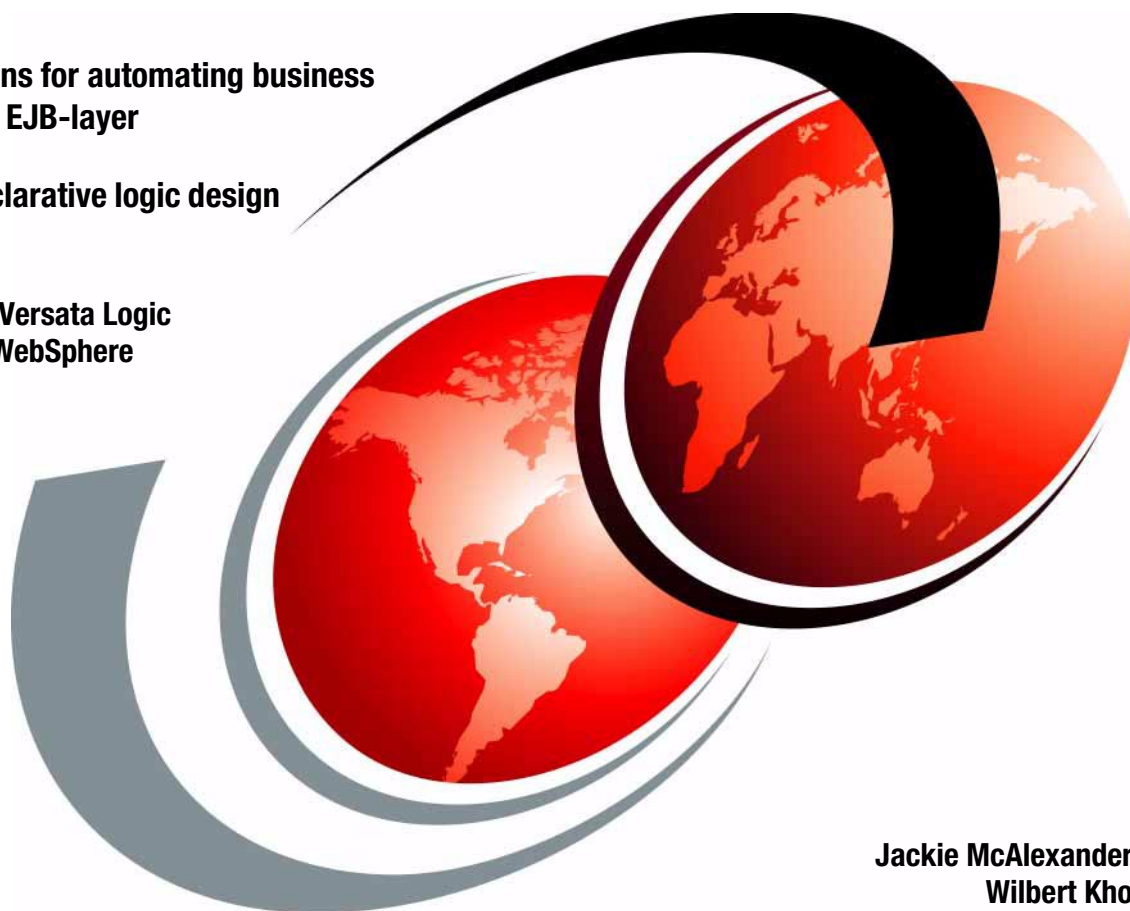


# Application Development Using the Versata Logic Suite for WebSphere

Learn options for automating business logic in the EJB-layer

Explore declarative logic design using rules

Understand Versata Logic services in WebSphere



Jackie McAlexander  
Wilbert Kho





International Technical Support Organization

**Application Development Using the Versata Logic Suite  
for WebSphere**

December 2002

**Note:** Before using this information and the product it supports, read the information in “Notices” on page vii.

**First Edition (December 2002)**

This edition applies to the Versata Logic Suite 5.5.1.

