, the archetype for an attribute or element depends on the attribute data type, but you can override the default by changing the presentation rule to use another archetype.

Note: Archetypes and presentation rules are available in the presentation design only.

To specify a non-default archetype for an attribute at the repository level:

1. Double-click a data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Click the Attributes:Presentation tab.
3. Select the attribute for which you are choosing an archetype.
4. Click the browse button for the Archetype Name field to open the Choose Archetype dialog.
5. Select an archetype from the list in the dialog and click the OK button.
6. Choose File → Save Transaction Logic.

The archetype you selected is now the default archetype for this attribute for all applications built from this repository.

Note: To specify a non-default archetype for an attribute in one application, use the Attributes tab on the RecordSource properties sheet in the Application Designer.

Defining a presentation rule to add an image to a data object in a Java application

You may use any `.gif` or `.jpg` image to represent a data object on forms and appear on toolbar buttons in a Java application. To do so, specify the image in a presentation rule for the data object.

Note: This type of rule is not available if you have not purchased presentation design capabilities with the Versata Logic Suite.

To assign an image to a data object:

1. Double-click the data object in the Versata Logic Studio Explorer to open the Transaction Logic Designer.
2. Click the Properties:Presentation tab.
3. Click the browse button for the Image Reference text box.
4. Navigate to the image file and double-click it.
5. Choose File → Save Transaction Logic.

Versata Logic Studio stores a copy of the selected image file in the `\Images` subdirectory of the repository directory.

Building rules expressions in the Rule Builder

To avoid typographical and syntax errors in rules, use the Rule Builder to build expressions graphically.

For details about the Rule Builder interface, see "Rule Builder" on page 230.

To build a rule expression:

1. Review the guidelines in "Business rule syntax" on page 244.
2. From any place in the Transaction Logic Designer in which you are creating an expression, click the browse button.
3. If you want to use conditional language in an expression for a formula derivation rule, click a construct from the `If-Then` Conditions list to enter it in the Rule Expression text box. Fill in the `<condition>` and `<expr>` parameters.
4. For all expressions, click the attributes, keywords, and/or operator buttons as necessary, to enter each item that you need in the Rule Expression text box.
5. To include a method in a rule, find it in the Methods box and double-click it. Enter arguments as necessary. To include a method not listed in this box, click the DataObject Methods listing or the Object Browser listing to open the Enterprise Object Browser and select a method.
6. If necessary, enter additional expressions to achieve the processing logic you need.
7. Repeat steps as needed to build your expression.
8. Click the OK button to save the definitions and close the Rule Builder.
9. Check the syntax of the expression by clicking the syntax checker button.

Note: Be sure to include spaces between variables and operators in rule expressions. If you do not include a space between a variable and operator, the syntax checker returns an error.

Generating business rules reports

You can generate business rules reports that provide summary records of the data model and business rule definitions in the Transaction Logic Designer. These reports can be a useful tool for allowing users to review the business rules. You can generate reports to the screen, to a file, or directly to a printer.

Versata Logic Suite reports are generated using the Crystal Reports tool. The original reports are located in the `\Reports` subdirectory of the product installation directory.

To generate a business rules report:

1. Choose File → Print Reports → Business Rules to open the Business Rules Reports dialog.
2. In the dialog, move the data object(s) for which you want to generate reports into the Selected Data Objects list box on the right.
3. Select the type of data object rules and/or attribute rules to include.
4. Select the report output. You may want to print to the screen first, in order to verify the format and content before you save it to a file or send it to the printer.
   - For file output, specify the output file name. The file name must be unique. You cannot choose an existing file name and overwrite the previous report information with this new report information.
   - For printer output, you must have a default printer set in order to specify this option.
5. Click the Print button.

Note: To review the Java code that implements rules for a data object, open the data object's component file in the Code Editor. (To do so, in the Files view of the Versata Logic Studio Explorer, click the Files button, and double-click the data object's implementation file.) Once the file is open, press CTRL+P or choose File → Print <file_name>.

Business Rules Report dialog

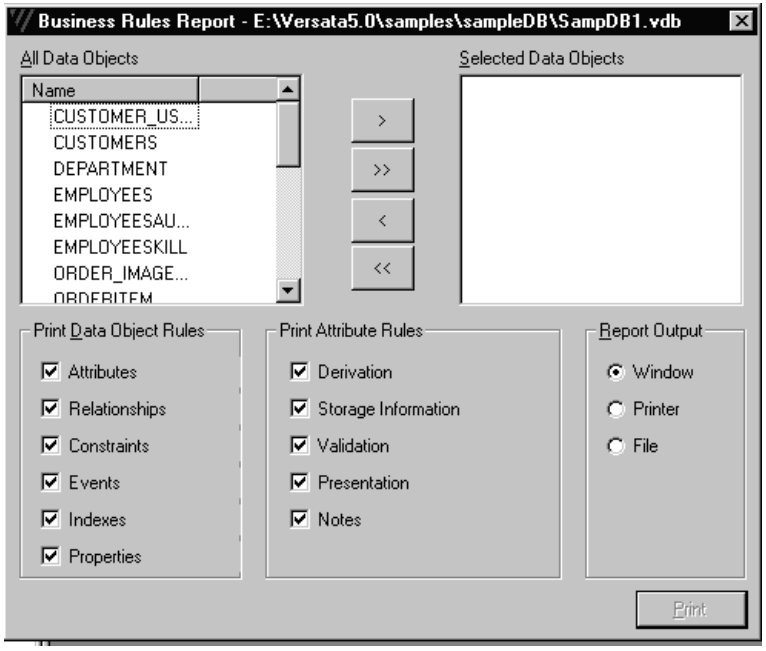

Figure 12 Business Rules Report dialog

Printing data object rules

Choose one or more of the following types of business rules to be included in the business rules report for one or more data objects.

Type of business rule	Information in report
Attributes	Attribute definitions for any or all of the attribute rules options you select
Relationships	Parent and child data object(s), and cascade options for each of the current data object's parents and children
Constraints	Constraint name (in bold typeface), condition type, rule, and (optional) error message for each currently defined constraint
Actions	Action name (in bold typeface), and one or more of the options for description, condition, and method call
Properties	Singular and plural captions (with presentation design only)

Printing attribute rules

To print a report on attribute rules, enable the Attributes option in the data object rules list. Then choose one or more of the following types of attribute rules to be included in the business rules report.

Type of Attribute Rule	Information in Report
Attribute Type Information	Attribute data type and size, if appropriate
Derivation	Defined sums, counts, parent replicates (including its maintained/unmaintained status), formulas, and defaults
Validation	The Coded values list or condition used to validate the attribute and the validation error message, if applicable
Presentation	Captions, formats, status bar messages, default archetypes, and whether an attribute will appear in the default layout (with presentation design only)
Notes	Attribute descriptions and comments

Updating business rules

As you more clearly define the needs of users, as business requirements are updated, and as you test and debug previously defined rules, you will need to update business rules defined in the Transaction Logic Designer.

To update the business rules:

1. Use the Transaction Logic Designer to make changes to business rules.
2. Rebuild and compile rules.
3. If the rule changes include any of the following, you need to use the Server Manager wizard to redeploy the data objects with changed rules to the development database.
   - Constraints
   - Updatability validation rules
   - Nullability validation rules
   - Referential integrity (If you have elected to enforce referential integrity on the database server)
4. Use the Versata Logic Server Deployment wizard to deploy the updated rules components to the development Versata Logic Server.
5. If the rule changes include any of the following, rebuild and compile the client application.
   - Presentation rules (with presentation design only): attribute-level (captions, formats, status bar messages), data object-level (captions, images), or relationship-level (captions)
   - Data type definitions
   - Coded values list validation rules
   - Updatability rules
6. Run the application locally against the development database to test the changes.
7. If your application is in production, complete the following additional steps:
   - If the rule changes included types listed in step 3, use the Server Manager wizard to deploy the data objects with changed rules to the production database.
   - To make the Server Manager automatically select for deployment the data objects whose rules have changed, enable Auto-select Data Objects in the Connect for Auto Selection dialog. Otherwise, in the Select Data Objects dialog, manually select the data objects with changed rules. In the Data Model Deploy Options dialog, enable Synchronize the Repository with the Server.
   - Use the Versata Logic Server Deployment wizard to deploy the updated rules components to the production Versata Logic Server.

Business rule syntax

Versata Logic Suite's declarative rules language provides conventions for you to enter SQL-like rule expressions consisting of supported elements.

Rule expressions can be divided into four types:

- Conditional expressions (also known as qualification expressions)
- Formula expressions
- Default expressions
- Action expressions

Each type of rule expression has syntax particular to its use. Review the following general syntax guidelines common to all rule expressions, as well as the specific syntax conventions for each of specific types of expressions.

General guidelines for writing rules expressions

Follow these general principles when you write expressions in the Transaction Logic Designer:

- Rule expression syntax is not database-specific. The same rule expressions can be used with all database servers supported by Versata Logic Suite.
- All language in rule expressions is case insensitive, except for attribute references that use quoted identifiers.
- Attribute references in a rule expression must be local to the data object on which the rule is being defined. For sum and count qualification expressions, attributes must be local to the child data object that the rule references, rather than the parent data object on which the rule is defined.
- The use of an attribute reference to an attribute value before an update is supported via the `:Old` prefix. An attribute reference with an `:Old` prefix indicates the value of the attribute before the user update occurred. An attribute reference without this prefix indicates the current value of the attribute after the update.
- The `isNull` and `isOldNull` methods can be used to indicate whether the value of an attribute is NULL where the argument is of type String and provides the attribute name. You need to ensure there are no spaces in an attribute name in an `isNull` statement.
- Use the Rule Builder to avoid syntax errors.

Syntax for conditional expressions

Conditional, or qualification, expressions are used in the following ways:

- In sum rules, to define child condition(s) that limit child records to be summed
- In count rules, to define child condition(s) that limit child records to be counted
- In validation rules, to define condition(s) that limit the valid values for an attribute
- In constraints, to define multiple attribute condition(s) for the constraint
- In action rules, to define a condition that causes the action to be executed

The expressions may consist of combinations of supported identifiers, tokens, reserved words, methods that return a value, and/or constants.

Conditional expression syntax approximates the syntax of SQL `Where` clauses. The "`Where`" is implicit and does not have to be entered in the expression you define.

Note about using isNull in conditional expressions

If you are entering a conditional expression in the Rule Builder, and you put text like the following in the expression:

```
isNull('<arg1>') = true
```

If you select `<arg1>` and double-click an attribute to put the applicable attribute in the rule, the attribute is inserted with spaces on either side of it. These spaces cause the following error:

```
Null value encountered in '' while validating constraint
```

This error does not occur if you manually remove the spaces.

Syntax for formula expressions

Formula expressions are used in formula rules to calculate the value of an attribute. The expressions may consist of combinations of supported identifiers, tokens, reserved words, methods that return a value, and/or constants.

The ability to return a value based on a condition is supported through

- `If-Then`, `If-Then-Else`, `If-Then-Elseif-Else` statements
- `IIF` statements

Self-assignment is supported through the use of the `$value` keyword.

Syntax for default expressions

Default expressions are used in default rules to provide the value of an attribute when there is no user update. The expressions may consist of constants that return a value.

Note: Do not use methods in default expressions. If you attempt to use a method in a default expression, the literal string will be used instead of the value for the method. For example, if you typed in the method `VSSession.getUserName()`, the column would display the string `"VSSession.getUserName()"` instead of returning the value of `getUserName`.

Syntax for action expressions

Action expressions are used in action rules to define the method that is executed when a specified condition is met. The expressions may consist of methods, keywords, and attribute name identifiers passed as arguments.

Note about using LIKE in rule expressions

Use of the LIKE operator in conditional expressions for rules results in the Java compiler issuing a syntax error.

You can use a Java method to get around most cases where you might want to use the LIKE operator. You need to put the method code in and external Java file and add the file to the repository. Then you can reference the method in the rule expression.

So for example, instead of using rule text like the following:

```
Reject when UPDATING and PSWRD LIKE '%USER_NM%'
```

You can use a call to a helper object like the following:

```
Reject when SomeOtherObject.contains(PSWRD, USER_NM) = true
```

For this example, code like the following is in an external file called SomeOtherObject.java that has been added to the repository:

```
public class SomeOtherObject {
public static boolean contains(String s, String substring)
{
char [] master = s.toCharArray();
char [] sub = substring.toCharArray();
return foundSubString;
}
}
```

For more information about this workaround, see the KnowledgeBase.

Elements supported in rule expressions

Identifiers supported in rule expressions

- Rule expression identifiers are used for method names and attribute names.
- Identifiers may consist of alphanumeric or underscore ("_") characters. Identifiers may not begin with a number.
- Double-quoted identifiers are supported for use with attribute names and method names that are case sensitive or contain spaces.

Reserved words in rule expressions

The following words are reserved:

AND	END	INSERTING	OR
BETWEEN	ESCAPE	IS	SOME
DELETING	IF	LIKE	THEN
ELSE	IIF	NOT	UPDATING
ELSEIF	IN	NOT LIKE	$VALUE
TRUE	FALSE	NULL	:OLD

Constants supported in rule expressions

Rule expressions support integer, float, and string constants in standard formats, such as the following examples. Hex constants are treated as integers. Single quotes are supported for use with string literals.

Constant	Value
Integer	0, 123, -45
Float	0.5, 5e23, 45.2, 2.3e-2
String	‘a string’ ‘a string with a new line’
Hex	0xA2, 0x00F

Tokens supported in rule expressions

The following tokens are supported:

Token	Description
>	Greater than
<	Less than
(	Left Parentheses
)	Right Parentheses
*	Multiply
/	Divide
+	Plus
-	Minus
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
,	Comma

Token	Description
;	Semicolon
/*<*text*>*/	Comments
[0-9]*	Integer
[0-9]+”.”[0-9]*([Ee][+-]?[0-9]+)? | [0-9]+[Ee][+-]?[0-9]+	Float
\‘*[AsciiChars]**\’	String
0x[0-9a-fA-F]+	Hex
>	Greater than

BNF for rule expression syntax

BNF for Rule Expression Syntax			
<rule expression>	:=		<statement>;
<statement>	:=		<scalar expression or IIF>
		|	<if then else statement>
		|	<conditional expression>
<scalar expression or IIF>	:=		<scalar expression>
		|	<IIF>
<scalar expression>	:=		<term>
		|	<scalar expression> + <term>
		|	<scalar expression> - <term>
<term>	:=		<factor>
		|	<term> * <factor>
		|	<term> / <factor>
<factor>	:=		<primary>
		|	+ <primary> /*Unary Plus*/
			- <primary> /*Unary Minus*/
<primary>	:=		<literal constant>
		|	<attribute reference>
		|	<function reference>

BNF for Rule Expression Syntax *(Continued)*			
		|	(<scalar expression>)
<IIF statement>	:=		IIF (<conditional expression>, <scalar expression or IIF>, <scalar expression or IIF>)
		|	IIF (<conditional expression>, <scalar expression or IIF>)

<if then else statement>	:=		IF (<conditional expression>) THEN
			<self assign or ifelse>
			<else list>
			END IF

<self assign or ifelse>	:=		<self assignment>
		|	<if then else statement>
<self assignment>	:=		$value = <scalar expression>
<else list>	:=		<else clause>
		|	<else if> <else list>
		|	<else if>
<else clause>	:=		ELSE <self assign or elseif>
<else if>	:=		ELSEIF<conditional expression> THEN <self assignment>
<conditional expression>	:=		<Boolean term>

BNF for Rule Expression Syntax *(Continued)*			
		|	<conditional expression> OR <Boolean term
<Boolean term>	:=		<Boolean factor>
		|	<Boolean term> AND <Boolean factor>
<Boolean factor>	:=		<Boolean primary>
		|	NOT <Boolean primary>
<Boolean primary>	:=		<comparison predicate>
		|	<between predicate>
		|	<like predicate>
		|	<test for NULL>
		|	<in predicate>
		|	INSERTING
		|	UPDATING
		|	DELETING
<comparison predicate>	:=		<scalar expression> <compare ops> >scalar expression>
<between predicate>	:=		<scalar expression> BETWEEN <scalar expression> AND <scalar expression>
		|	<scalar expression> NOT BETWEEN <scalar expression> AND <scalar expression>

BNF for Rule Expression Syntax *(Continued)*			
<like predicate>	:=		<attribute reference> LIKE <literal constant>
		|	<attribute reference> LIKE <literal constant> ESCAPE <literal constant>
		|	<attribute reference> NOT LIKE <literal constant>
		|	<attribute reference>NOT LIKE <literal constant> ESCAPE <literal constant>
<test for NULL>	:=		<attribute reference> IS NULL
		|	<attribute reference> IS NOT NULL
<in predicate>	:=		<attribute reference> IN (<expression list>)
		|	<attribute reference> NOT IN (<expression list>)
<expression list>	:=		<scalar expression>
		|	<expression list>, <scalar expression>
<function reference>	:=		IDENTIFIER (<expression list>)
		|	DBMS_IDENTIFIERS (<expression list>)
<attribute reference>	:=		IDENTIFIER
		|	":OLD".IDENTIFIER
<compare ops>	:=		"="
		|	"<>"

BNF for Rule Expression Syntax *(Continued)*			
		|	“<”
		|	“>”
		|	“<=”
		|	“>=”
<literal constant>	:=		INTEGER_CONSTANTS ***[0-9]****
		|	FLOAT_CONSTANTS ***[0-9]+”.”[0-9]*([Ee][+-]?[0-9]+)? | [0-9]+[Ee][+-]?[0-9]+***
		|	STRING_CONSTANTS **\‘*[AsciiChars]**\’**
		|	HEX_CONSTANTS ***0x[0-9a-fA-F]+***

CHAPTER 8

Building and Deploying Business Objects

Chapter overview

Read this chapter to understand how the Versata Logic Studio allows you to package transaction logic and data structure information into business object files. These business object files make transaction logic operational against real data sources at run time.

After reading this chapter, you should have a basic understanding of how the files for Versata Logic Server business objects are generated, compiled, and deployed to the Versata Logic Server.

This chapter includes the following:

- "Overview of business object generation and deployment" on page 257, describes the steps involved in creating files for business objects and copying them to the Versata Logic Server, including the following:
  - "Setting deployment options" on page 257
  - "Files created during object generation" on page 259
  - "Files created during object compilation" on page 259
  - "Additional files for deployment" on page 261
  - "Deploying to IBM WebSphere Application Server 4.0" on page 262
  - "Setting up deployed objects in the Versata Logic Server Console" on page 263
  - "Redeploying business objects" on page 264
- "Using menu options to build and compile business objects" on page 265, provides instructions for building and compiling business objects directly in the Versata Logic Studio.
- "Using the Versata Logic Server Deployment wizard" on page 268, provides instructions for using the Deployment wizard to package business object files and copy them to a Versata Logic Server on IBM WebSphere Application Server 4.0 Single Server Edition. This deployment to a staging area allows you to test transaction logic before a production deployment.
- "Testing transaction logic" on page 279, provides an overview of how to test business rules once business objects have been deployed. More detailed information is available in the *Administrator Guide* and in the supported third party debugger's information.
- "Deploying business objects to a production environment" on page 281, provides instructions for copying business object files' deployed packages to a Versata Logic Server on IBM WebSphere Application Server 4.0 Advanced Edition and running a batch file to register files on WAS.

Note: For more detailed information about business object files' contents, see "Understanding Business Object Files" on page 285.

Overview of business object generation and deployment

After you have defined data objects and query objects in the Versata Logic Studio, you can build and compile files for these business objects. These files package the rules logic code, object instantiation code, and other required code into objects that can be accessible to running applications. Java files that contain business object definitions are generated, then these files are compiled into class files.

The next step is deployment, meaning packaging the compiled class files and copying them to a Versata Logic Server, so that at run time, these files can instantiate business objects as necessary. These instantiated business objects process changes to underlying data sources. This processing includes the execution of transaction logic (business rules) defined in the Transaction Logic Designer.

Deployment of business objects to the Versata Logic Server is a two-step process.

- The first step is deployment to a development environment. The Versata development environment is the Versata Logic Server on WAS 4.0 Advanced Edition - Single Server option (AES) running on Windows.

  The Versata Logic Studio provides a wizard for this task. For a description of this wizard, see “Deployment wizard user interface” on page 269. For instructions for development deployment, see “Deploying business objects to a development environment Versata Logic Server” on page 273.
- The second step is deployment to a production environment. The production environment is the Versata Logic Server on WAS 4.0 Advanced Edition (AE) running on Windows, AIX, or Solaris.

  This task involves copying of the `<repository>_Deployed.ear` file. For instructions, see “Deploying business objects to a production environment” on page 281.

Setting deployment options

Each data object and query object has deployment options that you should set before you build, compile, or deploy Versata Logic Server objects.

EJB deployment

The first option relates to whether to deploy each object as an EJB. You can deploy each data object as an entity bean and each query object as a session bean. By default, each object is deployed as a Java class file.

If you want to enable an object for remote access, deploy it as an EJB. Remote object access is required for another remote object to invoke remote methods on an object. For information about remote method invocation, see "Accessing remote objects from clients" on page 350.

When you deploy an object as an EJB, the object's files are added to the repository `.jar` file in the same manner as those for non-EJB objects, so the implementation files are available to provide object instantiation and logic processing. In addition, the object's files are packaged to create an object in compliance with the EJB standard for remote access, and the EJB object is installed on the IBM WebSphere Application Server, with the Versata Logic Server as its EJB container.

Objects are not deployed as EJBs by default, because EJB capability is required only for remote access, and each object that is deployed as an EJB slows the deployment process. To mark a data object to be deployed as an EJB, enable the Deploy as EJB Entity Bean check box on the Properties:Data Access tab of the Transaction Logic Designer. To mark a query object to be deployed as an EJB, enable the Deploy as EJB Session Bean check box on the Properties:General tab of the Query Object Designer.

Note: If you use the Deployment wizard to deploy an object as an EJB, and then later deploy it without enabling EJB deployment, the original EJB files remain in the repository `.jar` file. This does not cause problems, because the EJB is removed from the application server. To remove these files from the `.jar`, uncheck the incremental check box in the wizard dialog.

Attribute-level security deployment

For each data object and query object, you have the option of deploying the names of attributes to the Versata Logic Server so that attribute-level security can be set in the Versata Logic Server Console. Enable this option only for objects where you plan to set attribute-level security, as it can slow the deployment process.

To enable this option for a data object, enable the Deploy Attribute Security Data check box on the Properties:Data Access tab of the Transaction Logic Designer. To enable this option for a query object, enable the Deploy Attribute Security Data check box on the Properties:General tab of the Query Object Designer.

For information about setting up attribute-level security in the Versata Logic Server Console, see the *Administrator Guide*.

Note: If you deploy attribute-level security information to the Versata Logic Server, then do another deployment without this option enabled, the preexisting attribute security information remains in the Versata Logic Server Console. You need to manually remove this information if you no longer want it to be used.

Also, if you delete an attribute for which security data has been deployed, its security data is not deleted. This attribute is still displayed in the Versata Logic Server Console, even after deletion.

Files created during object generation

The following table lists the files that the Versata Logic Studio creates for each data object and query object when you elect to build objects. These files are located in the `<repository>\Source\Vls` directory, within group subdirectories, if applicable.

- For more details about each file's contents, see "Generated files for business objects" on page 290.
- For information about using Versata Logic Studio menu options to build objects, see "Using menu options to build and compile business objects" on page 265.

File	Type	Purpose of file
`<object>BaseImpl.java`	Base implementation file	Contains system-generated code, including rules.
`<object>Impl.java`	Main implementation file	Contains developer-defined, custom code.
`<object>.java`	Remote interface file (Generated only if object is to be deployed as EJB)	Defines support for transactions, threading, and security for the EJB.
`<object>Home.java`	Home interface file (Generated only if object is to be deployed as EJB)	Defines methods called by remote clients or objects to create, find, and remove instances of the EJB.
`<object>DD.xml`	Deployment descriptor file (Generated only if object is to be deployed as EJB)	Defines basic properties that determine characteristics of the invoked EJB.

Note: If the disk is full when you attempt to build objects, an error occurs. This error message incorrectly mentions a form; it should reference a data object.

Files created during object compilation

This table lists the files that the Versata Logic Studio creates for each data object and query object when you elect to compile objects. These files are located under the `<repository>\Lib` directory.

- For information about using Versata Logic Studio menu options to compile objects, see "Using menu options to build and compile business objects" on page 265.

File	Type	Purpose of file
<object>BaseImpl.class	Object base class	Executes system-generated code, including rules, for the object.
<object>Impl.class	Object class	Extends the base class to implement additional custom code.
<object>.class	Compiled remote interface file (Created only if object is to be deployed as EJB)	Used to invoke the business object's methods after home interface has been used to gain access to the EJB.
<object>Home.class	Compiled home interface file (Created only if object is to be deployed as EJB)	Used to gain access to the EJB.

Note: If the disk is full when you attempt to compile objects, a Versata termination error occurs.

Compiler defaults and option settings

By default, the compiler denoted by %JAVA_HOME%\bin\javac is used to compile Versata business objects. You can specify a different compiler to be used on the Executables tab of the Environment Options dialog. To open this dialog, choose Tools → Options from the Versata Logic Studio main menu.

For this release, the default Java compiler is <install_directory>\java\bin\javac.exe, where <install_directory> is the directory where IBM WebSphere Application Server is installed. This is the location where the JDK 1.3.0 is installed when you install it along with the IBM WebSphere Application Server.

At installation time, a batch file called setVersataEnv.bat is created in the Versata Logic Studio installation directory. This file sets the JAVA_HOME and JAVAC_OPTIONS variables. If you want to use a compiler other than the default, you can change these variables by manually editing this file. To change the javac variable, specify a different compiler on the Executables tab of the Environment Options dialog.

Note: Changing variables might not work in all cases. You need to review the compil.bat and other related files to ensure that the correct classpaths are picked up because they may be hardcoded in some places.

It is possible to invoke compiler options to set the maximum heap size for the Java compiler and to ensure that dependent Java files get compiled if necessary.

To set the maximum heap size for the Java compiler, specify the value in the following registry setting: `LOCALMACHINE/Software/<install_directory>/EnvironmentOptions/MaxHeap_ForJava`. The value is specified in bytes with a default of 64000000.

Do not define a value for the `-D` option in the `setVersataEnv.bat` file. This option specifies where compiled files are saved and interferes with Versata Logic Suite conventions.

For more information about batch files, see the *Administrator Guide.*

Additional files for deployment

You need to deploy some files to the Versata Logic Server in addition to those that are built and compiled in the Versata Logic Studio. This section summarizes these files.

Required Versata Logic Suite JAR files

During both Versata Logic Studio and Versata Logic Server installations, several `.jar` files are copied to the WebSphere installation directory. These `.jar` files are necessary for your Versata-generated applications to run. The following table provides the location and description of each of these files.

Class files	Location	Description
`vlsEJB55.jar`	`%WAS_HOME%\lib\app`	System classes - server runtime
`vlsBeans55_Client.jar`	`%WAS_HOME%\lib\app`	System classes - client classes of the context beans
`vfcEJB55.jar`	`%WAS_HOME%\lib\app`	System classes - client runtime
`vlsBeans55.jar`	`%VERSATA_HOME%\vls\lib\Versata_Logic_Server.ear`	System beans - `VLSContext` and `PLSContext`
`<repository>.jar`	`%VERSATA_HOME%\VLSComponents\Classes\<repository>.ear`	Business objects and business objects as beans. The `<application>.xml` file will have an entry for a Web module even when there are no HTML application deployed.