**© Copyright International Business Machines Corporation 2002. All rights reserved.**

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Notices</b> .....	vii
Trademarks .....	viii
<b>Preface</b> .....	ix
The team that wrote this redbook .....	ix
Become a published author .....	x
Comments welcome .....	xi
<b>Chapter 1. Overview of self-service pattern</b> .....	1
1.1 Patterns, defined .....	2
1.2 IBM Patterns for e-business .....	2
1.2.1 Business patterns .....	4
1.2.2 Using patterns for e-business .....	4
1.3 Self-service business pattern .....	5
1.3.1 Self-service application patterns .....	6
1.4 Self-service pattern and the Trade application .....	8
<b>Chapter 2. Trade application overview</b> .....	9
2.1 Trade application functionality .....	10
2.1.1 Trade client design using MVC .....	11
2.1.2 Multiple runtime modes .....	12
2.2 Versata: a new option .....	14
2.3 Details of the Trade EJB implementation .....	15
2.3.1 Database schema .....	15
2.3.2 Container managed EJBs .....	16
2.3.3 Use of copy helper access beans .....	17
2.3.4 Basic business logic in Trade .....	17
2.4 Potential enhancements to Trade business logic .....	18
<b>Chapter 3. Business logic automation using rules</b> .....	19
3.1 Scenarios for automating WebSphere applications .....	20
3.2 Design and runtime environment .....	21
3.3 Transaction rules for automated business logic .....	22
3.3.1 Example of a rule .....	22
3.3.2 Characteristics of rules .....	24
3.3.3 Rules and EJB domains .....	25
3.3.4 Classification of declarative logic (rules) .....	26
3.4 Business uses of rules .....	31
3.5 What the Versata Logic Suite is not .....	36

<b>Chapter 4. Architecture of the Versata Logic Server within WebSphere</b>	39
4.1 What the Versata Logic Server is, and how it works	40
4.2 Managing the Versata Logic Server in WebSphere	41
4.3 Versata business objects	43
4.4 Versata Logic Server classes	44
4.4.1 Persistence as a layer in the server MVC	45
4.4.2 Rule-enabled objects as WebSphere components	46
4.4.3 ResultSet access and just-in-time object instantiation	47
4.5 Looking to the future: EJB 2.0 and JCA	51
4.5.1 EJB 2.0: Container Managed Relationships (CMR)	52
4.5.2 EJB 2.0: local interfaces	53
4.5.3 Java Connector Architecture (JCA)	53
4.5.4 Other J2EE standards used by the logic server	53
4.5.5 Recap of the Versata logic services	54
<b>Chapter 5. Rule-based development</b>	57
5.1 Introducing Versata Studio Business Logic Designer	58
5.1.1 Project approaches and roles	58
5.1.2 Versata repository	60
5.2 Step 1: Importing an object model	60
5.3 Step 2: Adding relationship rules	62
5.4 Step 3: Identifying additional rules	64
5.5 Review of the steps	72
<b>Chapter 6. Designing an HTML client application</b>	73
6.1 Versata Presentation Designer	74
6.2 Overview of the completed application	76
6.2.1 Login page	76
6.2.2 Home page	78
6.2.3 QuoteBuy page	79
6.2.4 Portfolio page	80
6.2.5 Profile page	81
6.3 Beginning application design	82
6.3.1 Choosing the application style	82
6.3.2 Archetypes: a brief overview	83
6.3.3 Designing the Home page	84
6.3.4 Designing the QuoteBuy page	94
6.3.5 Creating the Portfolio page	101
6.3.6 Creating the Profile page	101
6.4 Completing the application design	102
<b>Chapter 7. Deploying the TradeX application</b>	103
7.1 Business object deployment	104
7.2 Database deployment	106