Optional external dependent classes or JAR files

When using external `.class` or `.jar` files, make sure that your applications can find the files. You should also plan for whether the files will be referenced by a single Versata repository or

multiple Versata repositories. Doing so can prevent duplicate copies of the files, which can be harder to manage.

Class files

If you have external dependent `.class` files for a single repository, you can simply include the `.class` files in the repository `.jar` file. If the `.class` files are for multiple repositories, put the files in the `%WAS_HOME%\lib\app` folder.

JAR files

If you have external dependent `.jar` files, you must decide if the files are needed for multiple Versata repositories or a single Versata repository. If the `.jar` files are needed for multiple repositories, put the files in the `%WAS_HOME%\lib\app` folder. For a single repository, or to keep external classes in a separate `.jar` file, use the following steps:

To keep external classes in a separate .jar:

1. Copy the external `.jar` file in the root level of the `.ear` folder:
   `%VERSATA_HOME%\VLSComponents\classes\<repository>.ear`
2. Add the name of the `.jar` in the classpath entry of the `war` manifest file:
   `%VERSATA_HOME%\VLSComponents\classes\<repository>.ear\<repository>.war\META-INF\manifest.mf`
3. The external classes that are common for all the repositories should be copied to the `%WAS_HOME%\lib\app` folder.

Deploying to IBM WebSphere Application Server 4.0

Using the Versata Logic Suite with IBM WebSphere Application Server 4.0 gives you the flexibility of deploying to either the Versata development environment or the Versata production environment.

The Versata development environment is the Versata Logic Server on WAS 4.0 Advanced Edition - Single Server option (AES) running on Windows. Use the Versata Logic Studio development and deployment wizards to build, and then deploy your applications to this environment. Doing so allows for fast application development and deployment, enabling you to quickly test and debug your applications before deploying them to the Versata production environment.

When using the Versata Studio deployment wizards to deploy to Versata's development environment, you can hot deploy or dynamically reload application components without having to restart the Versata Logic Server. Hot deploying and dynamic reloading application components also allow you to quickly test applications before deploying them to Versata's production environment.

After you have sufficiently tested your application in the Versata development environment, deploy your application `<repository>_Deployed.ear` file to Versata's production environment. The production environment is the Versata Logic Server on WAS 4.0 Advanced Edition (AE) running on Windows, AIX, or Solaris. WAS 4.0 AE provides a highly scalable Versata production environment, allowing for multiple VERSATA application server instances and clones on both local and remote machines.

- For information about deploying to Versata's development environment, see "Using the Versata Logic Server Deployment wizard" on page 268.
- For information about hot deploying to the development environment, see "Hot deploy and dynamic reloading task reference" on page 276.
- For information about deploying to Versata's production environment, see "Deploying business objects to a production environment" on page 281.
- For information about deployment of client application files, see the *Application Developer Guide*.

Setting up deployed objects in the Versata Logic Server Console

After business objects have been deployed to the Versata Logic Server, you can set up data source connectivity and security in the Versata Logic Server Console.

During deployment each business object is assigned to a Versata Logic Server data server whose type and connection properties match those used for the last data model deployment of that object. Data objects from multiple repositories can share the same data server in the Versata Logic Server Console if they are deployed using the same connection properties.

If no data server with matching connection properties is found, a new data server definition is created with the name "Data Server#", and the object is associated with the new data server. You then need to define a data server type and connection properties for this data server. This situation occurs most often when the business object represents a non-supported data source and requires a custom Versata Connector.

In the development environment, it is recommended that you use the default security manager so you can perform all security tasks, such as defining users, assigning them to roles, and assigning privileges to objects in the Versata Logic Server Console. Integration with IBM WebSphere Application Server functionality is available primarily for use in production environments.

For more information about administration and security for business objects, see the. *Administrator Guide*.

Redeploying business objects

Business objects are often redeployed to the Versata Logic Server numerous times during development, and sometimes in production as well. When you redeploy to the Versata Logic Server, you have the choice of deploying files only for business objects that have changed since the last deployment. For this choice, enable the Incremental Build of Deployed Jar option in the Choose Versata Logic Server for Deployment dialog.

In cases of redeployment, objects are redeployed so that security data in the Versata Logic Server Console need not be redefined. This is accomplished as follows:

During Versata Logic Server deployment, the system checks to see if the repository name/object name combination for each business object already exists in the Versata Logic Server. If a match is found, the determination of what happens depends on the type of object:

- Non-data server objects: new information overlays old information, but all security references remain intact.
- Data server objects: are not explicitly deployed during the Versata Logic Server deployment process. Instead, the Versata Logic Server deployment wizard creates new data servers as follows:
  - If a data object has never been deployed before, the wizard looks for an existing connection that has the same connection properties (login, password, ODBC DSN, schema, database), and if it finds a match, the wizard assigns the new data object to the matching data server.
  - If a data object has never been deployed before and the wizard cannot find an existing connection that has the same connection properties, it creates a new data server and assigns it the connection properties of the new data object.
  - If a data object has already been deployed and is assigned to a data server, the wizard does not try to match to an existing connection, nor does it cause the creation of a new connection. Instead, the object remains assigned to its current data server to ensure that no assignments made by the administrator are overwritten during deployment.

Using menu options to build and compile business objects

The Build menu in the Versata Logic Studio provides the following options for building and compiling business objects:

- The top section of the menu is context-dependent and contains the following options:
  - Rebuild Selected (Ctrl+F9)
  - Compile Selected (Ctrl+F8)

  These options apply to the currently selected object, file, or group.
  - On the Objects tab of the Explorer, you can select an object and choose one of the top menu options to rebuild or compile all of the files for that object.
  - On the Files tab of the Explorer, you can select a file for an object and choose one of the top menu options to rebuild or compile that particular file.
  - On either tab of the Explorer, you can select a group and choose one of the top menu options to rebuild or compile all files for objects in the group.
- The second section of the menu contains the following options:
  - Rebuild Components (Shift+F9)
  - Rebuild All Components
  - Compile Components (Shift+F8)
  - Compile All Components

  Select an "All" option to rebuild or compile all business objects in the repository. Select an incremental option to rebuild objects that have changed since the last build or to compile objects that have changed since the last compile.

  When a group is selected in the Explorer, the incremental and "All" options operate on the whole repository, not just the group. To rebuild or compile only objects in the selected group, choose a "Selected" option or an option from the group's Files tab right-click menu.
- The third section of the menu, if any, applies to the currently open application. For information about this section's options, see the *Application Developer Guide*.
- Rebuild and Compile menu options also are available from the right-click menus of individual files and groups on the Files tab of the Versata Logic Studio Explorer.

Note: If you select a Compile menu option, the objects to be compiled are checked and rebuilt as necessary before the compile occurs.

Errors may occur if you choose a Compile option for an individual file or group before all objects in the repository have been compiled at least once. Because classes may reference each other, you may have to compile the entire repository before you can compile an individual file or group. If your repository directory or any of its subdirectories are read-only, compiles fail.

In some cases, remote interface files may not be compiled when you choose the Compile Components option. If this occurs, choose the Compile All Components option to ensure remote interface files get compiled.

If you have made changes to column captions for a query object, an incremental build option may not properly rebuild the query object. If this problem occurs, the workaround is to close and reopen the repository, then retry the changes and the rebuild.

If you have just converted your repository to release 5.5, you may encounter generation errors the first time you select the Rebuild All Components menu option. If this error occurs, reselect this menu option. If errors continue to occur, you may need to correct duplicate role names in the repository. For information, see the *Migration Guide*.

Saving changes to rebuilt query objects

Because query objects are dependent on data objects, changes to data objects may cause changes to query objects. After you have rebuilt repository business objects, a dialog may appear asking you whether to save changes to query objects. Generally, you should click OK to save these changes and continue. However, if you have manually customized query text in the Query Object Designer, you should click Cancel in order to preserve the current text. In this case, you can make changes manually to reflect changes in underlying data objects without unnecessarily overwriting other query text.

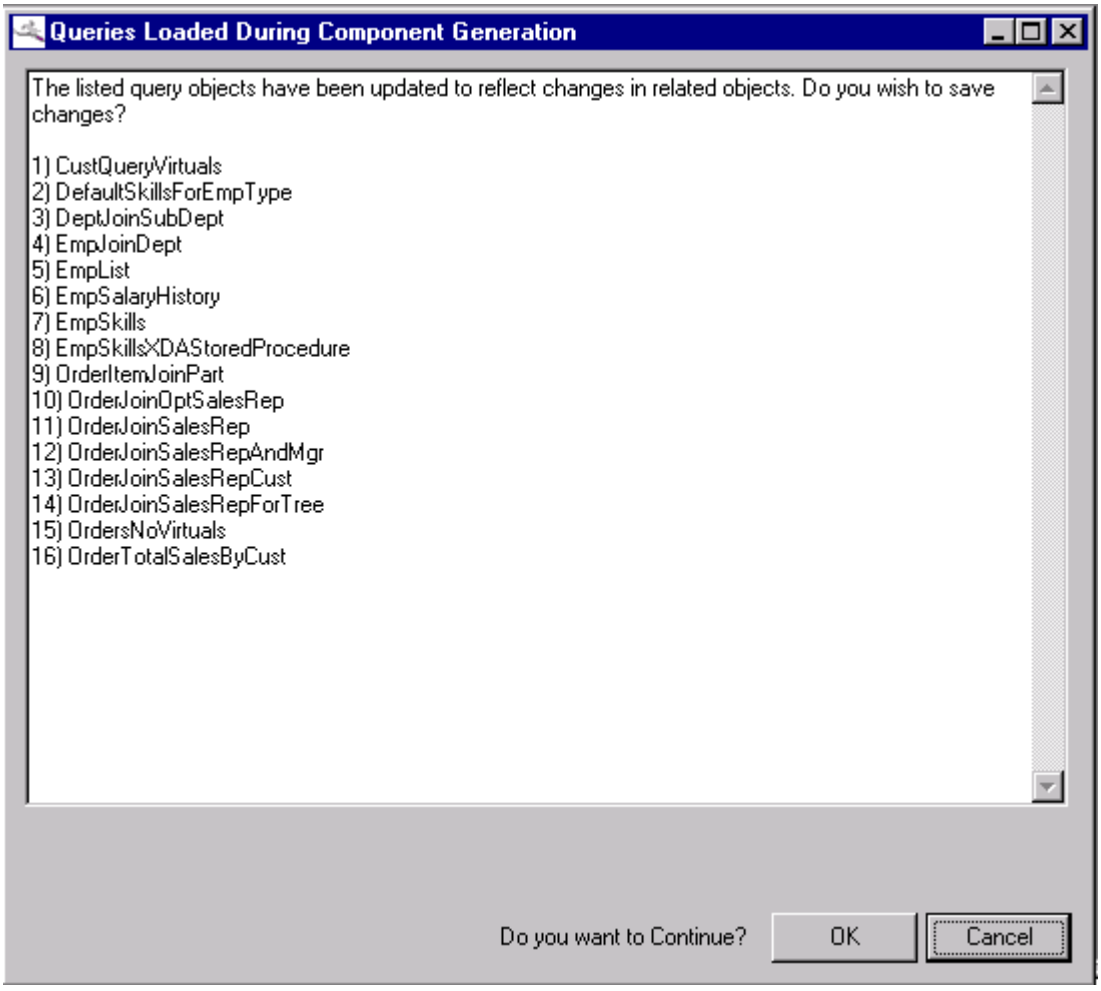

Figure 13 **Queries Loaded During Component Generation dialog**

Using the Versata Logic Server Deployment wizard

When deploying business objects, use the Versata Logic Server deployment wizard to deploy to Versata's development environment: the Versata Logic Server on WAS 4.0 AES, running on Windows.

The deployment wizard copies the business objects, and then registers the business objects to WebSphere 4.0 AES. The registration information is stored as an enterprise application inside the `<versata_logic_suite>\Config\versata-cfg.xml` file. Once you have deployed your application, open the WebSphere Console to see that your VERSATA_<repository> enterprise application is registered (Figure 14).

Note: Ensure that the `versata-cfg.xml` file is in use and appears at the top left panel of the WebSphere Console. If the `versata-cfg.xml` file does not appear, newly deployed and updated repositories will not be recognized by the Versata Logic Server.

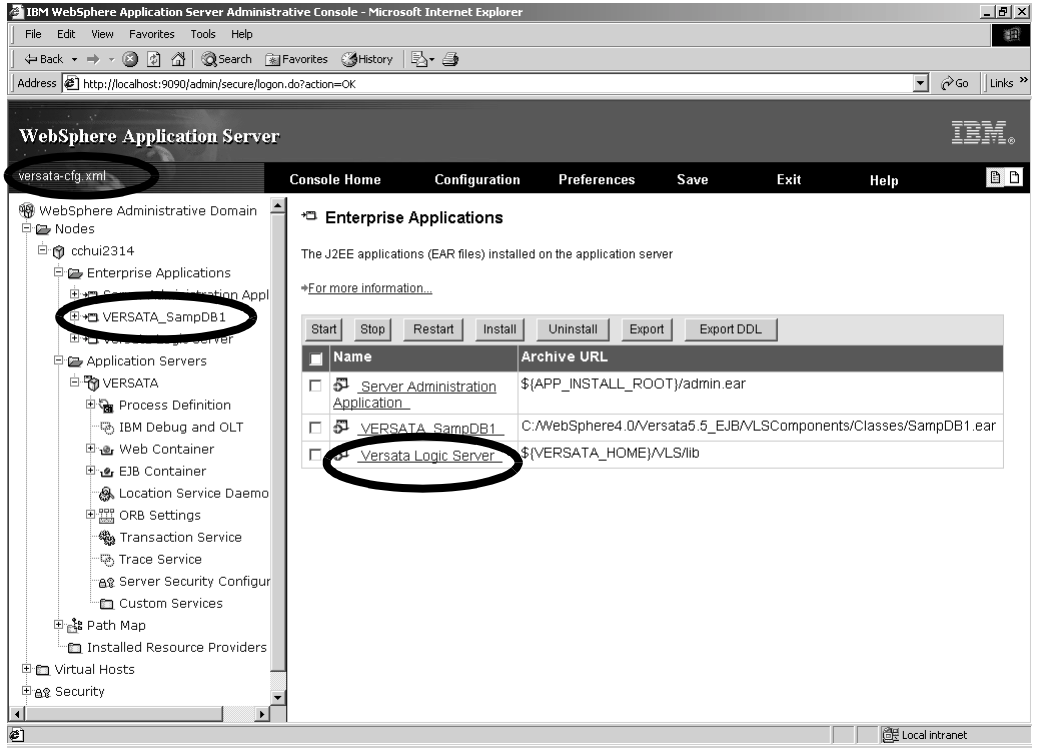

Figure 14 WAS 4.0 AES Console

Deployment wizard user interface

To start the Versata Logic Server Deployment wizard, do one of the following:

- Choose Versata Logic Server → Deploy Transaction Logic**.**
- Choose Managers → Deployment Manager, and in the Choose Deployment Target dialog, choose Deploy Transaction Logic under Versata Logic Server.

In the wizard dialogs, you can click the Help button or press F1 to obtain more information about the current task.

Deployment Options dialog

As the first step in the deployment process, a dialog appears where you can elect options for the rebuild and compile of business objects before they are deployed to the Versata Logic Server:

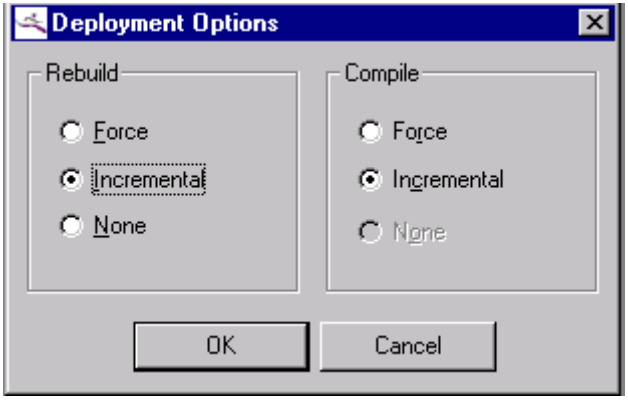

Figure 15 Deployment Options dialog

- Choose the Force option buttons to indicate that business objects should always be rebuilt and/or compiled before deployment.
- Choose the Incremental option buttons to indicate that only business objects that have changed since the last build and compile should be rebuilt and/or compiled before deployment.
- Choose the None option buttons to indicate that business objects should never be rebuilt or compiled before deployment.

After you have completed this dialog, objects are rebuilt and compiled, if indicated and necessary.

Choose Versata Logic Server for Deployment dialog

The next step in the deployment process is to choose the Versata Logic Server where business objects will be deployed, in the following dialog:

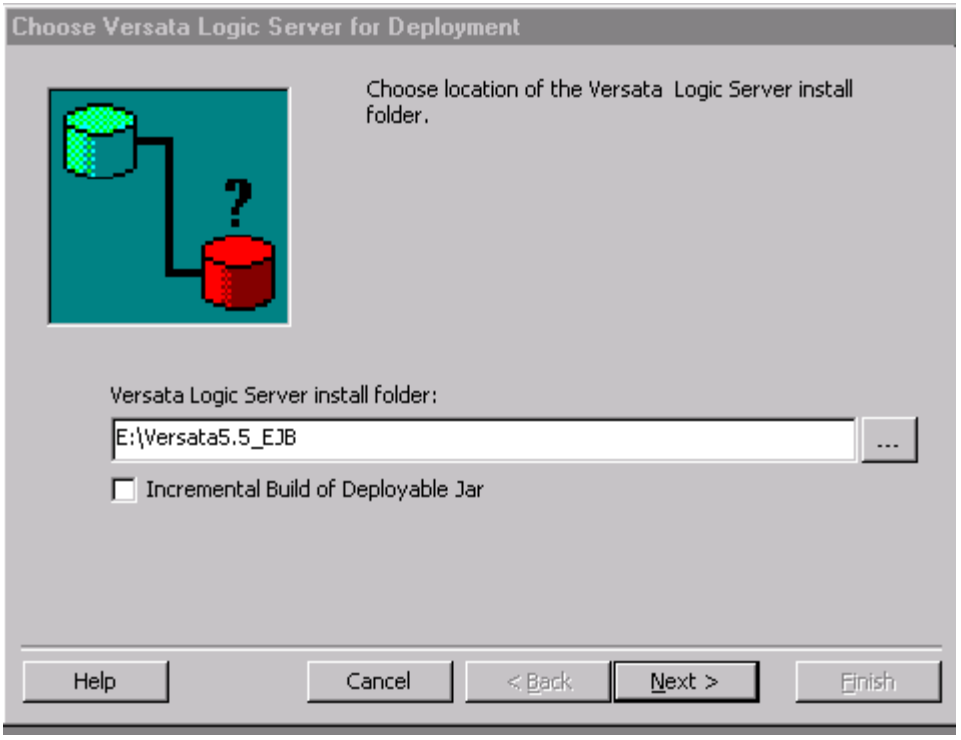

Figure 16 Choose Versata Logic Server for Deployment dialog

- In this dialog, you choose the Versata Logic Server where the J2EE Enterprise Application (EAR) containing your repository's business object files will be deployed. You indicate the Versata Logic Server by entering the path of the folder where the Versata Logic Server is installed. By default, this is your Versata Logic Studio installation directory.

To select from available folders, click the Browse button in the dialog that appears, browse until you find the right folder, then select it and click the Open button.Once you have entered the path of a folder, click the Next button.

Note that you must not use spaces in the path name, or it will not work.

- Also in this dialog, you have the option of choosing to deploy files only for changed objects, by enabling the Incremental Build of Deployable Jar check box. This option causes individual object files to be added to the existing repository `.jar` file instead of a complete regeneration of the `.jar`. This option allows you to preserve anything you have added to the `.jar` file as well as save time during deployment.

Finished dialog

In this dialog, you confirm the Versata Logic Server, choose options for deployment processing, specify a unique name for the Versata Logic Server application server process on IBM WebSphere Application Server, and then start the deployment processing.

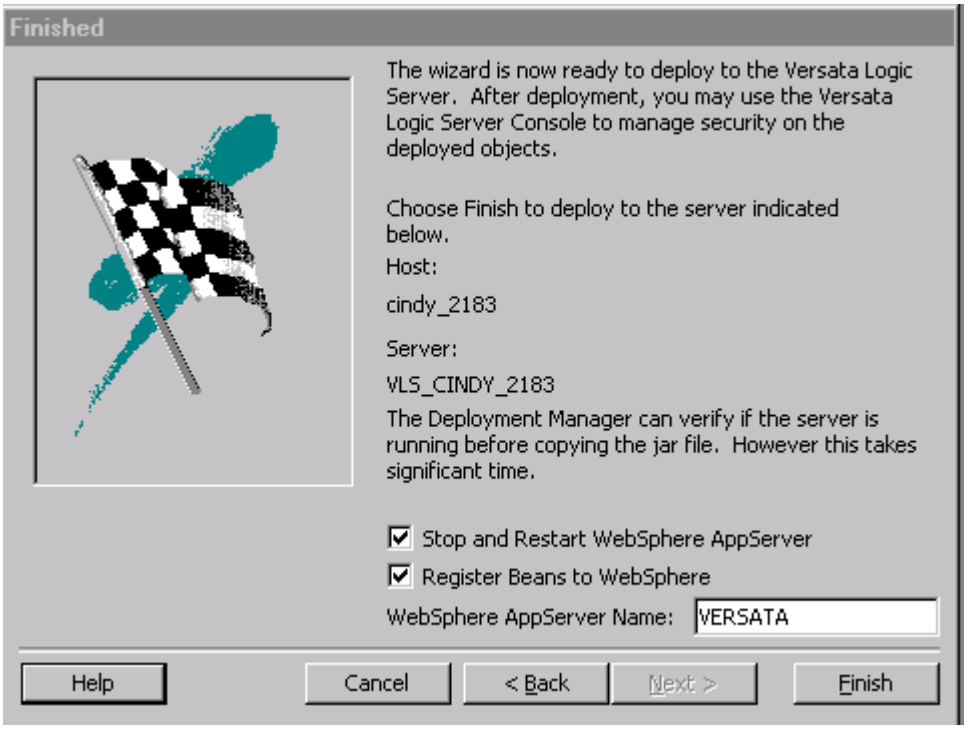

Figure 17 Deployment Finished dialog

- To confirm the Versata Logic Server, review entries in the Host and Server fields.
  - The Host field displays the location of the machine where the Versata Logic Server is installed.
  - The Server field displays the name of the Versata Logic Server specified in the configuration file you entered in an earlier dialog.

  These fields are read-only. You cannot modify them directly. If the host or server name is not correct, click the Back button to return to the Choose Versata Logic Server for Deployment dialog and enter a different install folder.

- The option for the wizard to stop and restart the Versata Logic Server on IBM WebSphere Application Server ensures that business object changes are applied. By default, this option is enabled. You need to start and stop the application server for changes to be applied, but in some cases, it may be faster for you to do it manually from the IBM WebSphere Administrative Console than for you to wait for the wizard to do it.
- The option for the wizard to register EJBs for business objects in IBM WebSphere Application Server uses WebSphere's XMLConfig tool for registration. By default, this option is enabled. Registration process information is displayed in a log file, so you can verify whether registration succeeded and use troubleshooting information in case of failure.
- The WebSphere AppServer Name field displays the name of the selected Versata Logic Server's application server process on the IBM WebSphere Application Server node where it is running. By default, this name is VERSATA. If you intend to run multiple instances of Versata Logic Server on the WebSphere node, change this name to something unique.

If you have deployed previously, a confirmation dialog may appear asking you whether existing files should be overwritten. When deployment is completed, a notification dialog appears.

Note: The Versata Logic Server verifies whether you have access to the file system. If you have problems deploying to the Versata Logic Server, check your permissions.

Deploying business objects to a development environment Versata Logic Server

To deploy business objects to a development environment Versata Logic Server:

1. Start the Deployment Manager and select Deploy Transaction Logic under Versata Logic Server deployment.
2. In the Deployment Options dialog, indicate the cases in which business object files should be rebuilt and recompiled before deployment:
   - Force indicates that files for all objects should always be rebuilt and/or recompiled before deployment.
   - Incremental indicates that only files for new or changed objects should be built and/or compiled before deployment.
   - None indicates that no files should be built and or compiled before deployment.
3. In the Choose Versata Logic Server for Deployment dialog, enter the path of the folder where the development Versata Logic Server is installed. By default, this is your Versata Logic Studio installation directory path. Note that you must not use spaces in the path name, or it will not work.

If you have previously deployed repository business objects to this Versata Logic Server, and you want to redeploy only changed business objects without overwriting the whole repository `.jar` file, enable the Incremental Build for Deployable Jar check box. Click Next.

4. In the Finished dialog, review the names of the host and of the Versata Logic Server. Make a note of these names, because you may need to record them as the server location in the Application Properties dialog for Versata Logic Studio-generated applications using this Versata Logic Server. Keep in mind that these names are case-sensitive.
5. Enable or disable the check box for the wizard to stop and restart the Versata Logic Server on IBM WebSphere Application Server. You can restart in order to ensure the objects are registered already.
6. Enable or disable the check box for the wizard to register business object EJBs in IBM WebSphere Application Server. These Beans must be registered.
7. If you want to run multiple instances of Versata Logic Server on the same IBM WebSphere Application Server node, change the WebSphere AppServer Name field from VERSATA to a unique application server process name.
8. Click Finish to execute the deployment.

   If you encounter any problems with deployment, check the deployment log file for troubleshooting information. This file is located in the `<install_directory>\Logs\vlsdeploy.log` folder.

Note: Customizations to the deployment descriptor (`<object>DD.xml`) file, such as a change to the jndi-name, may not be picked up during EJB deployment. To work around this issue, make the necessary changes in the WebSphere Administrative Console after deployment.

Note: Be careful to deploy only to a release 5.5 Versata Logic Server. The wizard allows you to deploy to a 5.1 Versata Logic Server.

Hot deploy and dynamic reloading

WebSphere 4.0 AES allows you to hot deploy and dynamically reload application components. The Versata Logic Studio deployment wizards support both of these features. The following sections provide information on hot deploying and dynamic reloading using the Versata Logic Server deployment wizard to deploy to Versata's development environment (the Versata Logic Server on WAS 4.0 AES, running on Windows).

For a summary of deployment scenarios involving hot deployment and dynamic reloading, see "Hot deploy and dynamic reloading task reference" on page 276.

Note: The hot deployment and dynamic reloading features are not applicable for deployment to Versata's production environment (the Versata Logic Server on WAS 4.0 AE, running on Windows, AIX, or Solaris).

Hot deploying to Versata's development environment

Hot deployment is the process of adding new components, such as enterprise beans, servlets, and JSP files to a running application without having to stop the VERSATA application server instance, and then restart it again. Use Versata Logic Studio deployment wizards to hot deploy application components to Versata's development environment.

- For business object hot deployment:

  Business and query repository objects are packaged as `.class` files in a `<repository>.jar` file. When you deploy the `<repository>.jar` file for the first time, you must stop and restart the WebSphere Application Server. However, if you change any of the `.class` files within the `<repository>.jar` file that was previously deployed, you can then hot deploy the updated `<repository>.jar` file. The Versata Logic Studio deployment wizards will restart only the VERSATA application server instance and the VERSATA_<repository> enterprise application, rather than the entire WebSphere Application Server. This save significant time during deployment.

- For client application hot deployment:

  With Versata-generated HTML applications, you can hot deploy client application components (such as `<servlet>.class` files and `<page>.class` files). If you do not change any of your application `.class` files, the Versata Logic Studio deployment wizards will copy all of the other changed files without re-starting the VERSATA application server instance, allowing for even faster deployment.

  In addition to hot deploying updated applications to Versata's development environment, you can also hot deploy new applications that have not yet been deployed—without having to restart the VERSATA application server instance.

Dynamic reloading in Versata's development environment

Dynamic reloading allows you to change existing application components without having to restart the WebSphere Application Server or the VERSATA application server instance in order for the changes to take effect. Such changes can include:

- Changes to the settings of an application, such as changing the deployment descriptor for a Web module.
- Changes to the implementation of a servlet.

Use the Versata Logic Studio deployment wizards to dynamically reload application components in Versata's development environment.

Hot deploy and dynamic reloading task reference

The following table summarizes various tasks you can perform using the Versata Logic Studio that will update Versata application components. The table describes the action you must take to make the changes effective in the running VERSATA application server instance.

For additional information, refer to IBM's documentation on WebSphere 4.0 hot deployment and dynamic reloading. In particular, refer to IBM's documentation on changes to application components not listed in the table below, and how changes to these components affect the unit that needs restarting (for example the module, application, or the application server instance).

Task	WebSphere action	Versata Studio action	Notes
Initial deployment of a Versata repository	Restarts	The Versata Logic Studio deployment wizard detects this scenario, and if the Restart WebSphere Applications check box is checked, the wizard will restart the VERSATA application server instance.	Deploying business objects or an HTML application for the first time for each repository requires restarting the WebSphere Application Server. This is so the `versata-cfg.xml` file can recognize future changes to a repository.

Task	WebSphere action	Versata Studio action	Notes
Changes to a servlet such as: • Adding a new servlet, including a new definition of the servlet in the `web.xml` deployment descriptor for the application. • Changing the `.class` file of an existing servlet by either editing or recompiling it.	None	Users must uncheck the Restart WebSphere Applications check box in the Versata Logic Studio deployment wizard.	
Changes to an HTML page	None	Users must uncheck the Restart WebSphere Applications check box in the Versata Logic Studio deployment wizard.	The Versata template (`.htm` file) is not loaded by WebSphere. The HTML page is read by the corresponding Java `<page>.class` file every time the HTML page changes.
Changes to the `.java` file for a page. Such a change can include: • Updating the implementation class for an EJB. • Updating a dependent class of the implementation class for an EJB.	Restarts the WebSphere Application Server, the Versata Logic Server, and the VERSATA_ <repository> enterprise application.	Users must check the Restart WebSphere Applications check box in the Versata Logic Studio deployment wizard.	The page class is loaded by the VLS runtime supporting the PLSContext bean. These classes can be thought of as dependent class of the PLSContext bean. However, to enable dynamic reloading of these dependent classes, these classes are loaded by the classloader instance that loads the bean itself.

Task	WebSphere action	Versata Studio action	Notes
Changes to business objects such as: • Updating the implementation class for an EJB. • Updating a dependent class of the implementation class for an EJB.	Restarts the WebSphere Application Server, the Versata Logic Server, and the VERSATA_<repository > enterprise application.	Users must select the Restart WebSphere Applications check box in the Versata Logic Studio deployment wizard.	Business object classes are loaded by the Versata Logic Server runtime supporting the VLSContext bean. These classes can be thought of as dependent classes of the VLSContext bean. However, to enable dynamic reloading of these dependent classes, these are loaded by the classloader instance that loads the bean itself.

Testing transaction logic

After defining transaction logic (business rules) and deploying business objects to the Versata Logic Server, you can verify that the rules execute as you expected. The Versata Logic Server Console provides a rule tracing facility that you can use to review details of rules processing. In addition, you can use a third-party debugger to step through business object code.

Using Versata Logic Server Console rule tracing

For further information about Versata Logic Server Console functionality, see the *Administrator Guide*.

To test business rules with the rule tracing facility:

1. Start the Versata Logic Server Console. Ensure that the guest user is set up with proper role and privileges.
2. Execute an application, logging on as guest.
3. In the Versata Logic Server Console, expand the User Sessions object and select the guest object. (This object represents your current session on the application.)
4. Enable the Trace user activity check box, and leave the Versata Logic Server Console open.
5. Modify records in ways that should cause the business rules to fire.
6. Review the entries in the Versata Logic Server Console trace window. Note that a number of entries are written as the transaction is processed, and that most recent entries are at the top. Scroll down through the trace window so you can see the first entries at the bottom, then review the entries from the bottom up.
7. When an error is raised, click More Details on the error dialog to find out which data object and rule raised the error.
8. If the error is related to business rules, return to the Transaction Logic Designer and review the data object's rule definitions.
   - As needed, update your derivation, validation, or referential integrity rules.
   - You also may need to update the data object constraints and action rule definitions.
9. After you have updated business rules, you can redeploy them to the Versata Logic Server and repeat the steps in this procedure to retest business rules.

Debugging business object code

The Versata Logic Suite allows you to use a third-party debugger designed specifically for debugging Java and HTML client applications as well as business objects deployed in a supported application server platform.

This edition supports the use of IBM's Object Level Trace (OLT) and Distributed Debugger. For procedures explaining how to use IBM's Distributed Debugger with Versata applications and business objects, see the *Application Developer Guide*. To learn more about using the tools featured in IBM's Distributed Debugger, see the *IBM Distributed Debugger User's Guide*.

Deploying business objects to a production environment

Once you have tested business objects in the development environment, you can deploy them to a production environment Versata Logic Server running on IBM WebSphere Application Server 4.0 Advanced Edition. Before you deploy business objects to the Versata production environment, you must create a `<repository>_Deployed.ear` file. The `.ear` file contains all the business object `.class` files and HTML application files in the repository. Web server files and security data are not contained in the `.ear` file.

To create the `.ear` file, use either the WebSphere Application Assembly Tool, or use the Versata `wsEARCreate.bat` script. Directions are provided for both methods.

Creating the .ear file

To create the <repository>_Deployed.ear file using the WebSphere Application Assembly Tool:

1. Run the WebSphere Application Assembly Tool.
2. Open the `<versata_logic_suite>\VLSComponents\Classes\<repository>.ear` folder.

   Do not add any files inside the folder after you opened it in the WebSphere Application Assembly Tool. The files will not be included in the output `.ear` file.
3. From the WebSphere Application Assembly Tool main menu, choose File → Save As to open the Save dialog.
4. Locate the `<versata_logic_suite>\VLSComponents\Classes` folder in the list box.
5. Type `<repository>_Deployed.ear` in the File name text field.
6. Click the Save button.

   The WebSphere Application Assembly Tool creates the output `.ear` file.

To create the <repository>_Deployed.ear file using the wsEARCreate. bat script:

1. Open a DOS console window.
2. Go to the `<versata_logic_suite>\VLS\bin` folder.
3. Type `wsEARCreate.bat -repository <repository> [-earfile <output ear file>]`

   The `-repository <repository>` is mandatory, however the `[-earfile <output ear file>]` is optional. If you do not specify the `-earfile`, the `wsEARCreate.bat` script will create the `.ear` file in the `VLSComponents\Classes` folder.

The two commands listed below create `SampDB1_Deployed.ear` in `VLSComponents\Classes` folder.

- `wsEARCreate.bat -repository SampDB1 -earfile C:\Versata\Suite-5.5-WebSphere\VLSComponents\Classes\SampDB1_Deployed.ear`
- `wsEARCreate.bat -repository SampDB1`

Deploying the .ear file

Once you have created the `<repository>_Deployed.ear` file, you can then deploy it to the Versata production environment. You must also copy your application's Web files and security files over to the Versata production environment. Use the following steps to complete the deployment process. For UNIX deployment, use the steps below, replacing \ with / and .bat with .sh.

To deploy the .ear file and copy the application Web and security files:

1. Copy your `.ear` file over to the production platform into the `<versata_logic_server>\VLSComponents\Classes` folder.
2. Copy the folder `<document_root>\<repository>` over to the `<document_root>\<repository>` in the production platform.
3. Copy all the `.dat` files in the `<versata_logic_suite>\VLSComponents\Admin` folder over to `<versata_logic_server>\VLSComponents\Admin` folder in the production platform.
4. Go to `<versata_logic_server>\Vls\bin` in a DOS console window.
5. Type `wsEARDeploy.bat -repository <repository>` to deploy the copied `.ear` file to WebSphere 4.0 Advanced Edition.

   The `wsEARDeploy.bat` file is located in the `<versata_logic_server>\VLS\bin` folder. It's usage is as follows:

   - `wsEARDeploy.bat -repository <repository name> [-earfile <output ear file>] [-logfile <output log file>]`

     The `-repository <repository name>` is mandatory, however the `[-earfile <output ear file>]` and the `[-logfile <output log file>]` are optional. If you do not specify the `-earfile`, the `wsEARDeploy.bat` script will locate the `<versata_logic_server>\VLSComponents\Classes\<repository>_Deployed.ear` file and deploy it to the `<versata_logic_server>\VLSComponents\Classes\<repository>.ear` folder. If you do not specify the `-logfile`, the `wsEARDeploy.bat` script will put its progress data in `<versata_logic_server>\Logs\eardeploy.log` file.

The two commands listed below deploy `SampDB1_Deploy.ear` in `VLSComponents\Classes` folder.

- `wsEARDeploy.bat -repository SampDB1 -earfile C:\Versata\Suite-5.5-WebSphere\VLSComponents\Classes\SampDB1_Deployed.ear`
- `wsEARDeploy.bat -repository SampDB1`

Setting default deployment values

Use the `defdeploy.properties` file, located in the `VLS\bin\` directory to set the default deployment values for the `wsVLSDeploy.bat` and `wsHTMLDeploy.bat` scripts. Setting the default values for each property eliminates the need for repeatedly typing the values when running the `wsVLSDeploy.bat` and `wsHTMLDeploy.bat` scripts.

The following example shows the new default values that have been set for the `VLSFolder`, `Host`, and `Node` properties:

```
VLSFolder=D:\Versata\VLS-5.5-WebSphere
Host=cchui2240
Node=cchui2240
Port=900
AppServer=*
WebApp=*
```

Note: The * symbol used for the AppServer and WebApp properties tells the `wsVLSDeploy.bat` file to use the values from the `vlsdeploy.properties` file and the `wsHTMLDeploy.bat` file to use the values from `appdeploy.properties` file.

CHAPTER 9

Understanding Business Object Files

Chapter overview

Read this chapter to gain an understanding of the business objects generated and deployed by the Versata Logic Studio to the Versata Logic Server. This chapter discusses the files generated to process run-time business objects and some of the code contained in these files. After reading this chapter, you should have a clear understanding of the names and contents of generated business object files, and how to view them in the Code Editor.

This chapter includes the following:

- "Overview of Versata Logic Server business objects" on page 287, outlines business object definition and deployment, and introduces the basic architecture of business objects.
- "Generated files for business objects" on page 290, lists the different types of generated business object files, explaining naming conventions and providing details about the code contained in each type of file.
- "Reviewing file properties" on page 306, explains how to use the File Properties dialog to view information about generated and external files in the repository.
- "Working with external files" on page 308, describes how to make external files available in a Versata repository.
- "Using a code editor" on page 313, describes how to view and add custom code to generated business object files in the Versata Code Editor and how to use an external code editor for these purposes.

Overview of Versata Logic Server business objects

Versata Logic Server hosts the business objects generated by the Versata Logic Studio. You can define two types of business objects in the Versata Logic Studio: data objects and query objects.

- **Data objects.** Process the logic defined in business rules and perform data retrieval and processing. For information about creating and modifying data objects, see "Working with Data Objects" on page 81.
- **Query objects.** Select attributes from one or more data objects to create a composite object; they are similar to RDBMS views. By default, query objects provide only client behavior (retrieval and modification) and delegate transaction logic enforcement to the data objects on which they are based. For information about creating and modifying query objects, see "Working with Query Objects" on page 145.

Business object deployment

After you have defined data objects and query objects, you can build and compile these business objects in the Versata Logic Studio in order to generate 100% Java objects. These objects contain code to implement the business rules and other object definition information. You have the choice of deploying business objects as Enterprise JavaBeans (EJBs) or simply as Java class files. You should deploy an object as an EJB when you want to make it remotely accessible. You mark a data object for deployment as an EJB by enabling a check box in the Transaction Logic Designer. You mark a query object for deployment as an EJB by enabling a check box in the Query Object Designer.

Deployment of the generated business objects places them in a location accessible to the Versata Logic Server and to IBM WebSphere Application Server, where they can execute transaction logic for applications. These business objects provide the middle tier that links client applications to data sources. For information about building and deploying business objects, see "Building and Deploying Business Objects" on page 255.

If you deploy a data object as an EJB, it as packaged as an entity bean. Each entity bean encapsulates permanent data, which is stored in a data source such as a database or a file system, and associated methods to manipulate that data. In most cases, an entity bean must be accessed in some transactional manner. Instances of an entity bean are unique and they can be accessed by multiple users. Entity beans for Versata Logic Server data objects implement bean-managed persistence (BMP). Each EJB handles its own synchronization with its data source. BMP allows Versata Logic Server EJBs to run against a wide variety of data sources, both those for which IBM WebSphere Application Server provides data access and data sources that the Versata Logic Server can access through a system-supplied or custom Versata Connector.

If you deploy a query object as an EJB, it is packaged as a session bean. Each session bean encapsulates nonpermanent data associated with a particular EJB client. Unlike the data in an entity bean, the data in a session bean is not stored in a permanent data source, and no harm is caused if this data is lost. However, a session bean can update data in an underlying database, usually by accessing an entity bean. A session bean can also participate in a transaction. When created, instances of a session bean are identical. A session bean is always associated with a single client; attempts to make concurrent calls result in an exception being thrown.

Business object basic architecture

The business objects inherit business-logic processing behavior from a common framework that the Versata Logic Server provides. Any interactive GUI application (such as a Java stand-alone application, Java applet, HTML client, or JSP client running in a browser) can interact with Versata Logic Server business objects. Applications developed outside of the Versata Logic Studio can interact through standard API calls.

Versata Logic Server business objects also leverage the capabilities of the IBM WebSphere Application Server, particularly if they are implemented as EJBs. These capabilities include transaction management; business objects are fully compliant with the JTA standard using the JTA API. Objects that are implemented as EJBs provide remote access and can be reached by any EJB client using pure EJB APIs.

The Versata Logic Server itself manages state, transaction, and session information. All common logic-processing behavior is abstracted in the Versata Logic Server automation framework so that it is reusable across user components. This abstraction ensures that the user components are as thin as possible.

The generated business object code includes a rich set of common APIs that provide intra-object and inter-object access. So, when logic that is executing in one object needs to access another object, the first object can call a published API that finds the related object based on the state of the calling object. For example, logic executing in a customer's business object may need to find related orders objects.

The business objects also provide APIs to access their state information. For example, suppose a customer's object has a state in which properties such as name and address are defined. When these properties' values are changed for a customer, a new customer is created, or an existing customer is deleted, an API call is issued to make the change permanent and the system guarantees that the appropriate business rules are processed before the change can be considered successful. This guarantee includes recursive processing of the rules across multiple objects. For example, deleting a customer not only requires processing rules on the customer but also on related orders, including deleting the customer's orders.

Because the business objects are 100% Java, their behavior is extensible at several levels. For example, the objects can provide event-handling behavior in a particular situation or create subclasses that inherit and extend behavior from their superclasses.

Versata Connectors provide data access via pluggable modules. After logic processing is completed, the Transaction Logic Engine passes the objects to appropriate Versata Connectors to make the changes permanent in the data source. Using the Connectors instead of directly accessing the data during transactions ensures that the data storage mechanism is completely independent of the logic processing.

Generated files for business objects

The Versata Logic Studio generates the following files for each business object you build and compile:

- Base implementation file, `<object_name>BaseImpl.java`, is the base class for the object, that contains all system- generated code, including rules.
- Main implementation file, `<object_name>Impl.java`, extends the base class and serves as the container for any developer-defined, or custom, code.
- Internal implementation file, `<object>Impl.built`, is not listed in the Versata Logic Studio Explorer and not described in detail here. These files are stored in cache and used by the system to optimize when to rebuild and revalidate objects. You should not need to edit these files or maintain them in source control, as they are used for internal purposes only during design time and do not get deployed for run-time use.

If you indicate a business object should be deployed as an Enterprise JavaBean (EJB), the following additional files also are created for the object:

- Remote interface file, `<object_name>.java`
- Home interface file, `<object_name>Home.java`
- Deployment descriptor file, `<object_name>DD.xml`

When you deploy business objects, all of these files are added to a repository `.jar` file, which is packaged in a J2EE application that can be copied to a location accessible to the Versata Logic Server and to IBM WebSphere Application Server.

Note: In addition to the files described here, each business object also has an `.xml` file which contains metadata for that object. These `.xml` files are not listed in the Versata Logic Studio Explorer, as they are in a different category from the generated files that execute for run-time applications. For information about the `.xml` metadata files, see the *Project Guide*.

You can open these files and review their contents in the Versata Logic Studio, from either the Objects tab or the Files tab of the Versata Logic Studio Explorer. The Files tab of the Explorer lists files generated for repository objects as well as files that developers have added to the repository. Each file has an icon next to it. File icons with black text indicate the file is generated by the Versata Logic Studio. Icons with red text indicate the file is developer-defined. An icon with an included satellite icon indicates the object has been configured for remote access.

You can right-click a file and choose Properties to review information about it, including its full path location. You can open files as follows:

- To open a base implementation file, on the Objects tab of the Explorer, right-click the object and choose Open File → Base Implementation File. You cannot modify the contents of this file, but you can copy code from it.

- To open an implementation file, on the Objects tab of the Explorer, right-click the object and choose Open File → Implementation File. Or, on the Files tab of the Explorer, double-click the file.
- To open a remote interface, home interface, or deployment descriptor file, on the Files tab of the Explorer, double-click the file.

The selected file opens in the Code Editor. For information about modifying files in the Code Editor, see "Methods for instantiating business objects" on page 326.

Implementation files

For all deployed business objects, the Versata Logic Server uses the implementation files to instantiate objects for logic processing and saves to the data source. Versata Logic Server provides "Just-In-Time" (J.I.T.) business object instantiation, meaning objects are created only when necessary, for example when updates must be saved. The implementation files also contain logic execution code, which enforces rules during updates of object data.

The Versata Logic Studio generates by default most of the blocks of code for the business object implementation files, except for event blocks in the implementation file and the business object factory methods block in the base implementation file. The event blocks are where you can add your custom event-handling code. Use the Code Editor to add custom code to a particular event for a business object. If you enable remote access for a business object, the Versata Logic Studio generates code for this purpose in the business object factory methods block of the base implementation file.

Data object implementation files

The base implementation file for each data object (`<data_object>BaseImpl.java`) contains generated transaction logic. The implementation file for each data object (`<data_object>Impl.java`) contains custom server event handlers and custom server methods (both new and overloaded).

By default, each data object base implementation file extends `versata.vls.DataObject`. Each data object implementation file extends the base implementation file. To provide additional methods to a data object, you can create a subclass of the `versata.vls.DataObject` class to contain these methods, then specify this subclass as the superclass of the data object base class. You enter this property on the Properties:Data Access tab of the Transaction Logic Designer. For more information about creating data objects with additional methods, see "Creating a DataObject subclass with specialized methods" on page 339.

Blocks of code included in data object base implementation files and implementation files are described below.

Component import blocks

Each data object base implementation and implementation file begins with imports of JDK and Versata packages, as illustrated in the following examples:

```
//{{COMPONENT_BASE_IMPORT_STMTS
package SampDB1;
import java.util.Enumeration;
import java.util.Vector;
import versata.common.*;
import versata.vls.*;
import java.util.*;
import java.math.*;
//END_COMPONENT_BASE_IMPORT_STMTS}}
```

```
//{{COMPONENT_IMPORT_STMTS
package SampDB1;
import java.util.Enumeration;
import java.util.Vector;
import versata.common.*;
import versata.vls.*;
import java.util.*;
import java.math.*;
//END_COMPONENT_IMPORT_STMTS}}
```

Data object declarations and constructors

After the import statements, the base implementation file declares the data object base class as a subclass of `versata.vls.DataObject` or one of its subclasses. The implementation file declares the data object as a subclass of the base class. In each file, this declaration is followed by a constructor block that initializes the class, as shown in the following examples.

This example is from a base implementation file:

```
abstract public class CUSTOMERSBaseImpl extends DataObject{
   public CUSTOMERSBaseImpl () {
      super();
      addListeners();
}
```

This example is from an implementation file::

```
//{{COMPONENT_RULES_CLASS_DECL
public class CUSTOMERSImpl extends  CUSTOMERSBaseImpl
//END_COMPONENT_RULES_CLASS_DECL}}
{
       //{{COMP_CLASS_CTOR
       public CUSTOMERSImpl (){
              super();
       }

       public CUSTOMERSImpl(Session session, boolean makeDefaults)
       {
              super(session, makeDefaults);
       //END_COMP_CLASS_CTOR}}
   }
```

Data definition block

The data definition block in a data object's base implementation file defines the package name and metadata (data objects and attributes). Also, for each attribute, the data type is defined and a variable is set to indicate whether the attribute is updatable, nullable, derived, or incremented automatically, and a caption is defined. The following example illustrates the data definition block for a data object with only one attribute, in order to conserve space.

```
//{{COMPONENT_META_QRY
   private final static String deployedFromRepository =
"SampDB1";
   private static VSQueryDefinition  q= null;
   public String getPackageName() {
      return deployedFromRepository;
   }
   static {
      q= new VSQueryDefinition( "CUSTOMERS" );
      q.setPackageName(deployedFromRepository);
      //{{META_QUERY_COLUMN_CTOR
      c = new VSMetaColumn("CustNum", DataConst.BIGINT);
      c.setAutoIncrement(true);
      c.setAlterability(false);
      c.setNullability(VSMetaColumn.columnNoNulls);
       c.setCaption("Cust#");
      t.addColumn( c );
   t.addUniqueKeyColumn( "CUSTOMERS Unique Key", "CustNum" );

      t.useQuotedIdentifier(false);
      t.setOptLock( DataConst.OptLockingOnApplicable );

   //END_META_QUERY_COLUMN_CTOR}}
      q.addTable( t );
      q.populateQCArray();
      VSQueryDefinition qTemp = (VSQueryDefinition)
getMetaQuery( "CUSTOMERS",deployedFromRepository );
      if ( qTemp == null ) {
         addMetaQuery(q, deployedFromRepository);
      }
      else
         q = qTemp;// Keep the old query as it has cached object
//END_COMPONENT_META_QRY}}
```

Note: The implementation file for a data object does not contain a data definition block.

Rules blocks

The rules block is generally the largest block of code in the data object's base implementation file. This block consists of processing code for inserts, updates, and deletes. The code that enforces rules is built into this processing code. The rules enforcement code is based on the rules you defined in the Transaction Logic Designer. This block includes code to instantiate the object so that rules logic can be executed against it, including factory methods for data objects, and `get` and `set` methods for each object attribute, code to cascade updates to related objects' records as necessary, code that adds default listeners for factory events, and code that creates a default Versata Connector for the data object. This block also includes a `RecomputeDerivations()` method, that executes rules logic against preexisting records. See "Recomputing derivations" on page 354 for more information. For more information about factory methods, see "Factory methods" on page 326.

Because this block is lengthy and has diverse contents, an example is not provided here. To view examples of this code, build and compile business objects in the sample repository, then open a data object base implementation file in the Code Editor and search for `//{{COMPONENT_RULES`.

The rules block in a data object's implementation file simply gets the object, as shown in the following example:

```
//{{COMPONENT_RULES
     public static VSMetaQuery getMetaQuery() {
         //return CUSTOMERSBaseImpl.getMetaQuery();
        return getMetaQuery("CUSTOMERS", "SampDB1");
     }
     public static CUSTOMERSImpl getNewObject(Session
session, boolean makeDefaults)
     {
        return new CUSTOMERSImpl(session, makeDefaults);
     }

   //END_COMPONENT_RULES}}
```

Data object event blocks

Each data object implementation file provides event blocks where you can add your own custom code that is executed at the specified event time. The Versata Logic Studio preserves the code in these blocks whenever objects are rebuilt and recompiled. In addition to the event handlers that you write, the Versata Logic Studio generates listener code to register an object as a receiver of the event callback.

Exposed events for data objects include: `afterCommit`, `afterDelete`, `afterInsert`, `afterQuery`, `afterRollback`, `afterUpdate`, `beforeCommit`, `beforeDelete`, `beforeInsert`, `beforeQuery`, `beforeResultSetFill`, `beforeRollback`, and `beforeUpdate`.

For more information about the code in event blocks, see “Methods for instantiating business objects” on page 326.

The following event code example from the sample repository writes a record to the EMPLOYEESAUDIT data object whenever a salary change occurs in the EMPLOYEES data object:

```
public void afterUpdate(DataObject obj)
{
long empId = ((EMPLOYEESImpl) obj).getEmpID();  // get values from current row
BigDecimal newSalary = getSalary(); // running in Employees Object,
BigDecimal oldSalary = getOldSalary();          // so this. is implicit

EMPLOYEESAUDITImpl empSalHist =  EMPLOYEESAUDITImpl.getNewObject(getSession(),
true);
   empSalHist.setOldSalary(oldSalary);
   empSalHist.setNewSalary(newSalary);
   empSalHist.setEmployeeID(empId);
   empSalHist.save();     }

   // note that this server code uses objects, not rows.  As a general rule:
   //    - client code: use rows (these consume fewer server resources)
   //          -- except when you need to invoke Remote Methods
   //    - server code: use objects (since you are already in the server)

}
```

Note: Data object base implementation files do not contain event blocks.

Other data object code blocks

Both the base implementation files and implementation files for data objects include some additional blocks of code.

Each base implementation file contains an abstract custom methods block. This section includes abstract prototypes for any custom methods added to the implementation file. The following excerpt from the ORDERSBaseImpl file in the sample repository provides an example:

```
//{{ABSTRACT_CUSTOM_METHODS
   //Print abstract prototype for custom methods.
   abstract public void beforeCommit( Session session,Response
response );
   abstract public void sendBigOrderMail( long SalesRepID,long
CustNum,long OrderNumber ) throws ServerException;
   abstract public void purge( );

   //END_ABSTRACT_CUSTOM_METHODS}}
```

Each implementation file contains an event add listeners block and a factory methods block. The Versata Logic Studio generates code for the event add listeners block in files containing custom event code. You can add your own custom factory methods to the factory methods block. For information about factory methods, see “Factory methods” on page 326. You should add any other custom methods after the factory methods block and before the end of the file.

Methods inherited from the superclass

Generated data objects share many common methods inherited from `versata.vls.DataObject`, which is the default superclass for data object base classes. You should be aware of these methods, as they are used throughout data object implementation files. These methods include the following:

- `isChanged(AttributeName)` method indicates whether the value of an attribute has been modified in the current transaction.
- `raiseException` method throws an exception in the server.
- Methods to obtain context information, including current session, Versata Connector in use, current user, and current date. Methods include `getSession`, `getXDAConnector`, `getUser`, and `getDate`. A `setSession` method also is provided.
- Methods to demarcate transaction boundaries, including `beginTransaction`, `commitTransaction`, and `abortTransaction`.
- `IsNull` and `IsOldNull` methods that indicate whether the value of an attribute is null.

Query object implementation files

The base implementation file for each query object (`<query_object>BaseImpl.java`)contains generated transaction logic. The implementation file for each query object (`<query_object>Impl.java`) contains custom server event handlers and custom server methods (both new and overloaded).

By default each query object base implementation file extends `versata.vls.QueryObject`. Each query object implementation file extends the base implementation file. To provide additional methods to a query object, you can create a subclass of the `versata.vls.QueryObject` class to contain these methods, then specify this subclass as the superclass of the query object base class. You enter this property on the Properties:General tab of the Query Object Designer. For more information, see “Properties tab” on page 171.