7.2.1	Setting up an ODBC Data Source Name (DSN) . . . . .	106
7.3	Reviewing or setting the data server . . . . .	108
7.4	Granting access to TradeX users . . . . .	112
7.5	Client application deployment . . . . .	114
7.6	Executing deployed applications . . . . .	117
7.7	Generating business and application logic reports . . . . .	118
<b>Chapter 8.</b>	<b>Enhancing TradeX business logic . . . . .</b>	<b>119</b>
8.1	New requirements . . . . .	120
8.2	The TradeXv2 repository . . . . .	120
8.2.1	Requirement 1: Sell partial holdings . . . . .	121
8.2.2	Requirement 2: Customize rules based on account type . . . . .	135
8.2.3	Requirement 3: Calculate commissions based on account type. . . . .	137
8.2.4	Requirement 4: Limit margin selling . . . . .	139
8.3	Modified client application using new business logic . . . . .	142
8.3.1	Capability 1: Creating QueryObjects . . . . .	142
8.4	The TradeXv2 application . . . . .	147
8.5	Concluding the TradeX extended business logic . . . . .	149
<b>Chapter 9.</b>	<b>Integrating the IBM Trade2 client . . . . .</b>	<b>151</b>
9.1	Method 1: Using the Versata client libraries . . . . .	152
9.1.1	The TradeAltAccess class from IBM . . . . .	152
9.1.2	Changes to TradeAltAccess to accommodate Versata . . . . .	153
9.1.3	TradeVFC.java . . . . .	154
9.1.4	The TradeVFC buy() method . . . . .	156
9.1.5	The TradeVFC getPortfolio() method . . . . .	159
9.2	Method 2: Utilizing EJB interfaces . . . . .	160
9.3	Alternative for JSP access: Versata JSP Toolkit . . . . .	163
9.3.1	Supported functionality . . . . .	164
9.3.2	Tag library overview . . . . .	165
9.4	Conclusion . . . . .	165
<b>Chapter 10.</b>	<b>Integrating Versata Logic Suite with WebSphere Studio Application Developer . . . . .</b>	<b>167</b>
10.1	Introduction . . . . .	168
10.1.1	WebSphere Studio Application Developer . . . . .	168
10.1.2	Integrated testing with WebSphere Application Server . . . . .	168
10.2	Versata Logic Server within WSAD . . . . .	169
10.2.1	Preparing Versata application for import into WSAD . . . . .	169
10.2.2	Importing applications into WSAD . . . . .	171
10.2.3	Configuring the server to test the application . . . . .	182
10.2.4	Running and debugging the application . . . . .	187
10.3	Importing modified application into Versata . . . . .	191
10.3.1	Exporting the application from WSAD . . . . .	191

10.3.2 Import the application into the repository . . . . .	191
<b>Chapter 11. Developing with UML and rules.</b> . . . . .	193
11.1 UML and the Rational Unified Process . . . . .	194
11.2 RUP phases and iterations . . . . .	194
11.2.1 Versata and the Inception phase . . . . .	194
11.2.2 Versata and the Elaboration phase. . . . .	199
11.2.3 Versata and the Construction phase. . . . .	205
11.2.4 Versata and the Transition phase . . . . .	208
11.3 Conclusion: Rule-based design and development . . . . .	209
<b>Chapter 12. A Versata Case Study: American Management Systems . .</b>	211
12.1 The technical decision process . . . . .	213
12.2 The Versata decision. . . . .	214
12.3 The project . . . . .	215
12.3.1 The team . . . . .	215
12.3.2 The application . . . . .	216
12.3.3 A specific look at performance . . . . .	217
12.3.4 System architecture . . . . .	219
12.3.5 The schedule. . . . .	221
12.3.6 Development/deployment issues . . . . .	222
12.4 The bottom line and the future . . . . .	222
<b>Appendix A. Benchmark results</b> . . . . .	225
Benchmark configuration . . . . .	226
Results . . . . .	227
Extrapolation to extended Trade2 logic . . . . .	228
<b>Appendix B. Additional material</b> . . . . .	229
Locating the Web material . . . . .	229
Using the Web material . . . . .	229
System requirements for downloading the Web material . . . . .	230
How to use the Web material . . . . .	230
<b>Related publications</b> . . . . .	231
IBM Redbooks . . . . .	231
Referenced Web sites . . . . .	231
How to get IBM Redbooks . . . . .	231
IBM Redbooks collections. . . . .	232
<b>Index</b> . . . . .	233



# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.*

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.


This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

## COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

# Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Balance®	MQSeries®	S/390®
CICS®	OS/390®	SP™
DB2®	Perform™	VisualAge®
IBM®	Redbooks™	WebSphere®
IBM eServer™	Redbooks (logo)™ 	

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:

Lotus®	Word Pro®
--------	-----------

The following terms are trademarks of other companies:

Versata, Versata Studio, Versata Interaction Server, Versata E-Business Automation System, Versata Logic Server, Versata Connections, Versata Presentation Server, Versata Adapters are trademarks of the Versata Corporation in the United states, other countries, or both.

PowerEdge is a trademark of Dell Inc. in the United states, other countries, or both.

PowerBuilder is a trademark of Sybase Inc. in the United States, other countries, or both.

TurboTax is a trademark of Intuit Inc. in the United States, other countries, or both.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Visual Basic, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

# Preface

Patterns for e-business are a group of proven, reusable assets that can help speed the process of developing applications. This IBM Redbook demonstrates a method of developing and managing the business logic in the “self-service business pattern” (formerly known as the user-to-business pattern).