Blocks of code included in query object base implementation files and implementation files are described below.

Component import blocks

Each query object base implementation and implementation file begins with imports of JDK and Versata Logic Suite packages, as illustrated in the following examples:

```
//{{COMPONENT_BASE_IMPORT_STMTS
package SampDB1;
import java.util.Enumeration;
import java.util.Vector;
import versata.common.*;
import versata.vls.*;
import java.util.*;
import java.math.*;
//END_COMPONENT_BASE_IMPORT_STMTS}}
```

```
//{{COMPONENT_IMPORT_STMTS
package SampDB1;
import java.util.Enumeration;
import java.util.Vector;
import versata.common.*;
import versata.vls.*;
import java.util.*;
import java.math.*;
//END_COMPONENT_IMPORT_STMTS}}
```

Query object declarations

After the import statements, the base implementation file declares the query object base class as a subclass of `versata.vls.QueryObject` or one of its subclasses. The implementation file declares the query object as a subclass of the base class.

The following example is from a base implementation file.

```
abstract public class OrderItemJoinPartBaseImpl extends
QueryObject
```

The following example is from an implementation file.

```
//{{COMPONENT_RULES_CLASS_DECL
public class OrderItemJoinPartImpl extends
OrderItemJoinPartBaseImpl
//END_COMPONENT_RULES_CLASS_DECL}}
```

Query object constructors

The constructor block in each query object base implementation file includes code to populate metadata for the query object, to define SQL text for the query object, and to create a Versata Connector for the query object, as well as factory method code. Some example excerpts of this code appear on the following pages.

The constructor block in each query object implementation file simply gets the object, as shown in the following example:

```
{
   //{{COMPOSITE_COMPONENT_METHODS
      public static VSMetaQuery getMetaQuery() {
           return getMetaQuery("OrderItemJoinPart", "SampDB1");
       }

   //END_COMPOSITE_COMPONENT_METHODS}}
```

The following code from the OrderItemJoinPart base implementation file populates metadata for query object attributes:

```
{
//{{BASE_COMPOSITE_COMPONENT_METHODS
// Constructors

static {
   VSQueryDefinition q = new VSQueryDefinition(
"OrderItemJoinPart" );
   q.setPackageName("SampDB1");
   // Construct a query column definition.
   // Syntax is:
   //new VSQueryColumnDefinition( "<tableName>", "<tableAlias>",
"<col
   umnName>", "<columnAlias>" );
   // Alternate syntax is:
   //   add( "<tableName>", "<tableAlias>","<columnName>",
   "<columnAlias>" );

   //{{QRYDEF_COLUMN_CTOR
   q.add ("PART", "PART", "Name", "Name");
   q.add ("ORDERITEM", "ORDERITEM", "QtyOrdered", "QtyOrdered");
   q.add ("ORDERITEM", "ORDERITEM", "PartPrice", "PartPrice");
   q.add ("ORDERITEM", "ORDERITEM", "Amount", "Amount");
   q.add ("PART", "PART", "ImageName", "ImageName");
   q.add ("ORDERITEM", "ORDERITEM", "PartAutoBucks",
"PartAutoBucks");
   q.add ("ORDERITEM", "ORDERITEM", "AutoBucksEarned", "AutoBuck
   sEarned");
   q.add ("ORDERITEM", "ORDERITEM", "PartNum", "PartNum");
   q.add ("ORDERITEM", "ORDERITEM", "ShippedFlag",
"ShippedFlag");
   q.add ("ORDERITEM", "ORDERITEM", "OrderNum", "OrderNum");
   q.add ("ORDERITEM", "ORDERITEM", "Notes", "Notes");
   q.add ("PART", "PART", "AutoBucks", "AutoBucks");
   q.add ("PART", "PART", "Make", "Make");
   q.add ("PART", "PART", "Model", "Model");
   q.add ("PART", "PART", "UnitOfSale", "UnitOfSale");

      //END_QRYDEF_COLUMN_CTOR}}
```

The following code from the OrderItemJoinPart base implementation file defines the SQL text and childmost data object for the query object:

```
q.setSQL(" SELECT PART.Name AS Name,  ORDERITEM.QtyOrdered AS QtyOrdered,
ORDERITEM.PartPrice AS PartPrice,  ORDERITEM.Amount AS Amount,
ORDERITEM.PartAutoBucks AS PartAutoBucks,  ORDERITEM.PartNum AS PartNum"  +
 ",  ORDERITEM.ShippedFlag AS ShippedFlag,  ORDERITEM.OrderNum AS OrderNum,
ORDERITEM.Notes AS Notes,  PART.AutoBucks AS AutoBucks,  PART.Make AS Make,
PART.Model AS Model,  PART.UnitOfSale AS UnitOfS"  +
 "ale FROM <dbschema>.PART PART,  <dbschema>.ORDERITEM ORDERITEM WHERE
PART.PartNum = ORDERITEM.PartNum");

q.setChildMostTableName("ORDERITEM");
```

Query object event blocks

Each query object implementation file provides event blocks where you can add your own custom code that is executed at the specified event time. Versata Logic Studio preserves the code in these blocks whenever objects are rebuilt and recompiled. In addition to the event handlers that you write, Versata Logic Studio generates listener code to register an object as a receiver of the event callback.

Exposed events for query objects include: `afterQuery`, `beforeQuery`, and `beforeResultSetFill`.

For information about writing server event code, see “Adding files to a repository” on page 308.

Note: Query object base implementation files do not contain event blocks.

The following example event code is from the CustQueryVirtuals implementation file in the sample repository:

```
//{{EVENT_CODE

//{{COMP_EVENT_beforeResultsetFill
public static void beforeResultsetFill(DataRow rowToBeAdded,
Response response)
{
   System.err.println("CustQueryVirtual: " +
   rowToBeAdded.getData("Name").getString() );

   Enumeration e = rowToBeAdded.getAllColumnValues();
   Data d;
   while ( e.hasMoreElements())
{
      d = (Data)e.nextElement();  // get Data Object
      System.err.println(" Data: " + d);                        }

   System.err.println("  CustQueryVirtual : " +
    rowToBeAdded.getData("AddressLine"));
   rowToBeAdded.getData("AddressLine").setString("lll");
   System.err.println("  CustQueryVirtual: " +
    rowToBeAdded.getData("Name").getString() );

   System.err.println("");

}
//END_COMP_EVENT_beforeResultsetFill}}

   //END_EVENT_CODE}}
```

Other query object code blocks

The implementation files for query objects contain an additional block of code to contain any custom factory methods that have been added to the object. You should add any other custom methods after this block and before the end of the file.

Remote and home interface files

The home and remote interface files, which are generated for each Versata Logic Server business object, are used only for objects deployed as EJBs. These files allow remote clients and objects to access business object methods.

Home interface file

The home interface file (`<object>Home.java`) defines methods called by remote clients or objects to create, find, and remove instances of the EJB. This interface is used to obtain a reference to an EJB's remote interface. It provides bean creation and is similar to a class factory in CORBA. Home interfaces for Versata Logic Server business objects extend `javax.ejb.EJBHome`. The following example from the sample repository illustrates code in a home interface file

```
package SampDB1;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;
import versata.common.*;

/*
** Home Interface ORDERSHome
*/

public interface ORDERSHome extends javax.ejb.EJBHome {
   public ORDERS findByPrimaryKey (PrimaryKey key) throws
RemoteException, FinderException;
   public Enumeration findObjects(SearchRequest sr) throws
RemoteException, FinderException;
   public ORDERS create(Properties values) throws
RemoteException, CreateException;
}
```

Remote interface file

The remote interface file (`<object>.java`) provides an interface that remote clients or objects can use to invoke the business object's methods after they have used the home interface to gain access to the EJB. This interface adds support for transactions, security, and threading. Remote interfaces for Versata Logic Server business objects extend `versata.common.BusinessObject`. Each remote interface file inherits all generic methods from `versata.common.BusinessObject` and may contain other methods that need to be remotely accessible, such as instance methods. One of the most important generic methods provided is a `save()` method that saves the business object on the Versata Logic Server to the data source and can throw a `VSORBException`.

The following example from the sample repository illustrates code in a remote interface file. This file does not include any remote methods. You need to add any methods you want to make available remotely by copying them from the base implementation file or implementation file.

```
import versata.vls.* ;
import java.util.* ;
import java.math.* ;
import java.rmi.* ;


/*
** interface ORDERS
*/

public interface ORDERS extends BusinessObject
{
   public void purge() throws java.rmi.RemoteException;
}
```

Deployment descriptor file

The deployment descriptor file (`<object>DD.xml`) contains attribute and environment settings that define how the EJB container invokes EJB functionality. The Versata Logic Studio generates a deployment descriptor that contains all required settings. The following example from the sample repository illustrates deployment descriptor settings for an entity bean. Note that session beans require a few additional attribute settings: a state management attribute, which defines the state of the session bean as either STATEFUL or STATELESS, and a time-out attribute which defines the bean's associated idle time-out value in seconds.

```
<entity-bean dname="SampDB1/ORDERS.ser">
<primary-key>versata.common.PrimaryKey</primary-key>
<re-entrant value="false"/>
<home-interface>SampDB1.ORDERSHome</home-interface>
<remote-interface>SampDB1.ORDERS</remote-interface>
<enterprise-bean>SampDB1.ORDERSImpl</enterprise-bean>
<jndi-name>SampDB1/ORDERS</jndi-name>

<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="READ_COMMITTED"/>
<run-as-mode value="SYSTEM_IDENTITY"/>

<method-control>
<method-name>init</method-name>
<parameter>class java.util.Properties</parameter>
<parameter>class java.util.Properties</parameter>
<transaction-attr value="TX_MANDATORY"/>
<isolation-level value="READ_COMMITTED"/>
<run-as-mode value="SYSTEM_IDENTITY"/>
</method-control>

</entity-bean>
```

Note: Customizations to the deployment descriptor (`<object>DD.xml`) file, such as a change to the jndi-name, may not be picked up during EJB deployment. To work around this issue, make the necessary changes in the WebSphere Administrative Console after deployment.

Reviewing file properties

The Versata Logic Studio Explorer provides a dialog where you can review the properties of repository business object files and user-defined files. This dialog, the File Properties dialog, is available from both the Objects tab and Files tab of the Explorer.

The File Properties dialog displays the following information about the selected file:

- Name
- MS-DOS name
- Full path location
- Type (when opened from Files tab only)

  Available types include: Generated Component File, Home Interface, Remote Interface, Deployment Descriptor, and External File. Each External File is defined as one of the following subtypes: Interface, XDA Connector, or Other. This subtype is determined from comment text added by Versata Logic Studio when you add the file to the repository, based on the menu options you select to add the file.
- Size (when opened from Files tab only)
- Date of creation
- Date of last modification
- Date of last access
- Whether it is a hidden, system, archive, or read-only file.

This dialog also allows you to modify the read-only attribute of the file.

Note: If a file's attributes are changed outside of Versata Logic Studio while its File Properties dialog is open, no notification is provided that the information in this dialog is no longer accurate.

Reviewing file properties from the Objects tab

To open the File Properties dialog from the Objects tab of the Explorer:

1. Right-click an object and choose File Properties dialog.

   The File Properties dialog opens, displaying a list of files associated with the object.

   - Each data object has an `Impl.java` file and an `.xml` file. It may have a `.csv` file that contains its data, one or more `.xml` files for any of its relationships with other data objects, and interface and deployment descriptor files if it is deployed as an EJB.
   - Each query object has an `Impl.java` file and an `.xml` file. It may have interface and deployment descriptor files if it is deployed as an EJB.
2. In the dialog, select a file from the list box.

Reviewing file properties from the Files tab

To open the File Properties dialog from the Files tab of the Explorer:

1. Right-click a file and choose File Properties dialog.

Modifying a file's read-only attribute

The File Properties dialog allows you to modify a repository file's read-only attribute.

To modify a file's read-only attribute:

1. Open the File Properties dialog. If you have opened it from the Objects tab of the Explorer, select a file in the list box.
2. Enable or disable the Read Only box.
   - Click Apply to confirm the change and leave the dialog open.
   - Click OK to confirm the change and close the dialog.
   - Click Cancel to dismiss the dialog without any changes.

Note: A file that is read-only appears with a lock icon in the Versata Logic Studio Explorer. After you make this file writable, the lock icon continues to display. To remove this icon, choose Versata Logic Server → Refresh.

Working with external files

Files must be included in the repository or the classpath in order to be available for use with Versata Logic Studio-generated applications. In certain circumstances, you may need to add files to the repository (for example, custom Versata Connectors and subclasses of `versata.vls.DataObject`). Generally, though, you need only add a file to the Versata classpath so that the file can be located. For information about how you can make a file available to Versata Logic Server business objects and applications without actually adding it to the repository, see "Adding files and packages to the classpath" on page 310.

Adding files to a repository

To make an existing file available to a repository, you can either add it as a reference or copy it into the repository. Referencing writes the location of the file into the repository metadata. Copying writes the location of the file into the repository metadata and copies the file into a standard subfolder for repository files. Copying a file makes it easier to manage as part of the repository, for example when doing backups or compiling. Referencing a file allows you to maintain the file in another location if you need it there for another purpose. In both cases, the file is listed in the Versata Logic Studio Explorer.

If you have not yet created a file and you want it to be available to a repository, you have the option of creating it in the Versata Logic Studio.

The Versata Logic Studio offers three menu options for making a file available to the repository. These menu options are available on the Files tab of the Explorer when you right-click the Versata Logic Server folder or one of its contained group folders:

- **New File/New XDA Connector.** Creates a new file in the repository and opens it in an editor for you to write code or text.
- **Add Files.** Adds a reference to the existing file(s) in the repository.
- **Add File Copies.** Copies the existing file(s) into the repository.

Note: If you add a file with a name containing numbers, you may encounter compilation errors.

Referencing an existing file in a repository (Add Files)

To reference a file in a repository:

1. On the Files tab of the Versata Logic Studio Explorer, right-click the folder where you want to reference the file, and choose Add Files.
2. In the Import dialog, browse to the file, and click the Open button.

The file is added to the Versata Logic Studio Explorer listing, with red lines in its icon. The file remains in its existing location on the filesystem and is not copied into a repository subdirectory.

You can double-click the file icon to open it for editing. If it is a Java file, it opens in the Code Editor. If it is a text file, it opens in Notepad.

Copying an existing file into a repository (Add File Copies)

To copy a file into a repository:

1. On the Files tab of the Versata Logic Studio Explorer, right-click the folder where you want to copy the file, and choose Add File Copies.
2. In the Import dialog, browse to the file, and click Open.

   The file is added to the Versata Logic Studio Explorer listing, with red lines in its icons. Also, the file is copied into the appropriate repository subdirectory.

   You can double-click the file icon to open it for editing. If it is a Java file, it opens in the Code Editor. If it is a text file, it opens in Notepad.

Creating a new file in a repository

To create a new file in a repository:

1. On the Files tab of the Versata Logic Studio Explorer, right-click the folder where you want to create the file, and choose New XDA Connector or New File.
2. In the Choose File Name dialog, enter a name for the file. If the file is a Java file, you can omit the extension. For a text file, enter the extension.
3. If the file is a Java file, complete the Create XDA Connector Class or Create Java Class dialog. This dialog provides the name of the package to include the new file, the name of the interface it implements, and the name of the superclass it extends.

   For Versata Connectors, this dialog displays defaults for the implemented interface (`XDAConnector`) and extended class (`XDAConnectorImpl`). For all Java files, this dialog displays a default package name that matches the repository name. Edit these defaults and make additions as necessary.
4. Click the Finish button when you are done.

   The file is added to the Versata Studio Explorer listing. If it is a Java file, it opens in the Code Editor. If it is a text file, it opens in Notepad.
5. Write code or text in the file. To save the file, click the Save toolbar button, or choose the Save menu option from the File menu.

Removing a user-defined file from a repository

The Versata Logic Studio Explorer provides a right-click menu option you can use to delete a non-generated file from a Versata repository. This option allows you to completely delete the file from the filesystem, or just to delete its reference from the repository and leave it on the filesystem.

An option to move files also is available. For information about this option, see "Working with groups" on page 68.

To remove a file from a Versata repository:

1. On the Files tab of the Versata Logic Studio Explorer, right-click the file and choose remove.
2. Click a button in the Versata Action Choice dialog:
   - To remove the file reference from the repository and remove the actual file from the filesystem, click Yes.
   - To remove only the file reference and leave the file on the filesystem, click No.
   - To dismiss the dialog without any action, click Cancel.

   If you have added a copy of the file or newly created it in the repository, you most likely will want to click Yes. If you have added a file reference to the repository and the file itself is located elsewhere on the filesystem, you most likely will want to click No. If you are not sure where the file is located on the filesystem, review the full path displayed in the Action Choice dialog.

Note: If you added a Java file using the Add Files option, meaning you added a reference but did not copy it to the repository, if you choose Remove and click Yes, not all class files associated with the Java file are removed. The class file with the same name is removed, but not all of the inner classes.

Adding files and packages to the classpath

You can set up and add files and packages to the classpath in the following ways:

- Add files, folders, and packages that you plan to use globally for all Versata Logic Suite projects to the Environment Options dialog. To open this dialog, choose Tools → Options from the Versata Logic Studio main menu. The Classpaths tab in this dialog provides separate text boxes to enter files and folders used by the client, those used by the server, and those used by both. Files and folders entered in this dialog always appear in the class path.

- Register files that you plan to use sometimes in the Enterprise Object Browser. To register a file, when it is open, choose Tools → Add Object to System Registry from the Versata Logic Studio main menu. To open the Enterprise Object Browser, choose Tools → Enterprise Object Browser.

  Once you have registered a file in the Enterprise Object Browser, you can indicate whether it is used by a Versata Logic Studio-generated application by enabling or disabling its check box in the Application References dialog. Open this dialog by choosing Application → References from the Versata Logic Studio main menu.

  You can indicate whether the file is used by business objects by enabling or disabling its check box in the Versata Logic Server References dialog. Open this dialog by choosing Versata Logic Server → References from the Versata Logic Studio main menu. For more information about referencing objects, see "Referencing registered objects" on page 312.

Note: You cannot register a folder or package in the Enterprise Object Browser. You can only register individual files.

Registering objects

Registering a file makes the objects in the file, and their locations, available to the system environment. Registered object classes are available to all repositories for referencing and copying, but their source code generally is not available.

You may register the following types of files: COM objects, Java class files, JavaBeans, and Enterprise JavaBeans (EJBs). These files have the following extensions: `.class`, `.jar`,, .zip and `.idl`.

To register an object:

1. In the Versata Logic Studio, choose Tools → Add Object to Registry.
2. In the dialog that appears, select the object type, then click the Add button.
3. In the browser dialog that appears, browse to the file and click the Open button.

To view registered objects:

From the Versata Logic Studio main menu, choose Enterprise Object Browser from the Tools menu.

Enterprise Object Browser

The Enterprise Object Browser displays registered objects, including classes provided with Versata Logic Suite and any other libraries registered by developers, as well as all object classes in the currently open repository. In the Enterprise Object Browser, you can select a class and display its members (methods and variables). You can copy the name for a class or member and copy the definition for a member, in order to paste it into a file. For classes and members for objects contained in the repository, you also can view source code.

- The System and Repository options allow you to choose whether to view objects contained in the currently open repository or other objects available on the system.
- The Object Libraries drop-down list allows you to choose a category of registered objects to display.
- The Methods and Variables frames allow you to specify which categories of members to display for a selected class.
- To display members for a class, select it in the Classes list box.
- Click the Refresh button to display objects that have been newly registered since the Enterprise Object Browser opened.
- The Copy Name, Copy Definition, and View Source Code buttons are enabled according to the item currently selected in the browser.

Note: Do not rename an object when the Enterprise Object Browser is open. If you do this, the Enterprise Object Browser shuts down with no warning.

Referencing registered objects

Once you have added an object to the system registry for Versata Logic Studio, you can reference it in repository objects. In order for the repository object to find the referenced object in run time, you need to add it to the project classpath.

The Versata Logic Server dialog and the Application References dialog (with presentation design only) in the Versata Logic Studio allow you to indicate objects to be referenced. All registered libraries are listed in these dialogs. To open a dialog, choose References from the Versata Logic Server menu or Application menu.

- **Local References**. On this tab you can indicate registered Java classes and Beans that are referenced. The Versata Logic Server ensures that the referenced objects are on the classpath when applications are run locally. You need to manually copy the files to the correct location when deploying, and set the classpath to that location.
- **Remote References**. On this tab you can indicate objects containing methods to be invoked remotely. The objects on this tab contain the stub interfaces required for remote access. The Application Deployment wizard for Versata Logic Studio-generated Java applications can automate copying of the referenced package(s). For Versata Logic Server references and HTML application references, you need to copy objects manually and set the classpath to the correct location.

Using a code editor

The Versata Logic Studio provides an integrated Code Editor that you can use to view code for generated Java files and external Java files that you have added to the repository. You also have the option of viewing and modifying Versata repository files' code in an external Java code editor. You set this option in the Environment Options dialog. You can use each editor at different times. If you press the SHIFT key while choosing a menu option to view a file's code, the non-default editor is used.

Using an external Java code editor

The Executables tab of the Versata Logic Studio's Environment Options dialog includes a Default Java Code Editor option. Your selection for this option indicates which program to use to display repository Java files whenever they are opened in the Versata Logic Studio. By default, the option on this tab is set to use the Versata Code Editor.

To use an external Java code editor by default:

1. Choose Tools → Options from the Versata Logic Studio main menu.
2. In the Environment Options dialog, choose External Java Editor for the Default Java Code Editor option.
3. Enter or select the full path of the external Java code editor. You can click the Browse for File button to find the file.
4. Click OK.

To use an external Java code editor when the default is the Versata Code Editor:

1. Ensure that the full path of the external Java code editor is entered in the Environment Options dialog, but leave the Versata Code Editor selected as Default Java Code Editor.
2. As you are choosing an Explorer menu option to view a file's code, press the SHIFT key.

Using the Versata Code Editor

The main purpose of the Code Editor is to write event-handling code, subclasses, and other custom code in the Java code files generated by the Versata Logic Studio. When you open these files in the Code Editor, smart code blocking indicates which code is editable and which code should not be modified. The Code Editor also may be used as a built-in text editor to edit files not generated by the Versata Logic Studio, or to create new files. You can open multiple instances of the Code Editor at one time in order to edit multiple files at once.

Note: You can edit generated files for business objects in the Versata Java Code Editor. You also can edit files generated for application user interfaces. For details on the generated files that can be edited, see “Tips for editing code in the Versata Code Editor” on page 317.

Viewing code in the Versata Code Editor

Before you begin adding event handlers or writing other custom code in Versata Logic Studio-generated files, it is a good idea to become familiar with their contents. The Versata Code Editor allows you to view all code in generated files.

Full Mode View

The following screenshot illustrates the Full Mode View of the Versata Code Editor. This is the default setting. This view allows you to scroll through all of the object's code from beginning to end. To view code for a particular method or contained object, select it from the Members drop-down list.

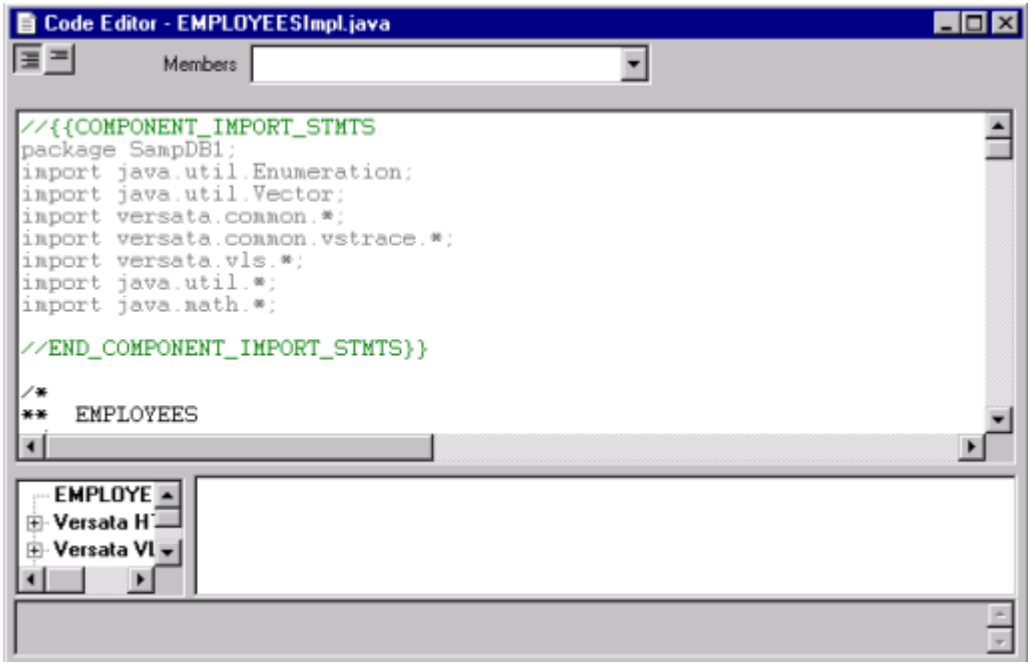

Figure 18 **Full Mode View of Code Editor**

Event Mode View

The following screenshot illustrates the Event Mode View of the Versata Code Editor. This View allows you to view code for a selected event. To set the Code Editor to this View, click the Event Mode view button, the right button of the two available. To view code for an event, select it from the Events drop-down list. If the open file is for an application form or page, first select an object from the Objects drop-down list.

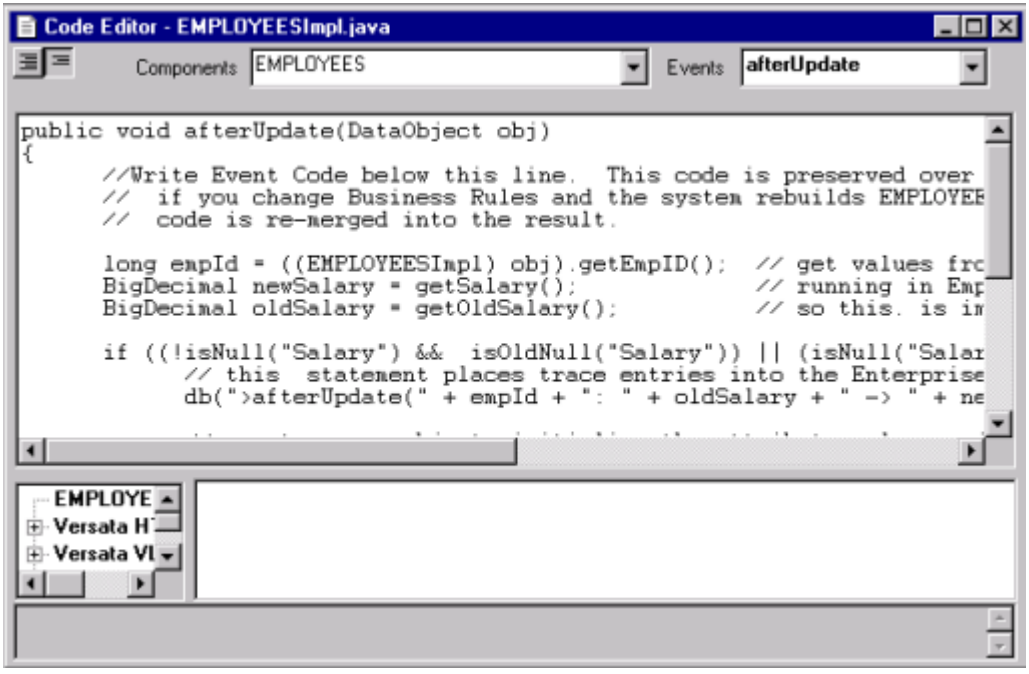

Figure 19 **Event Mode View of Code Editor**

Note: The Event Mode View is not available for objects' base implementation files, because these files do not contain event blocks.

Smart code blocking

When you edit code generated by the Versata Logic Studio, the Code Editor uses smart code blocking to protect code that you should not change.

Code is generated in code blocks. Each code block is designated by these markers:

```
//{{BlockName
<code>
//END_BlockName}}
```

To help you identify editable text within generated files, Versata Logic Suite color codes text. Black text is editable; gray and green text is not.

Note: To edit generated code, always use the Code Editor. Other IDEs do not implement smart code blocking and color-coding, so you may inadvertently make changes in a block of generated code rather than in a block reserved for your additions. If you make changes in a block of generated code, the block is regenerated the next time the object is built and your edits are lost.

Tips for editing code in the Versata Code Editor

Review the following tips before you begin editing code in the Versata Code Editor.

- To include references to existing methods, variables, or other code, double-click the text in the Code Editor Explorer to enter it automatically where the cursor is positioned.
- The Code Editor Explorer provides syntax helpers that you can use to set up expressions correctly in your custom code. To add other text, type it in as you would in any other editor.
- The following expressions are available in the Syntax Helpers list in the Code Editor Explorer: `if`, `while`, `switch`, `for`, `System.out.printIn`. To add helpers to the list, edit the file `VSSyntax.txt` in the Versata Logic Suite root installation directory.
- If you are adding code to a generated file, ensure the cursor is positioned in the correct block, usually an event block. To find an event block, click the Event Mode View button in the Code Editor and select an event from the Events drop-down list.
- To edit event code in HTML applications, use one of the following methods. In the Code Editor, click the Event Mode View button, and select a page object and an event from the drop-down lists. Or in the application diagram, select a page node, and choose Application → View Server Page Events from the popup menu, then in the Code Editor, select an event from the drop-down list. In either case, add code in the designated area.
- To restrict other developers from changing the methods and variables in a class you have written, declare the class using the `final` keyword. A method declared with the `final` keyword cannot be overridden and a variable declared with the `final` keyword cannot change from its initialized value.

Opening the Versata Code Editor as a simple text editor

The File → New File and File → Open File commands open the Code Editor as a simple text editor. Smart code blocking is disabled, so you should *not* use these commands to edit generated files. The files that you edit using these commands are not associated automatically with the repository or added to the metadata. If you want to add these files to the repository, you must do so explicitly. For information, see "Adding files to a repository" on page 308.

Printing code from the Versata Code Editor

When a file is open in the Versata Code Editor, Versata Logic Studio's File menu includes two print options you can use to print the code in the open file. The Print <File_Name> option prints all code in the open file. The Print Custom Code option prints the code you have added to the file.

Types of files that can be edited in the Versata Code Editor

You can edit the following types of files in the Versata Code Editor:

- **Implementation files for data objects and query objects.** From the Objects view of the Versata Logic Studio Explorer, right-click an object and choose Open File→ Implementation. Or, from the Files view, right-click a file and choose Edit, or double-click the file.

  To edit server event code in these files: in the Code Editor, click the Event Mode View button and select an event from the drop-down list. Add code in the designated area.

Note: You cannot edit base implementation files in the Code Editor. You can, however, review their contents and copy from them.

- **Java interface files for data objects and query objects.** From the the Files view of the Versata Logic Studio Explorer, right-click a file and choose Edit, or double-click the file.
- **Deployment descriptor files for data objects and query objects.** From the Files view of the Versata Logic Studio Explorer, right-click a file and choose Edit, or double-click the file. Note that the Code Editor does not provide Mode View buttons or drop-down lists for this type of file, because it is in XML and does not contain methods or events.
- **Other files in the repository.** From the Files view of the Versata Logic Studio Explorer, right-click a file in the Other Files folder and choose Edit.
- **Server page files (p<page_name>.java) for HTML applications**. The application must be open. In the application diagram, select a page node and choose Application → View Server Page Code. Or from the Objects view of the Versata Logic Studio Explorer, right-click a page and choose View Server Page Code from the popup menu. Or from the Files view, right-click a server page file and choose Edit.

To edit event code in these files: in the Code Editor, click the Event Mode View button, and select a page object and an event from the drop-down lists. In the application diagram, select a page node, and choose Application → View Server Page Events from the popup menu. In the Code Editor, select an event from the drop-down list. Add code in the designated area.

- **Form files (f<form_name>.java) for Java applications**. The application must be open. In the application diagram, select a form node and choose Application → View Code. Or from the Objects view of the Versata Logic Studio Explorer, right-click a form and choose View Code from the popup menu. Or from the Files view, right-click a form file and choose Edit.

  To edit event code in these files: in the Code Editor, click the Event Mode View button, and select a control and an event from the drop-down lists. Or in the Form Designer, right-click a control and choose Events. In the Code Editor, select an event from the drop-down list. Add code in the designated area.
- **Other application user interface files**. The application must be open. From the Files view of the Versata Logic Studio Explorer, right-click a file and choose Edit.
- **Other existing Java files outside of the repository.** From Versata Logic Studio's main menu, choose File → Open File. In the dialog that appears, browse until you find the file, then double-click it to open it.
- **New Java files you want to create.** For example, you may want to create subclasses of Versata classes, or create a custom Versata Connector. From the Versata Logic Studio's main menu, choose File → New File. In the dialog that appears, browse until you reach the location where you want to save the file, enter a name for the file, and click the Open button. In the confirmation dialog, click the Yes button.

Note: Event code is placed in the correct event block of the source code automatically. You do not need to create listeners or adapters for events because the Versata Logic Studio creates them as needed.

Some objects, such as data objects, can be edited in the Versata Code Editor but not created there, because they are created with wizards. Other objects, such as archetypes and data object attributes, have their own editors and cannot be edited in the Versata Code Editor.

CHAPTER 10

Extending Business Object Code

Chapter overview

Read this chapter for an introduction to extending business object code and examples of some common extensions. After reading this chapter, if you have limited experience with Java and EJBs, you should be able to copy some of the simpler event-handling code, and you should have a basic understanding of other possible customizations. If you have more extensive Java knowledge, you should be able to begin adding custom code to business objects.

This chapter includes the following:

- "Types of custom code" on page 323, provides a table listing the different types of extensions possible for Versata Logic Suite applications and objects, including those documented outside of this chapter.
- "Methods for instantiating business objects" on page 326, introduces the factory and instance methods used by the Versata Logic Server to create objects from Transaction Logic Designer definitions.
- "Server event-handling model" on page 333, introduces the event model for Versata Logic Server business objects and discusses how to add custom event code.
- "Subclassing business object classes" on page 339, provides instructions for subclassing `versata.vls.DataObject`, as well as some general information about subclassing Versata Logic Server classes.
- "Calling business object code from client applications" on page 341, describes how client applications created in the Versata Logic Studio access data from Versata Logic Server business objects, including the following specialized functionalities.
  - "Remote object access" on page 348
  - "Recomputing derivations" on page 354
  - "Computing results without saving" on page 355
  - "Java mail integration" on page 358
- "SQL expression evaluator" on page 364, describes this Versata Logic Server feature.
- "Working with Versata Logic Server security properties" on page 375, introduces Versata Logic Server security APIs.
- "Working with JTS transaction management" on page 377, explains the choice between using the Versata Logic Server or IBM's implementation of JTS for transaction management.

Types of custom code

The following table provides an overview of the types of custom code you can add to the files generated by the Versata Logic Studio. This table includes user interface files as well as business object files. Many of these customizations can be completed in the Versata Code Editor but not all of them. For more detailed information about editing user interface files, see the relevant chapters in *Application Developer Guide*.

Type of customization	Where to customize and effects of customization
Archetypes	Create a copy of a system archetype and save it as a repository or application archetype, then you can edit archetype macro code. Editing techniques are different for HTML and Java applications. Putting custom code into archetypes allows you to reuse it for all objects of a certain type across your repository.
Extended properties	When you set extended properties for an object, the Versata Logic Studio generates custom code. For data objects, set extended properties on the Properties:Extended tab of the Transaction Logic Designer. For attributes, set extended properties on the Attributes:Extended tab of the Transaction Logic Designer. For query objects, set extended properties on the Properties:Extended tab of the Transaction Logic Designer. For relationships, set extended properties on the Relationships:Extended tab of the Transaction Logic Designer. For HTML user interface objects, set extended properties in the properties sheets in the Application Designer. For Java user interface objects, set extended properties in the properties sheet in the Form Designer; for transitions and picks, you also can set properties in the Application Designer properties sheets.

Type of customization	Where to customize and effects of customization
Event-handling code	The Versata Logic Studio exposes both transaction logic (server) events and presentation (client) events. You can browse to the appropriate event blocks in the Versata Code Editor, where you can add code that is automatically preserved when objects are rebuilt. **Server code** Each data object and query object has events to which you can add code. The event blocks are in the generated implementation files. The events relate to data changes. The Versata Logic Studio generates event listeners for server events. **Client code** In HTML applications, client event coding is done in the generated file for each server page. The events exposed vary according to the type of HTML element. In Java applications, each control on a form has events to which you can add code, in the generated form file. The events exposed vary according to the type of Java control. Events generally relate to user actions. For data sources/data controls, events relate to data changes. When you add code to the event block, you ensure that it gets iterated over changes and rebuilds to your application. The Versata Logic Studio generates event listeners and event adapters for client events.

Type of customization	Where to customize and effects of customization
Subclasses	The Versata Logic Suite supplies packages of many of the classes it uses to generate business objects and applications. You can customize behavior by creating subclasses of these and using the subclasses to create some objects. To subclass, you need to create a new file. In this file, you need to declare that the subclass extends the class, and add custom variables or methods. **Server side** You can subclass `versata.vls.DataObject` and use it to generate data objects with custom methods. For more information, see "Server event-handling model" on page 333. **Client side** For HTML applications, any element may be subclassed. Select the Class Name in the properties sheet for the application object in the Application Designer or use `versata_class` in the HTML text for the page. For Java applications, you can subclass `versata.vfc.VSForm` to create a custom form or subclass any of the `versata.vfc.*` classes used to create custom Java controls. Set the ClassName in the properties sheet for the control in the Form Designer.
Other server methods	You can write your own methods. You then can add these methods directly to a business object file as custom code. Or you can reference the methods in the context of rule expressions.
Versata Connectors	If you add data objects for non-SQL data sources to your repository, or you want to provide specialized data access for a SQL data source, you need to write your own Versata Connectors. The Versata Logic Studio provides a wizard to set up the structure of a Versata Connector and open it in a Code Editor where you can add the code. You also will need to set up a data server type for the Versata Connector in the Versata Logic Server Console. For more information, see "Creating custom Versata Connectors" on page 391.
Importing third-party classes	You can import an entire library of classes to add or extend functionality. Imported classes can modify the behavior of existing methods or provide additional functionality. For example, you could use an import statement to add a third-party class. Write the custom import statement after the generated Import block. Add all classes that you are importing as packages to the classpath in order for them to be globally available for Versata Logic Server objects and applications.

Methods for instantiating business objects

This section describes system-supplied methods used to create instances of Versata Logic Server business objects and populate their attributes. The methods used to instantiate objects are called factory methods. The methods used to populate objects' attributes are called instance methods. For more information about these methods, see the Versata Class Libraries help (`vfc.hlp`) installed with the product.

Factory methods

Versata Logic Suite factory methods are methods used to instantiate objects. They are static methods that do not require an instance of a class in order to be invoked. Each implementation file for a Versata Logic Server business object includes factory methods, used to produce an object or objects against which rules code can be executed. The following are factory methods:

- `getNewObject`
- `getObjectByKey`
- `getObjects`
- `getMetaQuery`

All of these methods throw `VSORBException` when errors occur. Note that when clients need to instantiate business objects for remote access, they use different techniques.

The `getNewObject` method is called in objects' implementation files. The `getObjectByKey` and `getObjects` methods are called in objects' base implementation files. The `getMetaQuery` method is called in both the base implementation files and the implementation files.

The following code examples from the sample repository show factory methods in the base implementation file and implementation file for the CUSTOMERS data object.

This example illustrates the `getMetaQuery` and `getNewObject` methods in an implementation file:

```
//{{COMPONENT_RULES
      public static VSMetaQuery getMetaQuery() {
          //return CUSTOMERSBaseImpl.getMetaQuery();
         return getMetaQuery("CUSTOMERS", "SampDB1");
      }
      public static CUSTOMERSImpl getNewObject(Session session, boolean
makeDefaults)
      {
         return new CUSTOMERSImpl(session, makeDefaults);
      }

   //END_COMPONENT_RULES}}
```

This example illustrates the `getObjectByKey` method in a base implementation file:

```
/**
* <br>
* Factory method to create an object based on the unique key value which
* returns an object matching the key value.
* @param searchReq as SearchRequest : the key value as a SearchRequest object.
* @param aSession as Session : object to be associated with the objects.
* @return the object matching the Unique key.
*/
public static DataObject getObjectByKey( SearchRequest key, Session aSession )
   throws ServerException
{
   if ( aSession.getSecurityCheck() ) {
      try {
      if
(!aSession.getMyPrivilegeToObjectName(DataConst.AppObjectPrivilegeImpl_READ,
"CUSTOMERS", DataConst.AppObjectTypeCodeImpl_BUSINESS_OBJECT)) {
         throw new ServerException("", VSErrors.VSMSG_SecurityNoReadAccess,
"business", "CUSTOMERS","", null);
      }
      }
      catch( VSORBException e ) { e.printStackTrace();}
   }

   raiseBeforeQueryEvent( key, aSession );

   if (aSession.isTransactionInProgress()) {
      return
aSession.getTransactionInfo().getObjectByKey(CUSTOMERSBaseImpl.getMetaQuery(),
key);
   } else {
      return aSession.getObjectByKey(CUSTOMERSBaseImpl.getMetaQuery(),key);
   }
}
```

This example illustrates the `getObjects` method in a base implementation file:

```
/**
* <br>
* Factory method to get objects based on the filter (String), which returns
* an enumeration of objects matching the filter.
* @param searchReq as SearchRequest : the filter as a String. (e.g. State =
'NY').
* @param aSession as Session : object to be associated with the objects.
* @return Enumeration of objects matching the filter criteria.
*/
public static Enumeration getObjects(String filter, Session s) {
   SearchRequest searchReq = new SearchRequest();
   searchReq.add(filter);
   return getObjects(searchReq, s);
}
```

This example illustrates the `getMetaQuery` method in a base implementation file:

```
/**
* <br>
* MetaQuery on the component. This method returns a class defining
* the meta information of the component.
* @return VSMetaQuery : Meta data info class for the component.
*/
public static VSMetaQuery getMetaQuery() {
   return q;
    }
```

Example of a custom factory method

You also can write your own factory methods that return an object or an array of components other than the standard ones created in generated business objects. The following custom factory method code is from the `DEPARTMENTImpl.java` file in the sample repository.

```
public BusinessObjectCollection getAllSubDepartments() {
     Vector depts = new Vector();
     getMyDepartments(depts, this);
     BOCollectionImpl objImpl = new BOCollectionImpl(depts.elements(),
getMetaQuery());
     try {
        return
(BusinessObjectCollection)ServerEnvironment.getFactoryImpl().makeRemoteReferen
ce((Object)objImpl);
     }
     catch (Exception e) {
        e.printStackTrace();
     }
     return null;
  }

  public void getMyDepartments(Vector depts, DEPARTMENTImpl dept) {
     Enumeration depList = dept.getSubDepartments();
     while (depList.hasMoreElements()) {
        DEPARTMENTImpl dep = (DEPARTMENTImpl) depList.nextElement();
        depts.addElement(dep);
        getMyDepartments(depts, dep);
     }
```

Instance methods

Instance methods can be invoked after a class has been instantiated as an object. The Versata Logic Studio generates instance methods for each data object. You also can write your own instance methods. Instance methods are called on an object instance once a handle is obtained to the object and objects have been gathered by the caller using a static method. These methods sometimes are referred to as “getters” and “setters”.

Note: Instance methods are not provided for query objects. SQL text is used instead of “getters” for data retrieval. No “setters” are required, because query object attribute values are stored only as part of underlying data objects.

Business objects can be exposed to the client and to remote server objects if they are deployed as EJBs. Instance methods then can be made available remotely for remote method invocation from clients. In this case, you can get an object instance by calling `row.getBusinessObject()` on any row in a result set before invoking an instance method. For more information, see "Calling business object code from client applications" on page 341.

System-supplied instance methods

The Versata Logic Suite provides methods for each data object instance that obtain and set values for its attributes. These methods are contained in the rules block of the implementation file. Each attribute has a `get<Attribute_Name>`, `set<Attribute_Name>`, and `getOld<Attribute_Name>` method. This last method gets the value of the attribute before the most recent update. Virtual attributes are exceptions, because they have only a `get` method. Because their values are not stored, they cannot be set and old values cannot be obtained.

The data type for `get` and `set` methods is determined by the native data type for the attribute. For example, if the attribute's data type is a string, then the `get` and `set` methods return string values.

The following are examples of `get`, `getOld`, and `set` methods from the CUSTOMERS data object in the sample repository.

```
/**
* <br>
* method to get the City attribute for the CUSTOMERS
* @return String : the  value of the attribute City as String.
*/
public String getCity()
{
     return getData("City").getString();
}
```

```
/**
* <br>
* method to get the old City attribute for the CUSTOMERS
* @return String : the  value of the old attribute City as String.
*/
public String getOldCity()
{
     return getData("City").getPreviousString();
}
```

```
/**
* <br>
* method to set the City attribute for the CUSTOMERS
* @param String : value of the attribute City as String.
* @return nothing
*/
public void setCity(String value)
```

Examples of custom instance methods

The following are example instance methods that could be added to a CUSTOMERS data object:

- **`Public void placeOnHold()`** Updates the CUSTOMERS data object. Other mechanisms can accomplish this effect, such as calling the `save` method after the row value changes, but this method makes client coding much simpler and adds functionality, such as additional security constraints and removal of existing back orders.
- **`Public CustomerHistory() get12MonthCreditHistory()`** Passes objects to the client, which could display the information in an existing JavaBean or make it the data source of a `VSJavaDataControl` in a Java application.
- **`Public String getCreditReport()`** Connects to a credit agency to get a report using stored customer data as input, formats the result as an HTML string, and returns it to the client. Presumably, the client would then fire a second browser instance to display the information.

The calls to these routines might be added as event code on client buttons or included in client batch programs. They can be called from other server code as well and can be called from a business rule, although this would have to be routed through a utility class.

Server event-handling model

Most developers are familiar with the client-side event-driven programming model. This model divides code into segments related to events, which are distinct actions that users can initiate, for example, clicking a button. You can add code to an event in an application user interface, in order to modify the way it an event is processed.

The Versata Logic Server uses a similar event-driven model to simplify the addition of custom Java code that modifies business rules and data processing. Each business object exposes a number of distinct events in Versata Logic Server processing, which are implemented as Java listeners.

You will use these events to provide event handlers for specific situations. For example, you could use the `beforeCommit` event for an order to enforce a validation. Event handlers also could be used to provide security handling.

Transaction logic event code blocks are located in the `<business_object>Impl.java` file for the data or query object. The Versata Code Editor allows you to review the generated transaction logic events for these business objects. You can use the Verata Code Editor to add or modify event-handling code to be executed for each event. After you have added code, the Versata Logic Studio incorporates it into the event blocks for the data object or query object. This code is preserved when the files are rebuilt. For information about using the Versata Code Editor or an external code editor, see “Using a code editor” on page 313.

For a description and example of data object event code, see “Data object event blocks” on page 296. For a description and example of query object event code, see “Query object event blocks” on page 301.

How event-handling works

Objects receive events to which they have registered. They register to events by implementing listener interfaces defining events. Listeners respond to an event by providing a callback method that corresponds to the event.

Each server event object is subclassed from the superclass `versata.vls.VLSEvent`. All server listener interfaces are subclassed from `versata.vls.VLSEventsListener`.

Exposed events for data objects include: `afterCommit`, `afterDelete`, `afterInsert`, `afterQuery`, `afterRollback`, `afterUpdate`, `beforeCommit`, `beforeDelete`, `beforeInsert`, `beforeQuery`, `beforeResultSetFill`, `beforeRollback`, and `beforeUpdate`.

Exposed events for query objects include `afterQuery`, `beforeQuery`, and `beforeResultSetFill`.

Types of events

Business objects expose two types of events:

- **Transactional events.** Occur when a transaction is processed, modifying data. These events are exposed for data objects.
- **Query events.** Occur when data are retrieved. These events are exposed for both data objects and query objects.

Type of event	Event

Transactional	`afterCommit`
	`afterDelete`
	`afterInsert`
	`afterRollback`
	`afterUpdate`
	`beforeCommit`
	`beforeDelete`
	`beforeInsert`
	`beforeRollback`
	`beforeUpdate`

Query	`afterQuery`
	`beforeQuery`
	`beforeResultSetFill`

Order of processing for commit events

Commit events fire before commit and after commit of data to the data source. `beforeCommit` events occur after all rules processing for all objects in the transaction is complete. The following provides a rough outline of the processing sequence for saving two objects and committing the transaction:

Object1.`save`

Object1.`beforeUpdate` events

*** rules processing (no database save)

Object1.`afterUpdate` events

Object2.`save`

Object2.`beforeUpdate` events

*** rules processing (no database save)

Object2.`afterUpdate` events

***raise before commits to all updated objects

Object1.`beforeCommit` events

Object2.`beforeCommit` events

***send updates to database

Object1.`save` to database

Object2.`save` to database

*** commit the transaction

Object1.`afterCommit` events

Object2.`afterCommit` events

Note: Code for `afterCommit` events is executed after each row is saved.

Adding server event-handling code

To review or modify event-handling code for a data object or query object:

1. In the Versata Logic Studio Explorer, click the Files button and double-click the object's implementation file to open it in the Versata Code Editor.
2. The Code Editor has two buttons in its upper left corner; click the right button. Select an event from the drop-down list box.
3. Add event-handling code below the words "`Write Event Code below this line`".
4. Choose File → Save File.

Event-handling code examples

The sample repository includes examples of server event-handling code that you can use as models to write your own event handlers. Event-handling code is added to a business object, while code utilizing the server logic is in a sample application.

The following code from the EMPLOYEES data object in the sample repository records any salary changes in the EMPLOYEESAUDIT data object. You can run the `Server_EventAction_CreateChildren` application in the sample repository to see how this event-handling code executes.

```
public void afterUpdate(DataObject obj)
{
   //Write Event Code below this line.

   long empId = ((EMPLOYEESImpl) obj).getEmpID();
   BigDecimal newSalary = getSalary();
   BigDecimal oldSalary = getOldSalary();

   if ((!isNull("Salary") &&  isOldNull("Salary")) || (isNull("Salary")
   /&&  !isOldNull("Salary")) || (!newSalary.equals(oldSalary))) {


      EMPLOYEESAUDITImpl empSalHist =
      EMPLOYEESAUDITImpl.getNewObject(getSession(), true);
      empSalHist.setOldSalary(oldSalary);
      empSalHist.setNewSalary(newSalary);
      empSalHist.setEmployeeID(empId);
      empSalHist.save();     }

}
```

The following code from the ORDERS data object in the sample repository prevents the insertion of null orders (orders without any order items). This code is added to the `beforeCommit` event, because its transaction logic can be executed properly only after updates have been processed. You can run the `Server_Event_Commit` application in the sample repository to see how this event-handling code executes.

```
public void beforeCommit(Session session, Response response)
{
   db("Order>>beforeCommit with getOrderTotal=" +
getOrderTotal());
   String isUseTransForSave =
   session.getProperty("isUseTransactionForSave");
   if ( isUseTransForSave != null &&
   isUseTransForSave.equalsIgnoreCase("true"))   {
   db("..checking Order Total to assure some items placed");
   if ( getOrderTotal().compareTo(new BigDecimal("0.01")) == -1
   {  // less than 0.01
   raiseException("Sorry, Orders must have Line Item
Information"); } }
}
```

The following code from the CustQueryVirtuals query object in the sample repository provides an example of a `beforeResultSetFill` event handler. You can run the `Server_Event_QueryAttributes` application in the sample repository to see how this event-handling code executes.

```
public static void beforeResultsetFill(DataRow rowToBeAdded,
Response response)
{
   System.err.println("CustQueryVirtual: " +
    rowToBeAdded.getData("Name").getString() );

   Enumeration e = rowToBeAdded.getAllColumnValues();
   Data d;
   while ( e.hasMoreElements())
{
      d = (Data)e.nextElement();  // get Data Object
      System.err.println(" Data: " + d);                      }

   System.err.println("  CustQueryVirtual : " +
    rowToBeAdded.getData("AddressLine"));
   rowToBeAdded.getData("AddressLine").setString("lll");
   System.err.println("  CustQueryVirtual: " +
    rowToBeAdded.getData("Name").getString() );

   System.err.println("");

}
```

Subclassing business object classes

In addition to adding event-handling code to the implementation files for data objects and query objects, you may want to make other sorts of modifications to server processing. To make these kinds of modifications, you can subclass new server classes from classes in `versata.vls.*`. To create a subclass, write the source code in the Versata Code Editor (or another IDE tool), then compile the class file, and make the file available to applications. If you create a subclass of another server class, you may add the file to the repository or add it to a Versata Logic Studio classpath so that it can be used.

Experienced Java developers can use classes from other libraries or original classes that they write themselves. When using any class that is not subclassed from the system-supplied libraries, the class must implement the same interfaces that the comparable Versata Logic Suite classes implement, to ensure compatibility with the system-supplied libraries.

Subclassing versata.vls.DataObject

The class that you most commonly will subclass is `versata.vls.DataObject`, which is used as a default superclass for all generated data object base classes.

Creating a DataObject subclass with specialized methods

Creating a data object subclass with specialized methods rather than adding them to a particular data object renders the methods shareable among many business objects. When you reference a method that is a member of `versata.vls.DataObject`, you do not need to specify a class name, because it exists in the currently selected data object. You can create a subclass to provide extra business services, such as date functions, financial operations, and mail.

To create a subclass of DataObject with specialized methods:

1. In the Versata Logic Studio Explorer, click the Files button. Expand the Versata Logic Server folder.
2. Right-click the Versata Logic Server folder or one of its group folders and choose New File.
3. In the Choose File Name dialog, enter a file name and click the Next button. In the Create Java Class dialog, enter `versata.vls.DataObject` in the Extends field and click the Next button. In the Finished dialog, click the Finish button.
4. In the Versata Code Editor, add the specialized methods to the class file and save your changes.

Note: Review the `CorpReuseExtRulesDataObject` in the sample repository for an example of data object subclassing to extend rules. To open the file for this object, on the Files tab of the Versata Logic Studio Explorer, expand the JavaExtensions folder, and then double-click the file to view its code in Code Editor.

Applying a DataObject subclass to data objects

After you have created a subclass of `DataObject` and added methods to this file, you need to determine which data objects need the methods from that subclass.

To specify a DataObject subclass as the superclass for a data object:

1. In the Explorer, click the Objects button. Expand the Business Objects folder.
2. Double-click the object to open the Transaciton Logic Designer.
3. In the Transaction Logic Designer, click the Properties:Data Access tab.
4. In the Superclass frame, enter the name of the new class file you created and save your changes.
5. Repeat as necessary.

These data objects now include the new methods and you can reference these methods without specifying a class name. (If a rule expression references a method that does not exist in the rule's data object or any of its superclasses, the method name must include the class name.)