The book describes the process of developing a stock trading application, based on the IBM “Trade” benchmark, using business logic rules to automate the construction and interaction of the transactional (EJB) components. It demonstrates substantially enhancing the business logic of the application through rule changes.

Two methods of constructing the presentation layer of the application are examined. The first uses Versata presentation automation techniques. The second adopts the Model-View-Controller (MVC) framework of the existing IBM Trade application.

The book demonstrates how to use the JSPs, servlets, and Java beans of the existing Trade application to interface to the EJB-based business logic and explains the role of the runtime Versata logic services installed into the WebSphere Application Server.

## The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, San Jose Center.

**Jackie McAlexander** is a Software Architect and Sr. Marketing Technical Manager at Versata, Inc. Over the last 2 years she has extensively supported and documented the integration of the Versata Logic Server and WebSphere. With over 15 years of system design experience in databases, object-oriented technology, and Java, Jackie holds a degree in Computer Science from the University of Maryland.

**Wilbert Kho** is a member of the Worldwide WebSphere Technical Sales organization, IBM Software Group. He has over 15 years of experience in the Information Technology (IT) industry, ranging from operating systems security for large scale systems to object-oriented technology and application servers. Over the last 2 years, he has been focused on WebSphere and Versata. He holds a degree in Electrical Engineering from the University of the Philippines, a Masters of Science in Computer Science degree from Northern Illinois University, and an M.B.A. from the University of Phoenix.

Thanks to the following people for their contributions to this project:

**Val Huber**, Chief Technical Officer, Versata

**Nilesh Jain**, Versata, Oakland, CA

**Jim Liddle**, Versata, United Kingdom

**Alex Rubin**, Versata, Oakland, CA

**Richard Scott**, Versata, Phoenix, AZ

**Steven Sweeting**, Versata, Oakland, CA

**Max Tardiveau**, Versata, Oakland, CA

**Joe DeCarlo**, IBM International Technical Support Organization, San Jose, CA

## Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- ▶ Use the online **Contact us** review redbook form found at:

[ibm.com/redbooks](http://ibm.com/redbooks)

- ▶ Send your comments in an Internet note to:

[redbook@us.ibm.com](mailto:redbook@us.ibm.com)

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization  
Dept. 1WLB Building 80-E2254  
650 Harry Road  
San Jose, California 95120-6099





# Overview of self-service pattern

In this chapter we introduce patterns for software development. We describe the IBM Patterns for e-business, which are a group of reusable assets that can help speed the process of developing Web-based applications. These reusable assets consist of Business patterns, Integration patterns, Composite patterns, Application and Runtime patterns, and others.

In this redbook we have chosen to focus on the Business patterns, as Versata business logic automation provides a way to create end-to-end e-business applications that match the Self-Service business pattern, Stand-Alone Single Channel, or Directly Integrated Single Channel application patterns.

## 1.1 Patterns, defined

Webster's dictionary defines pattern as a model or plan used as a guide in making things. As such, patterns serve to facilitate the development and production of things. Patterns codify the repeatable experience and knowledge of people who have performed similar tasks before. Patterns not only document common solutions to common problems but also point out pitfalls that should be avoided.

The computer industry has seen rapid advances in hardware that were driven in good measure by the use of standards and well specified components for assembly. The software development community has been searching for ways to build better and more reliable software in the quickest time possible. The adoption of similar approaches used in computer hardware development to software construction gave rise to object-oriented design and development, application frameworks, design patterns, and component-based development.

IBM has also developed a methodology for planning and designing e-business system architectures and has named this approach the IBM Patterns for e-business.

**Tip:** The IBM Patterns for e-business Web site is at:

<http://www.ibm.com/developerworks/patterns>

Visit this site for updated information and links to other resources.

## 1.2 IBM Patterns for e-business

Patterns for e-business are a group of reusable assets that can help speed the process of developing Web-based applications. These reusable assets consist of the following elements:

- ▶ **Business patterns:** These identify the interaction between users, businesses, and data. Business patterns are used to create simple, end-to-end e-business applications.
- ▶ **Integration patterns:** These connect other Business patterns together to create applications with advanced functionality. Integration patterns are used to combine Business patterns in advanced e-business applications.
- ▶ **Composite patterns:** These are combinations of Business patterns and Integration patterns that have themselves become commonly used types of e-business applications. Composite patterns are advanced e-business applications.