Calling business object code from client applications

This section covers how applications created in Versata Logic Studio use business object methods, including remote method invocation.

The `versata.vls.DataRow` class has `getComponent`, `getData`, and `getMetaColumn` methods that get a handle to a row of a specified business object, including attribute values and metadata for the row.

The interface `versata.vfc.VSResultSet` has a `save` method. This method takes all the pending changes in the client and passes them to the Versata Logic Server session object, using the `save(ORBrow)` method. The session object then creates a business object for each row and calls the `save` method on the data object.

Data access to result sets

Applications can access data from data sources through the Versata Logic Server's Transaction Logic Engine. Its data access code executes direct queries on business objects to return collections of objects as result sets. Collections are not scrollable; you can only move forward through the results.

Object caching

Modules in the run-time client and the Transaction Logic Engineuse buffered updates and object caching to optimize response time for individual clients while minimizing the load on server and network resources.

Until a user saves the result set, update requests are buffered in the client by the current data control using the result set. Because the system caches the changes in the client, it does not have to call the server each time the user changes data in the client application. The system makes a transaction request to the Transaction Logic Engine only when the end user explicitly saves the updates. The Transaction Logic Engine instantiates the business objects during the transaction and these objects are cached until the transaction completes.

Caching parent data

The Transaction Logic Engine caches the parent objects while processing child objects. This caching improves performance because only one trip to the database is required for getting a parent object for multiple child updates, such as updating summed and counted attributes, and verifying referential integrity.

This type of object caching can be monitored in the trace file in the Versata Logic Server Console's tracing monitor by reviewing `update` statements to the parent object. For example, if a parent data object maintains the sum of a child attribute, there should be only one statement to update the parent sum in the trace file. Any other processing should take place in memory.

Caching security data

For information about caching security data, see the *Administrator Guide*.

How an application queries a database

The following is a detailed account of how an application created in the Versata Logic Studio queries a database:

1. When the system retrieves information from the database (through form initialization, query by form, or otherwise), the Versata Connector uses the name of the metaquery to locate the `VSMetaQuery` class associated with the RecordSource on the form or page.
2. The system concatenates the SQL text defined in the parameters with any `whereClause`, `orderBy` and metaquery supplied by the `searchRequest`. The system sends the resulting SQL to the database, and stores it internally in case the same query runs again.
3. When requested, the Versata Connector sends the query through a JDBC statement.
4. Once the query is sent, the system creates a result set object as a cache for rows returned from the server, and passes the result set's name to the Java application's data control or to the HTML application's data source. The data control or source must request records from the new result set to display them.
5. If more rows are retrieved than the `VSResultSet` can hold (16 rows, by default), the system holds excess rows in a JDBC buffer.
6. As the data control or source requests new rows (such as when the user scrolls down a grid), the Versata Connector takes them from the JDBC buffer and converts them into a `VSRow` object that is held in the `VSResultSet`.

Query instance

A single execution of a query on the database is a query instance. Presentation logic data access code creates an instance of `versata.vfc.VSQuery` as the query. The query instance references the metaquery, the current logical session, and any run-time parameters entered by the user. The query instance runs the query to produce the result set and applies any changes from the result set to the database. The query instance also coordinates the transactional behavior as changes are applied to the database.

The query instance references the metaquery to determine the shape of the result set. Because it can access this metaquery information, the query instance can provide defaults and determine whether input is valid, whether an attribute allows nulls, and whether an attribute is calculated, without accessing the server. The Presentation Logic Engine's validation optimizes performance by minimizing the number of times an application must access the Transaction Logic Engine.

Though the query instance manages the work flow and logic, it relies on the session object and the metaquery object in order to gather information regarding the type of database used, valid values for insert and update, and the shape of the query.

Query definition

A query definition is a metaquery. It is information about a query, defined in metadata, that can be used by multiple query instances. It describes the query objects and data objects involved in the query, and it is used by query instances to determine the shape of the result set. It is also used for client-side validation.

A query definition determines the shape of the result set, for example: the number, names, and data types of the attributes that are returned; the data objects to be included; which data object is set as the childmost data object. It also includes information about the values in each attribute: including whether there is a default value, whether it is a derived attribute, whether it participates in optimistic locking, and other properties.

Query definition generation

When you generate business objects, query objects and data object queries are translated into `VSQueryDefinition` and `VSQueryColumnDefinition` classes.

Each query object in the repository has a corresponding `VSQueryDefinition` object, which provides the following metadata:

- Data objects participating in the query.
- Which data object is the childmost in the query.
- User-defined SQL text.
- A collection of `VSQueryColumnDefinition` objects. These each have a one-to-one relationship to an attribute projected in the query object.

Each `VSQueryColumnDefinition` provides the following attribute-level metadata:

- Base data object name.
- Base attribute name.
- Data object alias.
- Attribute alias.

A derived attribute has an attribute alias, but no attribute name.

Query run-time behavior

At run time, when a query runs against the database, the name of the appropriate metaquery class passes to the query instance, which finds the actual class and concatenates the developer-defined SQL text with the additional `whereClause` and `orderBy` objects. The system sends the SQL statement that is generated to the Transaction Logic Engine, which executes it through a Versata Connector.

The `VSResultSet` created by the query class consists of rows retrieved from the query. These rows are ready for display and other user interface manipulation. The following limits apply to user actions on records retrieved through query objects:

- Inserts and deletes can be applied only to the childmost data object in the result set, unless the `ParentInsertable` flag is set. If the result set has no childmost data object, the system raises an error if the user attempts to insert or delete.
- Updates are always allowed even if there is no childmost data object.
- Non-derived attributes projected in the query object can be updated only if the data object has a primary key that is projected in the query object.
- Derived attributes cannot be updated.

An update to a joined query is split into multiple, individual data object updates. The updates occur in sequence, childmost data object first. If any element of the update fails, the entire sequence of updates is rolled back.

After a row is committed, the entire row is refreshed, including attributes from the modified data object and any attributes from data objects that were not modified.

You can set the `ParentInsertable` flag with query properties or with code. If this flag is set, inserts cause the insertion of both a child record and a parent record.

whereClause and orderBy

You explicitly define SQL text for query objects in the Versata Logic Studio. SQL text for data object queries is built at run time. The additional `whereClause` and `orderBy` are developer- or end user-defined SQL strings that are concatenated at run time with the developer-defined SQL text of a query object or system-generated SQL text of a data object query.

You can define SQL text in a number of places:

- In the Versata Logic Studio on a RecordSource or transition properties sheet.

  In both cases, add additional `whereClause` and `orderBy` information on the Query tab of the properties sheet. In Java applications, the generator automatically calls the `setQueryInfo` API of `VSDataControl`. The information passes to `VSFormNavigation` through the constructor parameter. In HTML applications, the generator automatically calls the `setQueryInfo` API of `DataSource`. The information passes to `PageNavigation` through the constructor parameter.

  The `whereClause` and `orderBy` created in this situation are persistent, that is, they will always be added to queries initiated on the associated data control or DataSource.
- To provide `whereClause` and `orderBy` information for Java applications at run time, you can use the `setQueryInfo` API of `VSDataControl` and the `setSearchAndSortCriteria` API of `VSFormNavigation` and `VSPick`. To provide `whereClause` and `orderBy` information for HTML applications at run time, you can use the `setQueryInfo` API of `DataSource` and the `setSearchAndSortCriteria` API of `PageNavigation` and `Pick`. In this case, the `whereClause` and `orderBy` are persistent as well.
- In Java applications , call the `executeQuery(whereClause, orderBy)` method of `VSDataControl` to include the supplied `whereClause` and `orderBy` in the query. You can call the `startForm(formName, whereClause, orderBy)` method of `VSForm` to supply the `whereClause` and `orderBy` to the root data control of the form when it is opened.

The executeQuery method appends the supplied whereClause to any persistent whereClause and substitutes the orderBy in the method for any persistent orderBy. These whereClauses and orderBys are not persistent. They are replaced when the end user provides further information through an onscreen query or through the Search and Sort Criteria dialog.

- You can use the Versata Code Editor to write code in the BeforeQuery event of a data control or DataSource. The event passes the query object, and you can call query.addFilter(whereClause)to add the whereClause to another developer-defined whereClause or to any user-defined information.

  For example, consider an application in which all queries use the following criteria:

  ```
  UserName = USER()
  ```

  Include this SQL text in the whereClause argument of the query.AddFilter method.

  Similarly, you can call query.replaceSortCriteria() to replace any developer-defined orderBy.

- To execute a query without using a data control or DataSource, pass the whereClause and orderBy when you create the query object, then open a result set.

  For example, you could use code like the following in a Java application:

```
VSQuery query = new VSQuery(metaQueryName, whereClause, orderBy);
VSResultSet rs = query.execute();
```

In Java applications, the end user specifies the SQL that resides in the whereClause and orderBy objects in one of two ways:

- Using the Search and Sort Criteria dialog to enter a whereClause and orderBy.
- Entering search criteria directly on a form, and using the Get Data button on the default toolbar.

The system appends the user-defined whereClause to any persistent developer-defined whereClause. The orderBy defined by the user replaces any developer-defined orderBy.

Server data access by SQL string

You can supply string select statements and Where clauses in your code to provide Transaction Logic Engine data access. Using this method has implications for cache management and rules enforcement, so use it only when absolutely necessary. You may prohibit the use of ad hoc SQL statements for data access by setting the SQLAllowed property of a data server in the Versata Logic Server Console to false. For information, see the *Administrator Guide*.

Most Transaction Logic Engine code performs services that support updates, where system-constructed rule enforcement code also executes. System code builds a sophisticated cache, to avoid rereading object instances multiple times within a transaction. This "object sharing" logic is required for complex update processing. The object cache sorts this complexity, and ensures that the system is operating upon a single consistent instance of each object. This cache management is automatically enabled for business object queries that are based on relationships and `SearchRequest` objects, but is not available for SQL string queries.

The Versata Logic Suite provides services to simplify SQL queries that return results. You can create queries by calling the `versata.vls.Session` method `GetResultSetBySQL`. Results return as rows, and attributes as data objects. Accomplish updates through separate update commands using the `versata.vls.Session` method `executeUpdate`, rather than by altering result rows. When you use basic SQL instead of queries on known components, the system has no information about the business objects in the query and so cannot perform query decomposition to disburse query object updates to underlying data objects.

The Versata Logic Server fully enables basic JDBC-level database access. This access can be useful in allowing you to process queries with multiple result sets and other server-specific features.

Note: Direct SQL updates are not subjected to business rules, so if you use this method be certain your updates have no side effects and are written correctly. You also must ensure that security provisions are made.

Methods to get related data object records

The rules compiler builds standard business object methods that access data from related objects.

The standard methods for each data object instance to obtain records from its related data objects are called `get<Parent_Role_Name>` and `get<Child_Role_Name>`. The system bases the names of these methods on the parent and child role names defined on the Relationships:Presentation tab of the Transaction Logic Designer. The Versata Logic Studio provides defaults that you can modify.

In a situation where a CUSTOMER data object is a parent of ORDER, and ORDERITEM is a child, the `getPlacedbyCustomer` method would get the parent CUSTOMER object for the given ORDER object, while the `getOrderItems` method would get the list of ORDERITEMS for the given ORDER object.

Other static methods

Other static methods on the business object, such as `getObjects`, provide the necessary services to return collections. The rules compiler also constructs `get` and `set` methods for each business object, which you can use to read and write attribute values. To modify the business objects in the server, issue `insert`, `update`, or `delete` methods directly to data objects in the result collection. For information about building business object collections, see "Building business object collections" on page 352.

Editing server result set code

You can use the above standard and static methods in your server code. Also, you can code SQL requests by using `SearchRequest` objects. We recommend using these component-based mechanisms as opposed to writing your own SQL strings in server data access code.

If you are creating result sets through your own code, call a `close()` on any result set once the user is finished with it. Otherwise, the resources that are held by a particular result set are not available.

Remote object access

Remote object access is supported in the Versata Logic Server environment for objects that are running in the Versata Logic Server and need to be accessed by client programs at the object API level. The EJB specification describes how access can be achieved; the Versata Logic Studio implements the EJB specification and automates the creation of most of the files that are required.

Making an object available for remote access is sometimes called "remoticizing." Invoking the methods in the remote object is called "remote method invocation."

For Versata Logic Server business objects, the remote interface, home interface, and deployment descriptor files required by EJB for remote access are generated automatically by the system when deployment as an EJB is elected. The home interface file for each object extends `javax.ejb.EJBHome` and includes the `findByPrimaryKey`, `findObjects`, and `create` methods (with Versata-specific parameters).

Making methods remotely accessible

The Versata Logic Studio provides tools for you to make generated business objects and their methods accessible to other objects. The following are the basic steps for making a generated business object's methods remotely accessible to a application designed in the Presentation Logic Designer:

1. Define the business object as available for remote access.
   - For a data object, you do so by enabling the Deploy as EJB Entity Bean check box on the Properties:Data Access tab of the Transaction Logic Designer.
   - For a query object, you do so by enabling the Deploy as EJB Session Bean check box on the Properties:General tab of the Query Object Designer.
2. Add the remotely available method(s) to the remote interface file for the business object.
3. Build and compile the business object files.
4. Deploy business objects to a development environment Versata Logic Server running on IBM WebSphere Application Server 4.0 Single Server Edition. The Versata Logic Studio generates the skeleton and stub files required for remote access. The Deployment Manager deploys the new skeleton and stub files as well as the business object files, and creates an EAR that is deployed to an enterprise application on IBM WebSphere Application Server.
5. Enable the `.jar` file containing the deployed business object as a reference for the application. Choose Application → References, click the Remote References tab, and enable the check box for the `.jar` file.
6. Once you have tested in the development environment, use the system-supplied batch file to deploy business objects to a production Versata Logic Server running on IBM WebSphere Application Server 4.0 Advanced Edition.
7. Add the method call to the client's event code. Versata Logic Suite provides two different ways for a client to access a server object's code, via row or via factory. The `Demo_BusinessObject_Methods` application in the sample repository provides sample code that illustrates these different types of access.

   Normally Versata clients retrieve data as rows. Each row corresponds to an instance of a business object of the type on which the result set of this row was defined. So, if there is an object's result set available, an object instance can be obtained by asking a particular row about it. You can imagine the existence of a "factory" that can "produce" required objects, whether there is a row in the client or not. In this case, first, a "factory" for an object is obtained from the server. Then a request is made on this factory to return an object instance based on a key.

   Once an object is obtained through either of the two mechanisms, the calls to any methods on this object are identical. In fact, calling methods on this object is just the same as if the methods were on a local object.

Integrating with custom applications and business objects

Client applications developed outside of the Versata Logic Studio can make remote method calls on Versata Logic Server business objects that are deployed as EJBs. These applications do so in the same manner they would make remote method calls on any EJB.

Applications generated in the Versata Logic Studio can make remote method calls on EJBs created outside of the Versata Logic Studio with, for example, IBM Visual Age for Java. Versata applications do not have any special requirements; you simply need to create the object in accordance with the EJB specification. You need to add the files for the EJB to the repository, or register the files in the Enterprise Object Browser and add them as references for the application. Also, you need to add and configure the EJB in IBM WebSphere Application Server yourself, as the Versata Logic Studio does not automate this step for external business objects. You can add client event code method calls to any method included in the EJB remote interface file.

Accessing remote objects from clients

Once you have enabled remote access for business objects, you can invoke remote methods on these business objects. These methods run within the Versata Logic Server's Transaction Logic Engine. You can use either of the following techniques for remote access:

- Obtain a remote object from an existing client row instance and issue the method.

  If you already have obtained a row from the Transaction Logic Engine, you can convert it to a remote object, and then address its methods.

  The following sample code illustrates this technique. This code obtains a DEPARTMENT object and calls its `defaultMission` method. This code is from the `btnSvr_actionPerformed` event on the DEPARTMENT form in the `Basic_Data_Access` sample application.

```
try {
   VSRow row = datT1DEPARTMENT.getCurrentRow();
   SampDB1.DEPARTMENT dept = (SampDB1.DEPARTMENT)
row.getBusinessObject()dept.defaultMission(choiceSetMissionServer
Code.getText ());
datT1DEPARTMENT.refreshCurrentRow();
datT1DEPARTMENT.refreshControls();
}
```

- Obtain a remote object from the Transaction Logic Engine by object factory. If you have not already fetched a row object that you can convert to a remote object, retrieve remote business objects by calling one of the factory methods. Remote factory methods belong to a helper class used to create business objects. These methods encapsulate EJB-required APIs.

  The example code below illustrates the use of this technique. First the code prepares a search request. It then uses the `getObjectByKey` method from the `versata.vfc.RemoteFactory` class. The system casts the returned object as an EMPLOYEES business object. This call runs the query, obtains the row, and returns a remote handle (`emp`) to the client. The client can then issue remote methods (`giveRaise`) on this object.

  This code is from the `actionPerformed` event of the VSOKButton on the ObjectByFactory form in the `Demo_BusinessObject_Methods` sample application.

```
try {
VSRow row = datT1EMPLOYEES.getCurrentRow
//Write Event Code below this line
//Get the business object Factory
VSSession s = VSApplicationContext.getSession();
//Get the object
versata.common.Parameter param = new
    versata.common.Parameter("EMPLOYEES", "EmpID",
     row.getData("EmpID").getString());
versata.common.SearchRequest filter = new
versata.common.SearchRequest();
filter.add(param);
SampDB1.EMPLOYEES emp = (SampDB1.EMPLOYEES)
    RemoteFactory.getObjectByKey(s, "EMPLOYEES", filter);
//Invoke a method
emp.giveRaise(Integer.parseInt(VSTextField1.getText()));
row.refresh();
}
```

  The Versata Logic Server maintains the row until you release your handle to it. This occurs when this routine exits and the `emp` object is deallocated. Then the system deallocates the Versata Logic Server object.

Creating rows versus creating objects

Just in Time Objects conserve server resources by deferring object instantiation until updates are submitted.

An important consideration when delivering data to the client is whether to send the data as instantiated objects, or as self-contained rows. The Transaction Logic Engine provides APIs to obtain either rows or objects, but sometimes it is better to use rows and sometimes it is better to use objects.

In general, when sending data rows back to the client, it is not advantageous to send them as objects, because each row would have to be instantiated on the server, which is a significant cost in shared memory. Also, when sending objects, client access to the row requires a server call, such as retrieving each column value to display on the screen. In most situations, it is better to use rows. The rows are used by the system in a “Just in Time Objects” scheme, which operates as follows: Rows are sent back as highly optimized byte arrays, which are converted to numbers, strings, and dates in the client. No server object is created for each row at this time. However, the component identity is saved with the query and accessible by each row, enabling the system to instantiate the component when necessary.

Objects are the better choice, however, when sending data back to a client that must access related data from the server, because of improved performance and cache checking. In this case, because components access data with the intent to alter it, it is faster to instantiate the component as an object immediately rather than to create intermediate row objects. In addition, the component access APIs also provide automatic cache checking, which optimizes performance.

Building business object collections

You can write server methods that return computed collections of objects that interoperate with existing automation services such as scrolling and updatable joins.You can write a server method that returns a new object called a business object collection, which is a set of business objects. You can build this collection with your code in addition to normal SQL commands. Second, you can convert these collections on the client to result sets, so they interoperate with all existing system services for scrolling, update, data bound behavior, and other services.

For example, if you want to retrieve all the subdepartments of a department, you cannot obtain this result with a SQL query. This example requires transitive closure and you must provide a programmatic means of concatenating a series of recursive queries.

The following example code accomplishes this goal. This example code is from the `BeforeQuery` event of the datT9DEPARTMENT data control on the Department form of the `Basic_Data_Access` sample application:

```
void datT9DEPARTMENT_BeforeQuery(VSQuery query,VSOutParam rs )
{
//Write Event Code below this line
VSRow row = datT1DEPARTMENT.getCurrentRow();
SampDB1.DEPARTMENT dept = (SampDB1.DEPARTMENT)
row.getBusinessObject();

BusinessObjectCollection depts = dept.getAllSubDepartments();
rs.setValue(VSApplicationContext.createResultSet
(VSApplicationContext.getSession(), depts, "DEPARTMENT"));
}
```

This example code is from the `getAllSubDepartments()` method of the DEPARTMENT data object used by the `Basic_Data_Access` sample application:

```
public BusinessObjectCollection getAllSubDepartments() {
     Vector depts = new Vector();
     getMyDepartments(depts, this);
     return (new BOCollectionImpl(depts.elements(),
getMetaQuery()));
   }
public void getMyDepartments(Vector depts, DEPARTMENTImpl dept) {
     Enumeration depList = dept.getSubDepartments();
     while (depList.hasMoreElements()) {
     DEPARTMENTImpl dep = (DEPARTMENTImpl) depList.nextElement();
     depts.addElement(dep);
     getMyDepartments(depts, dep);
}
```

In this example, the client event-handling code invokes the remote method `getAllSubDepartments()` that returns a business object collection. This method is defined in the DEPARTMENT data object. The data is retrieved by recursive calls to the relationship-based method to obtain subdepartments, concatenating each new result into a vector. When the transitive closure is complete, the method uses the new `BOCollectionImpl` service to convert the vector to a business object collection. The collection is returned to the client method, where it is converted into a result set and assigned to the data control's `recordset`. All operations that are automated for conventional result sets are automated for this result set, including buffered scrolling, bound controls, updates, picks, and error handling.

Recomputing derivations

After you have deployed the business objects that include rules execution code, rules execute against the data source(s) so that any data values entered subsequently conform to the rules or cannot be saved. However, since any preexisting data values may not conform to rules, the system provides a `recomputeDerivations()` method that you can execute to modify preexisting data so that it does not violate rules. You can use the Versata Logic Studio to create an administrative application that incorporates this API in its client event coding.

The system has reserved a special user defined event called `RECOMPUTE_DERIVATIONS`. When this event is set to be the current event, code for the `recomputeDerivations()` method for the current data object is executed. This code performs derivations on any preexisting data values for the data object's derived attributes. The execution of this method ensures that even values that were entered to the data source before deployment of business objects and rules code conform to rules.

To implement this functionality, you can create an administrative application that displays data for all of the data objects for which you want to enable recomputes. Then, on each form or page that displays data object data, you can add a button that contains client event code setting the current event to `RECOMPUTE_DERIVATIONS`, thus causing the `recomputeDerivations()` method to fire for the current data object. Add this event code to the button's `actionPerformed` event, so clicking the button causes the code to be executed. This event code can cause only the currently selected row to be recomputed, or can cause a recompute of all rows by setting up a loop. In most cases, you will want recompute all rows.

The following code sample illustrates event code used to cause a recompute of the current CUSTOMERS row, on a button named `VSRecompute`. This code is taken from the `Recompute_Derivations` sample application:

```
void VSRecompute_actionPerformed()
{
//Write Event Code below this line
datT1CUSTOMERS.getSession().setUserDefinedEvent("RECOMPUTE_DERIVATIONS")
;
datT1CUSTOMERS.getCurrentRow().save();
datT1CUSTOMERS.refreshControls();
}
```

The following code sample illustrates event code used to cause a recompute of all rows in CUSTOMERS, on a button named `VSRecomputeAllRows`. This code is taken from the `Recompute_Derivations` sample application:

```
void VSRecomputeAllRows_actionPerformed()
{
//Write Event Code below this line
datT1CUSTOMERS.refreshControls(); */

int pos = datT1CUSTOMERS.getResultSet().cursorPosition();
datT1CUSTOMERS.first();
do
   {

datT1CUSTOMERS.getSession().setUserDefinedEvent("RECOMPUTE_DERIVATIONS")
;
      datT1CUSTOMERS.getCurrentRow().save();
   } while(datT1CUSTOMERS.next()!=null);
   datT1CUSTOMERS.setCurrentRow(pos);

}
```

Note: The `recomputeDerivations()`method is available only for persistent attributes.

When you are designing your administrative application to perform recomputes, consider the order of computation. It is best to recompute from the "bottom up". For example, recompute ORDERITEMS before ORDERS, and ORDERS before CUSTOMERS.

Computing results without saving

The Versata Logic Server provides a way for you to test rules definitions on applications generated by the Versata Logic Studio, allowing you to execute rules and review results without altering data in the data source. This "no-save" compute uses the default `save()` method to trigger the execution of business rules and user-defined methods on the server and return the results to the client, relying on the transaction control capability of the database to roll back the changes at the end of the operation.

The no-save compute locks database resources for reference or update during the transaction, but does not commit changes without a separate confirmation from the user. As soon as results are computed and refreshed to the client, the database immediately rolls back the transaction, so the session does not use database resources longer than necessary.

To compute rules results without saving, you need to design three client actions and attach their code to buttons in a test application. The three actions are: Submit for Compute, Confirm Submit, and Cancel Submit.

The Submit for Compute action starts a client-side transaction. A client-side transaction is used for the following reasons: the user needs to issue a rollback after the calculation, and the client result set needs to maintain the changes in the stack. This action sends all updates to the Transaction Logic Engine by executing the `updateDataSource` method for the data control (Java application) or DataSource (HTML application). Next, the action executes a refresh and requery on any row or data control, in order to capture all changes indirectly performed by the client within the transaction. Last, this action rolls back the transaction. After the Submit for Compute action has been completed, the normal Save, Query, and Undo buttons should not be available on the form.

The Confirm Submit action refreshes all updated rows, while keeping all user changes, in order to reset all temporary computed values resulting from the previous action. Next this action triggers a normal `save()` on the data control or DataSource.

The Cancel Submit action refreshes all updated rows, while keeping all user changes, in order to reset all temporary computed values resulting from the previous action. After this action is complete, the user can make further changes and resubmit. After the Cancel Submit action has been executed, the normal Cancel button should be available, so the user can choose to restore original values.

The following example code is for a form displaying data for a CUSTOMER, ORDERS, and ORDERITEMS, where users can change the quantity of ORDERITEMS.

This code example is for the Submit for Compute action:

```
void VSButton1_actionPerformed()
{
//Write Event Code below this line
try {
VSApplicationContext.getSession().beginTrans();
datT5ORDERITEM.updateDataSource();
//get all changes on parent or child row(s)
datT3ORDERS.getCurrentRow().refresh(true);
datT1CUSTOMERS.getCurrentRow().refresh(true);
//refresh control
datT3ORDERS.refreshControls();
datT1CUSTOMERS.refreshControls()/
VSApplicationContext.getSession().rollback();
}
catch(Exception ex) {ex.printStackTrace();}
}
```

This code example is for the Confirm Submit action:

```
void VSButton2_actionPerformed()
{
//Write Event Code below this line
//restore all derived values
datT5ORDERITEM.getCurrentRow().refresh(true);
datT5ORDERITEM.updateDataSource();
}
```

This code example is for the Cancel Submit action:

```
void VSButton3_actionPerformed()
{
//Write Event Code below this line
datT5ORDERITEM.getCurrentRow().refresh(true);
datT3ORDERS.getCurrentRow().refresh(true);
datT1CUSTOMERS.getCurrentRow().refresh(true);
datT5ORDERITEM.refreshControls();
datT3ORDERS.refreshControls();
datT1CUSTOMERS.refreshControls();
}
```

Java mail integration

The Versata Logic Server provides Java mail integration in conformance with the Java Mail API specification. This functionality allows an application, or the server it is running against, to send an SMTP mail to an internet mail address. The Versata Logic Server provides a convenience class, `versata.vls.SendMail`, that wraps this functionality. The `sendMail` method also is exposed as a remote method on a session object, allowing a client application to send Internet mail.

The following table lists the methods provided by this class and their purposes:

Method (argument)	Purpose
`SendMail (String hostName)`	Creates a sendmail object to send a single-part message. HostName is the name of the mail server, for example, exchange.
`SendMail (String hostName, boolean isMultiPart)`	Creates a sendmail object to send a multi-part or single-part message.
`setRecipientsTo (String[] to) throws SendMailException`	Sets the recipients' Internet mail addresses.
`setRecipientsCC (String[] cc) throws SendMailException`	Sets the cc recipients' Internet mail addresses.
`setIFrom (String from) throws SendMailException`	Sets the sender Internet mail address.
`setSubject (String subject)`	Sets the subject.
`setMsg (String msg)`	Sets the message. You can use this method to attach a URL – make the URL the parameter.
`addMultiPartMsg (Object data, String mimeType)`	Adds a multi-part message object. This method can be called multiple times to add messages of different types. Can be used to send text data only.
`attachFile(String path)`	Can be used to attach a file of any type.

Method (argument)	Purpose
`send(int transactionType) throws SendMailException`	Sends the mail. Possible values for transactionType are: 0 (non-transactional), 1 (on commit), 2 (on abort).

The `versata.common.VSSession` class has two methods that provide remote access to the `sendMail` methods, giving the client application the ability to send mail:

- `public void sendMail(String[] to, String[] cc, String from,String mailServer, String subject, String msg, short transactionType) throws VSException;`
- `public void sendMail(String to, String cc, String from, String mailServer, String subject, String msg, short transactionType) throws VSException`

The following packages are needed to implement Java mail integration: `activator.jar` (Java activation framework classes) and `mail.jar` (Java mail classes).

Setting up an email notification system

Using business rules and SQL Server mail, you can create a system that automatically notifies pre-selected client application users when a given event has occurred. For example, you could notify the credit manager when any customer places an order exceeding a certain dollar amount, or you could notify the dispatchers in a service center when referrals to any repair team reach a certain number.

Note: This feature is available on SQL Server systems only.

There are four steps to setting up the system:

1. Create an e-mail user to send the notifications. See page 360.
2. Subclass `versata.vls.DataObject`. See page 360.
3. Write the method that sends the mail. See page 361.
4. Define the action rule that sends the mail messages. See page 362.

Creating the e-mail user

These steps provide an example procedure you can use to set up an e-mail user account that will send notifications from an application, if you are using Microsoft Exchange as your mail server.

Note that the Versata Logic Server uses MAPI so that Exchange Server is not required. You just need to check with your system administrator to obtain the name of the mail server. You may use MS Mail instead of Exchange. The procedure is substantially the same except that MS Mail uses mailboxes instead of profiles.

1. On the application computer, install an Microsoft Exchange mail client for the user.
2. In the Mail & Fax control panel, create a profile for the user.
3. On the SQL Server computer, use the SQL Server Enterprise Manager to select the repository database and add the same user. Grant the user system administrator permissions.
4. Use the Services control panel to make the user the SQL Server start-up account.
5. Use the Enterprise Manager to select the user's Microsoft Exchange profile for SQL Mail (Server → SQL Mail → Configure).
6. Use the Server Manager window in the Enterprise Manager to grant `xp_cmdshell` permission to the recipients of the notifications.
7. In the sender-user's MS Exchange address book, set up convenient mail groups and aliases for the recipient-users.

Subclassing versata.vls.DataObject

Versata Logic Server data objects by default are subclasses of `versata.vls.DataObject`. If you want a data object to include methods that are not members of `versata.vls.DataObject`, you can subclass `versata.vls.DataObject`, and define your data object to be a subclass of the new class.

1. In the Versata Logic Studio Explorer, click the Files button.
2. Right-click Versata Logic Server folder and choose New File.
3. Complete the Add File wizard. In the Choose File Name dialog, enter a name for the file, using a `.java` extension. In the Create Java Class dialog, enter `versata.vls.DataObject` in the Extends field. Click the Finish button. The new file appears in the Versata Logic Studio Explorer.
4. In the Explorer, click the Objects button. Double-click the data object where you plan to define the action rule to send mail. The Transaction Logic Designer opens.
5. On the Properties:Data Access tab, record the name of the new class in the Superclass field.

Writing the method

After you have subclassed `versata.vls.DataObject`, you can add a method that implements sending mail.

1. In the Versata Logic Studio Explorer, click the Files button.
2. Right-click the new subclass file and choose Edit. The Code Editor opens.
3. Add client-side code like the following:

```
VSSession _session = VSApplicationContext.getSession();
try {
   String to = "MirG@example.com";
   String cc = "SmithS@example.com";
   String from = "SmithS@example.com";
   String mailserver = "MAIL08A";
   String subject = "testing";
   String msg = "this is a test message from SmithS";
   int transType = 0;

   _session.sendMail(to, cc, from , mailserver, subject, msg,
transType);
   System.out.println("complete");
} catch (VSException ex) {
   ex.printStackTrace();
   VSApplicationContext.handleException("Mail Error" + new
VSDate(), ex);
}
```

4. Add server-side code like the following:

```
public void sendMail (String from, String to, String subject,
String msg, int type ) {
try {
// Change Mail Server Name here!
versata.vls.SendMail sm = new versata.vls.SendMail("exchange");
String fileName = "d:\\TestFiles\\test.doc";
String[] to = new String[1];
to[0] = new String("PasskeyRepos@example.com");
sm.setRecipientsTo(to);
sm.setFrom("PKREPOSIT@EPEnergy.com");
sm.setMsgText("Testing from SmithS");
sm.setSubject("AppServer User-Role Results");
sm.attachFile(fileName);
sm.send(0);
} catch (SendMailException e) {
e.printStackTrace();
System.out.println(e);
}
```

5. Choose File → Save File.

Defining the action rule

The last step is to define an action rule that calls the `sendMail` method when the defined condition is met.

1. In the Versata Logic Studio Explorer, double-click the data object where you plan to define the action rule. The Transaction Logic Designer opens.
2. Click the Actions tab.
3. Choose Edit → Add Action.
4. Enter a name for the action.
5. Define a conditional expression to indicate when the mail will be sent.
6. In the Action/Method Call field, enter the name of the method to be executed. Include the attributes to be passed.
7. Choose File → Save Transaction Logic.

Note: The sample repository includes an example of this type of email notification system, with an extra level of complexity. The repository includes a `MailEnabledDataObject` class that is a subclass of `versata.vls.DataObject`. The ORDERS data object uses `MailEnabledDataObject` as a superclass. `MailEnabledDataObject` includes a method called `sendMail`. The ORDERS data object includes a method called `sendBigOrderMail` that calls `sendMail`. The `sendBigOrderMail` method is called from the ORDERS.bigOrder action rule. The `sendBigOrderMail` method provides an extra level of complexity because it is totally declarative.

SQL expression evaluator

The Versata Logic Server's Transaction Logic Engine now includes a SQL expression evaluator. The purpose of the SQL expression evaluator is to perform the following functions:

- Support SQL `Where` clause evaluation for reconciling cached objects with the database rows if a SQL `Where` clause is used in the filter (`SearchRequest`)
- Support SQL `Where` clause evaluation as a general purpose functionality which can be used in client, server, or Versata Connector code
- Build a SQL parser that can be used to build other VLS functionality
- Enhance the `VSResultSet.findFirst` method

The following sample code is provided as part of the SQL expression evaluator feature:

- Code that demonstrates the use of the SQL expression evaluator as a general-purpose functionality. See "General SQL evaluator example" on page 371.
- `VSRowProvider` implementation that demonstrates the use of client-side filtering. See "Client-side filtering example" on page 371.

SQL parser

The Transaction Logic Engine has had an embedded SQL parser that can parse complete statements such as `Select`, `Delete`, `Insert`, and `Transaction`. However, the engine previously used only the SQL expression parser for the `Where` clause and then started parsing the expression string directly. The embedded SQL Parser also supports quoted identifiers.

The grammar for the SQL parser is taken from Oracle's grammar documentation with some changes. The parser was built using the `JavaCC` (Java compiler compiler) utility developed by Sun Microsystems. The grammar is compiled using `JavaCC` version 2.0, and the package name for all classes is `versata.common.sql.parser`.

The following enhancements have been made to the SQL grammar to incorporate Versata functionality.

- Support of Boolean constants (`true`, `false`)
- Support of some known functions.
  - `APPUSER()`, `USER()`, `DBUSER()`
  - `DATE()` (or `CURDATE()`, `CURRENT_DATE()`), `TIME()` (or `CURRENT_TIME`), `DATETIME()` (or `CURRENT_TIMESTAMP`)
  - `TO_DATE(StringValue, format)`, `Cdate(StringValue)`
  - `LOWER` (or `LCASE`), `UPPER` (or `UCASE`), `LENGTH`, `LTRIM`, `RTRIM`, `TRIM`

Note: Currently the parser does not support some of the ANSI SQL database functions, such as `SUBSTRING`, `CONCAT`, and `ABS`.

Parse tree data structure

The data structure for the parse tree consists of two main interfaces: `SQLStatement` and `SQLExp`, along with some other classes.

SQLStatement interface

All SQL statements such as `Select`, `Insert`, and `Delete` implement this interface.

SQL Statement	Java Class
`Select`	`SQLQuery`
`Delete`	`DeleteStmt`
`Insert`	`InsertStmt`
`Transaction`	`TransactStmt`
`Update`	`UpdateStmt`

SQLExp interface

This interface is implemented by all SQL expressions. Following are the classes that implement this interface.

Class	Description	Type (if applicable)
`SQLConstant`	Represents SQL constants. The `getValue` method of the `SQLConstant` object returns a string, number, or Boolean object.	`COLUMNNAME`: Column name. `NUMBER`: Numeric constant `STRING`: String constant `BOOLEAN`: Boolean constant `NULL`: null value
`SQLFunction`	Represents SQL function. This object consists of two variables: function name and parameter list (Vector). This structure is not created for aggregate functions. Aggregate functions are created as operators in the `SQLExpression` object.	

Class	Description	Type (if applicable)
`SQLExpression`	Represents expression that consists of operator and one or more operands: one operand in case of unary operator, two operands in case of Boolean and arithmetic operator, multiple operands in case of `IN` clause.	
`SQLQuery`	SQL query can also be part of expression, as in the case of a subquery.	

Other classes

Class	SQL Clause
`SelectItem`	Items in `select` statement `*`, `Count(*)` or aliased names
`FromItem`	Items in `from` clause, `AliasedNames`
`GroupBy`	`Group By` clause
`OrderBy`	`Order By` clause
`AliasedName`	Table alias or column alias

SqlParser class

This is the main parser class, which has instance methods such as `readStatement`, `readStatements`, and `readExpression` for parsing, and static methods such as `parseExpression`, `removeExpression`, and `clearExpressionCache` for parsing as well as caching the expression.

It is recommended that custom code does not instantiate this class directly. Instead use the static method `parseExpression` to parse the expression. This method caches the expression so that the same `Where` clause is not parsed multiple times.

SQLParser instance methods

Method	Description
`readStatement()`	Parses a SQL statement
`readStatements()`	Parses multiple SQL statements separated with ";"
`readExpession()`	Parses a SQL `Where` clause
Constructor	Takes either a `Reader` or `InputStream` representing character stream to be parsed

SQLParser static methods

Method	Description
`parseExpression(String whereClause)`	Parses the expression as well as caches the expression (`SQLExp`)
`ClearExpresssionCache`	Clears the cached expression list

SQLEval class

This class implements the evaluation method `eval` that takes `Tuple` and `SQLExp` as arguments and returns a Boolean value as the result.

```
public boolean eval(Tuple tuple, SQLExp exp) throws SQLException;
```

Tuple interface

Aa special interface called `Tuple` provides a general-purpose Boolean condition evaluator. This interface is very simple and can be implemented on top of any data such as array, vector, `VSRow`, `DataRow`. The following are the methods in this interface:

- `public Object getValue(String columnName) throws UnknownColumnName;`
- `public int getType(String columnName);`
- `public boolean isDefined(String columnName);`

The DataRow.DataRowTuple and VSRowTuple classes implement this interface to return data from DataRow and VSRow, respectively. You can call the DataRow.getTuple or VSRow.getTuple method to get the instance of Tuple.

```
evaluator.eval(row.getTuple(), whereClause);
```

Multiple eval methods

In addition to the eval methods described earlier, the SQLEval class also supports variations of eval methods, so that the custom code need not write the conversion from the Where clause to the SQLExp.

```
public boolean eval(Tuple tuple, String whereClause);
```

SQLEval constructor

The SQLEval class can be instantiated using the default constructor (for instance, the constructor without any argument) if the evaluation expression does not use any database-specific functions or date functions. If the expression contains database-specific functions or date functions, the database type must be passed through the constructor.

```
/**
* @paramdbTypeConstant representing database type. Constant
values are defined in DataConst class e.g. DataConst.ORACLE.
*/
public SQLEval(int dbType);
```

SQLEval.setProperty method

The SQLEval class supports the setProperty method to provide values for some globals (values not dependent on a row or tuple) such as user name or database user name. If the expression contains a function such as User(), AppUser(), or DBUser(), then the two properties SQLEval.USER and SQLEval.DBUSER must be set, as follows:

```
evaluator.setProperty(SQLEval.USER, “Guest”);
evaluator.setProperty(SQLEval.USER, session.getUserID());
evaluator.setProperty(SQLEval.DBUSER, “sa”);
evaluator.setProperty(SQLEval.DBUSER, con.getID());
```

In the future, more properties may be added to support additional global functions such as `rowCount`.

Subclassing the SQLEval class

You can subclass the `SQLEval` class to enhance the evaluator. To provide a custom evaluator, set the `evaluator` variable in the `SearchRequest` object.

```
SearchRequest filter = new SearchRequest();
filter.add(SearchRequest.STRING,
“(a > b and d < 2*e) or (s1 like ‘N*’)”);
filter.evaluator = new CustomEvaluator();
```

Understanding SQL expression evaluations

Boolean expressions

The SQL expression evaluator evaluates Boolean expressions as used in `Where` clauses. This is implemented by `SQLEval` and supports most of the SQL operators and expression with the following limitations.

- There is no support for subquery. `SQLException` will be thrown if it encounters expression of type `SQLQuery`.
- No support for aggregate functions such as `Max`, `Avg`, `Sum`, `Count`, `Min`. `SQLException` will be thrown if evaluator encounters aggregate function as one of the operators in `SQLExpression`.
- `LIKE` operator is not supported completely. Wild card expressions can contain only the following types of patterns.

  “prefix%suffix”, “%suffix“, “prefix%”, “%mid%”.

Numeric expressions

Numeric expressions are evaluated using double as a common data type. This evaluation is done to take advantage of Java native data types so that all arithmetic operators are applied using native operators. This evaluation also provides simpler and efficient conversion because all numbers (Java `Number` class) provide conversion to double value. It will be enhanced to support `BigDecimal` as an option. `BigDecimal` is required to evaluate expressions that require larger precision and scale, for example, ColumnX = 22/7.

Date constants

Date constants are represented using different formats on different databases. For example, consider the date “15 November 2000”. The following table illustrates ways of representing this date in different databases.

Database Type	Date Representation
Oracle	to_date(‘2000-11-15’, ‘yyyy-mm-dd’)
DB2	‘2000-11-15’
SQL Server (US*) * SQL Server default format is different for different locales	“November 15 2000 00:00”
Microsoft Access	cdate(‘2000-11-15’)
Informix	datetime(2000-11-15) YEAR TO DAY

Note: Currently the SQL expression evaluator supports only Oracle, DB2, and Microsoft Access formats.

Time constants and timestamp expressions

The `TIME()` and `DATETIME()` functions currently are not evaluated, because they may return different values when executed on the database server versus when executed in the Versata Logic Server.

Run-time changes required to use the SQL evaluator

- The Transaction Logic Engine has been enhanced to use the SQL expression evaluator for reconciling cached objects and rows returned by database queries. This is done by enhancing the `matchesFilter` method of the `DataObject` class. Since the `SqlParser` does not support all database functions it will throw a `ParseException` if the expression uses such functions. This exception is ignored by the `matchesFilter` method, which prints a warning message in the log that custom code should override `matchesFilter` to support evaluation using custom code. If the parser is able to parse the expression but the evaluator is unable to evaluate the expression, as in the case of an aggregate function, a similar warning message is printed in the log indicating that custom code should override the `matchesFilter` method. If a `SearchRequest` contains a `Where` clause which is part of the rule processing, then expressions will not be evaluated.

- The `VSDate` class has been enhanced to support the `compareTo` method. This is required for date comparisons.
- The `SearchRequest` class has been enhanced to add two additional variables: `exp` of type `SQLExp` and `evaluator` of type `SQLEval`.

Note: Expressions are evaluated from left to right. Put the simpler expression on the left side so that complex expressions may be short-circuited some of the times. For example:
`(X > 5) AND (Name LIKE 'A%) AND LTRIM(UPPER(s1)) = 'ABC'`

SQL expression evaluator examples

General SQL evaluator example

The following code example shows how to use the SQL evaluator in custom code.

```
DataRow row = //;
SQLExp exp = SqlParser.parseExpression("a > b and c > 2 *d");
SQLEval evaluator  = new SQLEval();
boolean b = evaluator.eval(row, exp);

DataRow row = //;
SQLExp exp = SqlParser.parseException("datecolumn = Date() &&
logon = User());
SQLEval evaluator = new SQLEval(DataConst.ORACLE,
session.getID(),null);
boolean b = evaluator.eval(row, exp);
```

Client-side filtering example

This example shows how a custom implementation of `VSRowProvider` can support client-side filtering without sending a query to the database. `VSRowProvider` can also be used to do client-side sorting as shown in the help text of this interface. An instance of `VSRowProvider` can be used to create a custom result set that can then be bound to the `DataControl` (Java) or `DataSource` (HTML).

Creating a custom result set using an instance of VSRowProvider

```
import versata.common.sql.parser.*;
import versata.vfc.*;
public class ClientRowProvider implements VSRowProvider
{
    private VSResultSet master;
    private int currentIndex = 0;
    private SQLExp exp = null;
    private SQLEval evaluator = null;
    private VSSession session = null;
    public ClientRowProvider(VSResultSet master, String
whereClause, VSSession session) {
        this.master = master;
   this.session = session;
       try {
     exp = SqlParser.parseExpression(whereClause);
    } catch(Exception ex) {
     ex.printStackTrace();
     throw new VSException(ex);
    }
   }
public boolean isReadOnly() {
        return false;
    }
    public VSMetaQuery getMetaQuery() {
        return master.getMetaQuery();
    }
    public VSMetaColumn[] getMetaColumns() {
return null;
    }
//public VSRow fetchNextRow() throws VSException;
    public boolean fetchNextRow(Object[] dataValues) throws
VSException {
currentIndex++;
```

```
 VSRow row = master.getRowAt(currentIndex);
     if (row == null)
      return false;
     try {
     while (!evaluator.eval(new VSRowTuple(row), exp)) {
      currentIndex++;
      row = master.getRowAt(currentIndex);
      if (row == null)
         return false;
     }
     } catch(Exception ex) {
      ex.printStackTrace();
      throw new VSException(ex);
     }
     for (int i = 0; i < dataValues.length; i++) {
             //fill data values
             dataValues[i] = row.getData(i+1).getObject();
         }
         return true;
}
public int getRowCount() {
         return -1;
    }
public void close() {
         //free the memory
         master = null;
     exp = null;
    }
public VSSession getSession() {
         return session;
    }
}
```

Binding ClientRowProvider to VSDataControl(Java)

```
VSResultSet masterRs = //resultset cached when form is loaded.
VSSession session = VSApplicationContext.getSession();
String whereClause = VSTextField1.getText();
VSResultSet rs = new VSResultSetInternal(masterRs, whereClause,
session);
VSDataControl dataControl = datT1<<TableName>>;
DataControl.setResultSet(rs);
```

Binding ClientRowProvider to DataSource (HTML)

```
VSResultSet masterRs = //resultset cached when page is loaded.
VSSession session = getPaentApp().getSession();
String whereClause =. . .
VSResultSet rs = new VSResultSetInternal(masterRs, whereClause,
session);
DataSource ds = datT1<<TableName>>;
ds.setResultSet(rs);
```

Working with Versata Logic Server security properties

Security for the Versata Logic Server is managed in the Versata Logic Server Console, where you can set up security properties declaratively, create extended security properties, and customize the security model to use a security manager other than the default Versata Logic Server security for some tasks. For production systems built with this release, you most likely will want to use IBM WebSphere Application Server security. You also can choose to use LDAP and JNDI. For information on setting up custom security managers, see the *Administrator Guide*.

Under the default security manager, all security enforcement is handled in the Versata Logic Server, which connects to files used for security. Applications usually do not have direct access to these security files. This section describes some of the methods available to write Versata Logic Server security code and provides examples of custom security code.

Versata Logic Server security APIs

Versata Logic Server provides some security APIs, which you can use to customize security so that some aspects are managed programmatically. These APIs are provided as methods of a number of special security classes in the `versata.vls.*` package, or as methods of the `versata.vls.Session` interface. The methods retrieve security properties so you can reference them in custom code. The methods include the following.

- The following method can be called to return security information for a user, including the user's role name and ID:
  - `Session.AppImpl.getUserRolePropertiesForUser(String userLoginID)`
- The following method can be called to return security privileges information for an object, including a role name and ID for each privilege:
  - `Session.getObjectPrivilegePropertiesForObject(String objectName, String objectType)`

- The following methods can be called to return security information for the specified object(s):
  - `Session.getUserProperties(String userLoginID)`
  - `Session.getRoleProperties(String roleName)`
  - `Session.getObjectProperties(String objectName, String objectType)`
  - `Session.getUserRoleProperties(String userLoginID, String roleName)`
  - `Session.getObjectPrivilegeProperties(String objectName, String objectType, String roleName)`
- The following methods can be called after you have called a factory method such as `getObjects`, to retrieve one or more objects.
  - `AppUserImpl.getProperties()`
  - `AppRoleImpl.getProperties()`
  - `AppObjectImpl.getProperties()`
  - `AppUserRoleImpl.getProperties()`
  - `AppObjectPrivilegeImpl.getProperties()`
  - `Session.getUserProperties`

Writing custom security applications

The security APIs provided with the Versata Logic Server allow you to build custom security applications for targeted requirements. These APIs are not designed to allow a complete rewrite of the Versata Logic Server Console, so they cannot reproduce all of its functionality. For more information about implementing custom security and examples, see the *Administrator Guide*.

Working with JTS transaction management

Transaction management is generally handled by APIs from Versata Logic Server classes. Another alternative is available for transaction management for EJB business objects.

- For business objects that use data stored in Oracle or DB2 Universal Database, transactions can be managed using IBM's implementation of JTS, which is a wrapper around the OTS implementation of Encina Transarc.
- Transactions can be started either with the Java Transaction API (JTA) or with the API in `VSSession` or `VLSContext`. The `Usejts` server property, which is set in the VLS Console, determines whether transactions are processed using JTS or the Transaction Logic Engine.

Note: If you use external connection pooling, you also need to use JTS (Java Transaction Service) to process transactions in order to take advantage of the two-phase commit provided by the underlying EJB server. As a result, the Transaction Logic Engine starts a JTS transaction and the commit comes from JTS.

- If developers use the JTA interface to demarcate a transaction boundary, the transaction is propagated automatically to the server. Developers do not have to write any code to register the `VLSContext` with the transaction. When a method such as `save` is called in the `VLSContext`, the server automatically checks whether a JTS transaction is in progress. If a JTS transaction is in progress, the server initializes transaction information with the JTS transaction instead of starting a Versata Logic Server transaction.
- The Versata Logic Server business objects' internal code still uses the `VLSContext` APIs (`begin`, `commit`, and `rollback`) to start and commit transactions.
- The `VLSContext` bean's transaction attribute is `TX_SUPPORTS`, so `VLSContext` methods can be called with JTS transactions.
- Data objects deployed as entity beans may have different transaction attributes for different methods. If the transaction attribute of a method is `TX_MANDATORY`, its transactions must be started with the JTA interface. If the transaction attribute of a method is `TX_REQUIRED`, the application server automatically starts a JTS transaction for it. The transaction attribute of the `save` method is `TX_REQUIRED`, while the transaction attribute of the initialization method is `TX_MANDATORY`.
- When a method is called in an entity bean, an instance of `VLSContext` is created if the transaction does not have a context associated with it. This context is destroyed as soon as the transaction is completed.

Note: Do not add code that calls methods on multiple instances of `VLSContext` within one transaction.

Suppressing creation of abstract methods

Whenever you add a custom method to an object's implementation file, the Versata Logic Studio automatically creates a corresponding abstract method in the object's base implementation file. If the purpose of the custom method is to override a method in the base implementation superclass, the addition of the abstract method defeats this purpose.

To provide a solution for this problem, Versata business objects provide the capability of suppressing creation of an abstract method in the base implementation file. To prevent creation of an abstract method in an object's base implementation file, place the keyword `@SuppressAbstract` in the line immediately following the method declaration in the object's implementation file. The following code provides examples of method syntax containing this keyword:

```
public void NewMethod()
//@SuppressAbstract
{
        ...
}
```

```
public void NewMethod()
{//@SuppressAbstract
        ...
}
```

Handling Java quotes inside Versata Logic Server code strings

If you are adding code containing Java quotes to a Versata Logic Server file, you need to mark each quotation mark with a backslash (\), as in the following example:

```
createTransferCredit()
insertObjects('this', 'AccountTransactionImpl',
'setTransAccountNumber(this.getTransferAccount() );
setTransCategory(\"A\"); setTransType(\"C\");
setTransAmount(this.getTransAmount() );
setTransferAccount(this.getTransAccount() ) ' )
```

If you do not include backslashes, you will receive errors.

CHAPTER 11

Working with Versata Connectors

Chapter overview

Read this chapter for an introduction to the data access code generated for Versata Logic Server data objects to connect to supported RDBMSs. After you read this chapter, you should have a basic understanding of this data access code and some general ideas of the requirements for custom data access code to non-supported data sources.

This chapter includes the following:

- "eXtensible Data Access (XDA)" on page 383, describes the architecture for Versata Logic Server data objects' access to data sources.
- "Understanding Versata Connectors" on page 384, describes the code generated for access to supported RDBMSs, which is packaged as objects called Connectors, including instantiation, classes and methods used, retrieval processing, and save processing.
- "Associating Connectors with data objects" on page 389, explains how to specify that a custom Connector should be used for a data object's data access, and how to set up data access information in the Versata Logic Server Console.
- "Creating custom Versata Connectors" on page 391, outlines the steps required to write your own custom Connectors.

eXtensible Data Access (XDA)

Versata Logic Studio-generated applications access data through the Versata Logic Server, which transforms data from different sources into a seamless and transparent set of data that the client understands. The Versata Logic Server does this by abstracting data access behavior from higher levels of application behavior and defining it in a small set of APIs that retrieve, filter, and save row data. Using these APIs, the rest of the system in all the tiers creates display and calculation functionality that is independent of data access. This unique way of making data accessible from any data source is called “eXtensible Data Access technology”, or XDA for short.

Versata Logic Suite’s XDA framework provides the interface between business objects on the Versata Logic Server and the databases, applications, or middleware that supply the physical data against which business rules are run. This framework consists of well-defined, generic Java methods for querying, fetching, updating, and saving data on any type of data source: relational, object, application, or middleware.

A key benefit of XDA is that it enables the integration of relational data with package, legacy and other non-RDBMS types of data. One repository can contain data objects that map to all these different types of data sources, with seamless enforcement of business rules, because rules logic processing is separated from the physical storage of data.

Understanding Versata Connectors

The methods required for data source connectivity are included in objects called Versata Connectors. A Versata Connector is a data access mechanism that is the interface between the data controls on a Versata Logic Studio-generated form or the elements bound to data sources on a Versata Logic Studio-generated page, the data object(s) in the Versata Logic Server, and the database. Versata Connectors receive requests from data objects and pass them to the database in the native syntax of the database. They also return the results to the data object so that it can pass the changes on to the data control in the Versata Logic Studio-generated form or to the data source on the Versata Logic Studio-generated page. The Connectors' code integrates with JDBC interfaces.

RDBMS-specific APIs, such as APIs for handling query definitions and `Where` clauses in Versata Logic Studio-generated applications, are managed in the Connectors. (Query definitions and `Where` clauses are implemented as SQL statements and passed to the database.) Error handling is provided in the Connectors, and there is a special SQL service that maps data type differences between the RDBMSs.

The Versata Logic Suite includes classes that can be implemented and extended to create Connectors to supported RDBMSs, including Oracle®, Microsoft SQL Server, Sybase®, Informix®, and DB2® UDB. Some Connectors for other data sources are available for separate purchase.

If you require connectivity to another type of data source, you can write your own custom Versata Connector code. Versata Logic Suite provides an interface file that you can implement and a class file that you can extend to create Connectors. Custom Connectors may be SQL connectors that extend or replace the behaviors of the supplied connectors, or non-SQL connectors supporting APIs such as CORBA or SAP.

Instantiating Connectors

Each data object in a repository creates and uses a Versata Connector to connect to, get, and persist data. For data objects that map to supported RDBMS tables, the Java implementation file for each data object includes a method that generates a Versata Connector to provide connectivity to any supported RDBMS. Data objects that map to data sources other than supported RDBMSs or require other special processing need to use custom Versata Connectors.

Data objects that use custom Connectors are first class objects, with the same business object code as data objects using standard Connectors. The only difference is in the Versata Connector code. Thus, you can define rules in data objects that use custom Connectors, and all Versata Logic Server services, including "just-in-time" object instantiation, automatic partitioning, and optimized rules processing, are available to these data objects. Also, custom Versata Connector-based objects are fully automated for Versata Logic Studio-generated application construction, meaning they can be used as sources of data to be displayed.

The following code provides an example of the method used to create a standard Versata Connector for a data object that maps to a supported RDBMS table.

```
/**
* <br>
* a factory method to create the XDAConnector object for this
class.
* @return XDAConnector : if succcessful returns an instance
* of the XDA Connector.
*/
   public static XDAConnector createXDAConnector()
   {
      XDAConnector xda = null;
      try {
      VSMetaTable table = getMetaQuery().getChildMostTable();
      if ( table != null )
         xda = ( XDAConnector)Class.forName
         ( table.getXDAConnectorClassName() ).newInstance();
      else
         xda = ( XDAConnector)Class.forName
         ( "versata.vls.XDASQLConnector").newInstance();

   }
   catch ( Exception ex )
   {
      ex.printStackTrace();
   }

   return xda;
      }
```

Connector classes and methods

Versata Logic Suite provides an interface definition for XDA, `versata.vls.XDAConnector`.

- The `versata.vls.XDAConnectorImpl` class provides a base implementation of this interface's methods. This class includes only those APIs used for internal system management.
- The `versata.vls.XDASQLConnector` class is a fully functional subclass of `XDAConnectorImpl` that provides connectivity to SQL data sources. This class provides a variety of services, including RDBMS-specific APIs, query definitions and `Where` clauses implemented as RDBMS-specific SQL statements and passed to the database, special SQL services that map data type differences between RDBMSs, and error handling.

The default Connectors are instantiated from the `XDASQLConnector` class. The `XDASQLConnector` class can be used as a base to create custom SQL Versata Connectors. The `XDAConnectorImpl` typically is used as a base to create custom non-SQL Versata Connectors.

If you would like to review the methods available in these classes, you can open the Enterprise Object Browser, select Versata VLS Classes from the Object Libraries drop-down list, and select a class from the list. Be sure to enable the display of private methods. You can find information about methods in the `vfc.hlp` file in the `Help` subdirectory.

The `versata.vls.XDAConnector` interface includes a number of methods that perform standard data access operations, including methods which perform queries, retrieval methods, and methods for saving. These methods include the following:

- `execute`, which causes a query to be performed
- `fetch`, `getDataArray`, `getObjectArray`
- `save`, which includes `insert`, `update`, and `delete`
- `getRowCount`, `getRowSum`
- `refresh`
- `synchronizeDataSource`

This interface also includes a `createConnection` method, which creates an object of the `versata.vls.Connection` class, and a `setProperties` method, which stores data source connection properties used to make connections.

The Versata Connector is an execution channel. It is stateless except when retrieval is in progress and is not reused across objects. Versata Connectors can be designed to interact with a data source directly, but in more sophisticated systems the communication link between the Versata Connector and data source is maintained by another important interface class: `Connection`. This is the framework for standard Versata Connectors for supported RDBMSs.

The `Connection` class provides methods for the following functionalities:

- Transaction control: If a business object is transaction-enabled, as are most RDBMS objects, its data source session needs to be part of the transaction management. The `Connection` class can be used to wrap around a physical link and can be registered in Versata Logic Studio.

- Connection pooling: Participation in connection pooling optimizes performance.

The `Connection` is a limited resource, which is expensive to create and destroy. It has a state, either active or idle, and is reused among sessions and objects.

Retrieval processing

This section outlines the methods executed to retrieve data using a Versata Connector in the XDA framework.

First, a query is inititiated. A client may initiate a query, for example, either through a data control (in a Versata Logic Studio-generated Java application) or data source (in a Versata Logic Studio-generated HTML application) or programmatically. The `MetaQuery` provides the definition of the query, including the data object or query object and attributes involved. The Versata Logic Server locates the data object, or the childmost data object for the query object, and loads the Versata Connector associated with the data object. All properties defined for the Versata Connector in the Versata Logic Server Console (data server properties) are passed as arguments.

The Versata Logic Server allocates a connection to the persistent data source for the data object. First the Versata Logic Server searches the local pool for a login session, then it searches the global pool for a matching data server and connection ID. The connection ID is customizable; the default is the database login. If no existing connection is found, the `createConnection` method from `XDAConnector` is called to create one. The `Session` holds on to the connection pool information.

The Versata Connector's `execute` method should return true if the operation is successful, regardless of whether any records match the query criteria. The `execute` method is passed the `ResultSet`, `MetaQuery`, `Filter`, `SortRequest`, and `Connection` objects. A successful query will open a database cursor at this point. The Versata Connector should maintain the position of the data buffer and return data in string format.

The Versata Logic Server calls `fetch` if the query includes virtual attributes, otherwise it calls `getDataArray`. The fetch call returns a `dataRow` in the `ResultSet`. If the query contains any virtual attributes, `fetch` is invoked to construct a `dataRow`. The server object is created using values from the row and the caller then populates all virtual attributes. The `getDataArray` method is used to fetch a fixed number of rows from the data source in a chunk. The rows of data are populated into a preallocated two-dimensional array which is passed to the method. The return value is the number of rows populated. This number can be smaller than the number of rows in the preallocated array, if the number of available rows is smaller.

When all data retrieval is complete, the Versata Connector is asked to release all resources. The Versata Connector should clean up resources (for example, close the database cursor) and should inform the associated `Connection` object that the query is over.

Save processing

This section outlines the methods executed to save data using a Versata Connector in the XDA framework.

First, a request is initiated. For example, a client initiates a request to save one or more records. This starts a transaction. Transaction control tries to allocate an available connection from the pool; it may require a different type of connection than a query request. For example, the `save` may require an exclusive connection. A database transaction is started.

The business object is instantiated and it locks itself on a database, so the database cannot be altered until `commit` occurs. Rules code is run against the business object and events are fired at the appropriate times. The business object is cached in the transaction control buffer.

Once all business object code has been processed without errors or exceptions raised, transaction control flushes all changes into the persistent data source. The Versata Connector determines what action is required (`insert`, `update`, or `delete`) by inspecting the `dataRow`'s status and this action is performed in the process of the save. The Versata Connector's `execute` method should return true if the operation is successful, regardless of the number of records saved. Once the action is complete, the connection is released from the transaction and put back in the connection pool if it is sharable.

The final `refresh` is used only if necessary to synchronize a query object with the data source. You can enforce refresh by setting a data server property.

Associating Connectors with data objects

Defining Connectors for data objects

You define the Versata Connector class to be used for a data object on the Properties:Data Access tab of the Transaction Logic Designer. If the data object maps to a supported RDBMS and no special handling is required, the default SQL Connector is used. If the data object maps to another type of data source or requires special handling, and uses a custom Connector, you need to enter the name of the custom Versata Connector class.

Setting up Connectors in the Versata Logic Server Console

Each data source for a Versata Logic Studio-generated application maps to a data server. A data server is a collection of properties describing connection and location information for the data source. Data servers are exposed in the Versata Logic Server Console. Each business object is assigned to a data server. This assignment is initialized when you first deploy business objects to the Versata Logic Server, and uses information from the most recent data model deployment.

Data servers are categorized by data server type. Each data server type corresponds to a JDBC API used for data source connectivity, typically a JDBC driver. Data server types for supported RDBMSs are preset in the Versata Logic Server Console. Each data server type has a set of associated connection properties that hold declarative data access information. Examples of connection properties include default user, DSN (data source name), schema name, and port. The values for these properties are used by Versata Connectors to establish connections with the physical data sources.

If you use Versata Logic Studio's Deployment Manager to automate deployment of data objects to the RDBMS database and to the Versata Logic Server, values for connection properties for these data objects' data server(s) are set automatically, based on the default data server type for the RDBMS where data objects were deployed.You can review and modify this information in the Versata Logic Server Console.

For data objects that use custom Connectors, you need to define a new data server type corresponding to a JDBC API for the physical data source, define connection properties for the data server type, and set up values for these properties manually. You can complete these tasks in the Versata Logic Server Console. The properties required for a data server type depend on the code in the Versata Connector. For information about defining data server types and connection properties, see the *Administrator Guide*.

Note: The `XDAConnector` interface has a `setProperties` method. You can use this method to store data server connection properties for a data source using a custom Versata Connector. For more information about this interface and its methods, see the `vfc.hlp` file in the `Help` subdirectory.

Creating custom Versata Connectors

You may want to write your own Versata Connector classes for the following reasons:

- To access data which is not accessible by the default Versata Connectors. For example, to access data in a legacy database or even data in memory, such as data from the Windows registry.
- To augment the queries that the default Versata Connectors send to the supported RDBMSs. For example, you might want to write a Versata Connector to use stored procedures to retrieve data from a Sybase or Microsoft SQL Server database. Many users of Sybase and Microsoft SQL Server prefer to use stored procedures to return data, since these can provide superior performance by reusing query optimization plans. The `Server_XDA_StoredProcedure` sample application illustrates how to write a Versata Connector that uses stored procedures for data retrieval.
- To access a table in a supported database that is not associated with the current database, a database of another type (such as an Excel spreadsheet or a legacy database), or any other data source for which the default Versata Connector is inappropriate. You might even use a Versata Connector to create non-default data access behaviors on the current server and database.

In order to design a custom Versata Connector, you need a good understanding of:

- the `XDAConnector` interface
- the target data source and middleware, and their APIs
- the Versata Logic Server's data object and `dataRow` behavior
- `Session`, `Connection`, `ResultSet`, and `MetaQuery` objects

The following tips may help you in getting started with design of a custom Versata Connector:

- Begin by building for a single instance of a data object. You may be able to generalize the design to provide access to all data objects, or you may need to adapt the initial design to create separate Connectors per data object. For example, separate Connectors may be required when the API interface differs for each operation, as in CICS and SAP.
- Defer building filtering (but make sure hooks are available).
- Begin by supporting two standard data types, such as string and integer; consider deferring support of other data types.

Adding a Versata Connector file to a repository

To create a Versata Connector, you need to create a Java source file for it. You may do this in three ways, as shown in the following table.

Method	Procedure	Comments
Use the New XDA Connector wizard to start with a basic code structure and add other code to the file in a Code Editor.	On the Files tab of the Versata Logic Studio Explorer, right-click the Versata Logic Server folder or one of its subgroup folders, and choose New XDA Connector to start the wizard. The Choose File Name dialog opens. Enter a name and click the Next button. The Create XDA Connector Class dialog displays defaults for the package that contains the class, any interfaces it implements and any classes it extends. Edit these if necessary. After you enter information about the class, the file opens in the Code Editor, with basic code already in it.	Writes the source file for the class in the Code Editor and compiles and registers it. You may extend a default Versata Connector class or write an original class. Versata Logic Suite saves these source files within `\<repository>\<repository>_JavaFiles` folder, in a subgroup folder if applicable. The compiled class files are in `\<repository>\<repository>_JavaFiles\Classes`. All classes in this directory are in the repository classpath automatically.
Reference an existing file.	On the Files tab of the Versata Logic Studio Explorer, right-click the Versata Logic Server folder or one of its subgroup folders, and choose Add Files to browse to the file.	Writes the location of the file in the repository metadata. The file must be registered before it may be referenced. Previously referenced classes and libraries may be viewed in the References dialog. To open the dialog, choose the References option of Versata Logic Server in the Versata Logic Studio main menu.
Copy an existing file into the repository.	On the Files tab of the Versata Logic Studio Explorer, right-click the Versata Logic Server folder or one of its subgroup folders, and choose Add Copies to browse to the file.	Writes the location of the file in the repository metadata and copies the file into the appropriate subdirectory of `\<repository>_JavaFiles`.

After you create the Java source file, set it up as the Versata Connector for the intended data object by right-clicking the data object on either tab of the Versata Logic Studio Explorer, and choosing Transaction Logic Designer. On the Properties: Data Access tab, select Custom and browse to the custom Versata Connector.

Writing code for a custom Versata Connector

Your custom Versata Connector should extend one of the system-provided classes. The `XDASQLConnector` class can be used as a base to create custom SQL Versata Connectors. The `XDAConnectorImpl` typically is used as a base to create custom non-SQL Versata Connectors. You may be able to find a SQL interface for a non-SQL data source, then subclass `XDASQLConnector` and override those methods that require different behavior.

The code in your custom Versata Connector should include the following:

- **`execute` method.** This method processes the multiple record retrieval logic, retain result, keyset, cursor handle, or object array structure for later fetches; and should be able to parse SQL or use parameters for filters.
- **`fetch` and `getDataArray` methods.** These methods get the next result, keyset, cursor record, or object instance, packaging it as an array of strings before returning.
- **`save` method.** This method includes `insert`, `update`, and `delete` processing. It requires the record(s) to be locked. For update, it uses a modified flag to determine which record(s) to update. This method tells the data source to change its data using its language or API.
- **`getRowCount` and `getRowSum` methods.** These methods are used for attributes derived through sum and count rules. You can implement this functionality through existing `execute` and `fetch` methods if direct calls do not exist in the data source.
- **`refresh` and `synchronizeDataSource` methods.** These methods must be able to reread all data for all record values from stored key information. If a connection is in progress, your code must lock the record in some way.
- **`Connection` methods.** You optionally can design your own `Connection` object to instantiate during `createConnection()`.

  `Connection` methods allow for transaction control and connection pooling. The `Connection.getID` and `XDAConnector.getConnectionID` methods should be implemented to return the same value in order for an existing connection to be reused by the next request that can share the same type of resource. Share the `Connection` if possible. `Connection` provides the `setSharable` and `isSharable` methods. If it is not possible to share the connection, release it as soon as the Versata Connector instance is released.

  The `Connection` object can control behavior for `beginTransaction`, `rollback`, and `commit`, and for read properties set in the Versata Logic Server Console for the data server type.

Whenever an activity gets invoked on a Versata Connector, a `Connection` object is passed to it. So the Versata Connector should be designed so it can set a flag on a `Connection` when it is being used, and can reset a flag when the task is finished. Because a `Connection` can be used by more than one Versata Connector, the Versata Connector should not directly invoke the `release`, as it cannot track whether other activities are using the `Connection` at the same time. The `Connection` itself should invoke the `release` method.

Note: If you subclass `XDASQLConnector`, your Versata Connector inherits all the APIs required by the Versata Logic Server for connection reuse.

Testing a custom Versata Connector

After you have written the code and made the Java source file available to the repository, you are ready to test your Versata Connector. To test a custom Versata Connector:

- Create a data object in the Versata repository to represent the data source.
- Deploy the data object to the Versata Logic Server.
- Set up a data server type and connection properties for it in the Versata Logic Server Console. Assign this data server type to the data server holding the just deployed data object. Then enter values for connection properties. For information, see the *Administrator Guide*.
- Build, deploy, and run a Versata Logic Studio-generated application or other client to test basic functionality.
- Verify that the implementation can support multiple users, and that it supports optimistic and/or row locking correctly.
- Test and document any limitations relating to support for multi-row update, positioned update in a list, `insert`/`update`/`delete`, query, refresh of row, and virtual attributes.
- Test rules involving multiple objects, including those representing data source types, including referential integrity, replicates, formulas, constraints, defaults, and rules on virtual attributes.
- Test exception handling.
- Check for proper resource usage and cleanup, including closing of connections, connection pooling if supported, reuse of connections, and closing of sessions.

You also may wish to perform performance testing, installation testing, and end-to-end testing.

Packaging a custom Versata Connector

You need to complete the following tasks to package a custom Versata Connector:

- Document middleware requirements and installation requirements.
- Document data server type and properties to be set in the Versata Logic Server Console.
- Build a `.zip` file containing specialized classes required for the Versata Connector.
- Document the paths of the `.zip` file(s) containing Versata Connector classes and of any classes required for the middleware. These paths will need to be added to the Versata application server command line arguments in the IBM WebSphere Administrative Console.

APPENDIX A

Transaction Logic Examples

Appendix overview

This appendix is provided to assist you in breaking down business requirements into declarative business rules. The examples of common patterns here can provide models for you to think about your own requirements.

Before you begin defining business rules, you should understand what declarative business rules are and the types of rules available. A familiarity with the order of rule processing operations is important for understanding how to extend business rules. For information about these areas, see “Understanding Transaction Logic” on page 183.

You also need to understand how to modify your data model because some business rule definitions may require data model changes, including the addition of data objects, relationships, and attributes. Details about these tasks are provided in “Developing a Data Model” on page 31.

For instructions for the tasks involved in defining business rules, see “Defining Business Rules” on page 211. For information about extending business rules in cases where declarative business rules do not fully implement requirements, see “Extending Business Object Code” on page 321.

The sample repository included with the Versata Logic Suite contains many examples of declarative rules. You can open this repository and open different business objects in the Transaction Logic Designer to review these rules and their expressions. Online help for the sample repository may provide additional explanations of these rules.

In this appendix, each example requirement is displayed in bold, and the solution for implementing the requirement with declarative business rules is described in the text following the requirement.

Calculation in parent, based on child data

For every order that includes a blue widget the freight charged is waived.

This requirement involves an Order and an OrderItem data object. The freight charge is computed in the parent data object, which is Order, while the numbers and/or names specifying the parts for the order are attributes of the OrderItem data object (a child of Order). The solution to implementing this requirement includes two declarative business rules:

- A conditional count derivation rule adding a "Number of Blue Widgets" attribute to the Order data object. The count contains a qualification expression specifying that order items be counted only where `"Part Name = Blue Widget"`.
- A condition added to the formula expression for the Order.Freight attribute, specifying `"If Number of Blue Widgets > 0 then $value = 0"`.

Note the use of `$value` for self-assignment in a formula expression ($value represents the Order.Freight attribute). Also note that counts can be used for existence checks, cardinality checks, and checks of whether an attribute value has changed.

Comparing values from sibling objects

If an order includes a total of ten or more blue widgets and red widgets, the eleventh blue widget is free.

This requirement involves the calculation of an amount attribute in the OrderItem data object, and this calculation requires the addition of two different sibling items, blue and red widgets. The solution to implementing this requirement involves three declarative business rules:

- A conditional count derivation rule adding a "Number of Blue and Red Widgets" attribute to the Order data object (which is a parent of OrderItem). The count contains a qualification expression specifying that order items be counted where `"Part Name = Blue Widget or Part Name = Red Widget"`.
- A replicate rule that copies the value for the "Number of Blue and Red Widgets" attribute in the Order data object to "Number of Blue and Red Widgets" in the OrderItem data object.
- A condition added to the formula expression for the OrderItem.Amount attribute, specifying `"If Number of Blue and Red Widgets > 10 and Part Name = Blue Widget then $value = $value - Price"`. In this expression, Price is equal to the price of the blue widget.

Constraining updates based on parent data

Only Sales Reps can place orders.

This requirement involves a Type hierarchy in the Employee data object, where each employee record has an Employee Type attribute. “Commissioned” is one possible value for the Employee Type attribute. This requirement also involves the Order data object, which is where the constraint is defined. In this example, Employee is a parent of Order, because an employee can place multiple orders, while an order can be placed by only one employee. The solution to implementing this requirement includes two declarative business rules:

- A Parent Replicate rule that copies the value for the Employee Type attribute in the Employee data object to the Order data object.
- A constraint in the Order data object, indicating `“Reject when Employee Type not equal to ‘Commissioned’”`.

This example is included in the Versata Logic Suite sample repository. More information about implementing type hierarchies in a data model is included in “Type hierarchies” on page 108.

Nesting rules

If an order has more than 100 order items, there is a 10% discount on all items less than $50.

This requirement involves the Order data object and its child, the OrderItem data object. This example requires the determination of whether an order contains more than 100 items, then the computation of the amount for all items worth less than $50, then a recalculation of the order value. The solution to implementing this requirement includes two declarative business rules, a sum and a replicate each with a nested rule:

- A sum derivation rule defining "Number of Order Items" for an order, which sums the amount ordered for each order item in the order.
- A formula rule defining a Yes/No "DiscountFlag" attribute in the Order data object, with an expression like: `"If 'Number of Order Items' > 100 then $value = True"`.
- A replicate rule that copies the value of the Order.DiscountFlag attribute to a DiscountFlag attribute in the OrderItem data object.
- A formula rule for a "Recalculated Price" attribute in the OrderItem data object, with an expression like `"If Price < $50 and DiscountFlag = True, then $value = .10 * Price"`.

Retrieving data with a user-defined method

Tax computation for an order is based on a tax rate obtained from a State_Tax_Schedule data object, which depends on the state where the order is placed and the date of order placement.

In this example, locating the correct row in the State_Tax_Schedule data object requires a complex query on date ranges, so a simple replicate from this data object to the child Order data object is not possible. The solution to implementing this requirement involves a custom business function (Java method) and a formula rule that references this method. Business functions can be used for many purposes to extend declarative rules functionality. The sample repository includes multiple examples. Data retrieval is a common use.

- You can write a Java method that retrieves the correct tax rate from State_Tax_Schedule.
- Define a formula rule for Order.Tax that references the tax rate value like a replicate.

The formula expression for Order.Tax in the Versata sample repository references a Java method called `TaxRate` that provides the functionality described above:

```
If (  Inserting  OR ( AmountItems  <>  :Old. AmountItems
   Then
      $value =  AmountItems  *
      TaxRate( getplacedByCustomer().getState(), getPlacedDate() )
End If
```

The Versata Logic Studio automatically builds a predefined set of Java methods for each data object in the repository, including the `getObject` methods. These methods can be used to retrieve data from related data objects.

In this example, the `TaxRate` method retrieves parameters from two objects: `PaidDate` from the Order data object and `State` from the Customer data object. These parameters are retrieved with `getObject` methods. These methods allow you to traverse a relationship chain to find an instance of the specified object and retrieve attributes from it. Note that corresponding `setObject` methods are also built, which can be used to set values for data in related data objects.

Overriding normal rule behavior with user-defined events

The purge operation deletes paid, shipped orders without causing an inventory adjustment.

In this example, there is a need to override default rule behavior, which normally would not allow deletion of an order that is paid and shipped. If this deletion was allowed, it would result in all parts in the order being added back to inventory incorrectly. The solution to implementing this requirement involves the creation of a user-defined purge event. The creation of this event requires the addition of a button and Java event-handling code to the user interface, as well as the addition of references to the event in two declarative business rules.

1. Create a “Purge” button on forms/pages where orders can be modified.
2. Add `actionPerformed` event code to the “Purge” button, similar to the following example code from the `Server_Extended_Rules_Mods` Versata Logic Studio-generated sample application:

```
void btnPurge_actionPerformed()
{
   VSUserDefinedEvent ude = new VSUserDefinedEvent("ORDERS.Purge",
   VSAction.EventTypeDelete);
   datT3ORDERS.setUserDefinedEvent(ude);
   datT3ORDERS.delete();
      // this will fire business rules for purge,
      //   so that Part onHand / Reorder quantities are not altered.
      //   contrast to delete, as described in Help.

}
```

3. Add a test for the Purge event to a constraint rule on the Order data object, similar to the following expression defined in a constraint for the ORDERS data object in the sample repository:

```
Reject when
Deleting  AND  isCurrentEvent( 'ORDERS.Purge') = false  AND
:Old.ShippedFlag != false  AND
:Old.OrderPaid != false
```

4. Add a test for the Purge event to the formula expression for the QtyOnHand attribute in the Part data object, like the following:

```
if Inserting then
   $value = 0
elseif (  isCurrentEvent( 'ORDERS.Purge') = false ) then
   $value = QtyOnHand - (QtyShipped - :OLD.QtyShipped) + (QtyReceived -
   :OLD.QtyReceived)
end if
```

Many different user-defined events can be set in the user interface. On the Versata Logic Server, only one event at a time is "current" for a session. The current event is essentially a global variable whose value can be tested in two ways:

- The `CurrentEvent( )` function returns the value, or name, of the current event if one is set.
- The `isCurrentEvent(java.lang.String eventName)` method returns a value of "True" if the named event is set.

Using batch programs to trigger calendar-driven rules

Note: The Process Logic Add-On provides another way to implement calendar-driven rules. For details about integrating Process Logic Add-On functionality with transaction logic rules, see the *Logic Integration Guide* included with that product.

Notify the contract administrator when a contract's expiration data has passed, if the contract is of type "Service" and has a value of more than $10,000.

In this example, there is no data update that can trigger the execution of rules. The solution to implementing this requirement can be to use a batch program for everything, or to combine the use of a batch program with declarative business rules. The data model is different in each case.

The first case, where a batch program is used exclusively requires a Contract data object, with ContractID, ExpirationDate, Type, Value, and Administrator attributes. In this case, the batch program could be run daily to obtain the current date and compare it with the ExpirationDate values for each record. In cases where the ExpirationDate matches the current date, the program could check the type and value attributes, and send emails as appropriate.

The second case, where a batch program is used in combination with declarative business rules, requires less code and thus can be easier to maintain. This solution requires a ContractHeader data object that is a parent of the Contract data object. The batch program updates ContractHeader, while this data update triggers rules in Contract, which handle the update to this data object. In this case, the ContractHeader data object includes ExpirationDate, CurrentDate, and ExpireFlag attributes. The batch program can be run daily to update the CurrentDate. Then three declarative business rules are defined to implement the requirement:

- A formula rule defining the ExpireFlag attribute in the ContractHeader data object, with an expression like: `"If CurrentDate > ExpireDate then $value = True"`.
- A maintained replicate rule copying the value for ContractHeader.ExpireFlag to an ExpireFlag attribute in the Contract data object.
- An action rule for the Contract data object, with a conditional expression like: `"If ExpireFlag = True and Type = Service and Value > 10000"`. The action for this rule would be to send email to the contract administrator.

Note that other attributes such as ExpireDate could also be replicated from ContractHeader to Contract as necessary. This type of action rule requires integration with a mail program. For information on how to reference methods for sending email in rules, see "Setting up an email notification system" on page 359.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X