- ▶ **Application and Runtime patterns:** These are driven by the customer's requirements and describe the shape of applications and the supporting runtime needed to build the e-business application.
- ▶ **Product mappings:** These are used to populate the solution. The product mappings are based on proven implementations.
- ▶ **Guidelines:** These are supplied for the design, development, deployment, and management of e-business applications.

Using an approach based on the IBM Framework for e-business, the Patterns leverage the experience of IBM architects to create solutions quickly, whether for a small local business or a large multinational enterprise. As shown in Figure 1-1, customer requirements are quickly translated through the different levels of Patterns assets to identify a final solution design and product mapping appropriate for the application being developed.

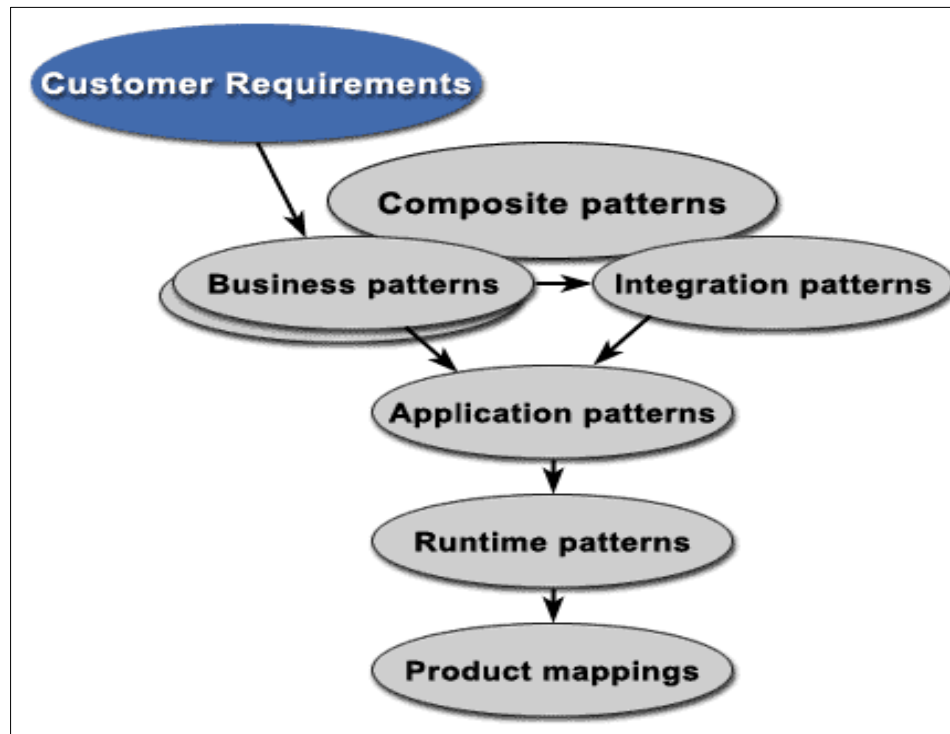


Figure 1-1 Patterns for e-business

The patterns are designed to meet 80% of most common customer requirements. If you use the patterns within a structured development methodology, you can extend their scope to meet almost all of your customer's requirements.

The Patterns for e-business are structured in a way that each level of detail builds on the last. At the highest level are Business patterns that describe the entities involved in the e-business solution. A Business pattern describes the relationship between the users, the business organization or applications, and the data to be accessed. The remainder of this chapter will focus on Business patterns, with emphasis on the Self-service pattern since the Versata business logic automation will be an implementation of this.

### 1.2.1 Business patterns

There are four primary Business patterns:

- ▶ **Self-service pattern:** This was formerly known as the User-to-Business pattern, which describes situations where users are interacting with a business application to view or update data.
- ▶ **Collaboration pattern:** This was formerly known as the User-to-User pattern, which describes the interaction between users. This would include e-mail and workflow processes.
- ▶ **Information Aggregation pattern:** This was formerly known as the User-to-Data pattern, which describes situations where users access and manipulate large amounts of data collected from multiple sources.
- ▶ **Extended Enterprise:** This was formerly known as the Business-to-Business pattern, which describes the programmatic interaction between two distinct businesses.

### 1.2.2 Using patterns for e-business

It would be very convenient if all problems fit nicely into these four slots, but reality says that things will often be more complicated. The patterns assume that all problems, when broken down into their most basic components will fit one of these patterns. When a problem describes multiple objectives that fit into multiple business patterns, the Patterns for e-business provide the solution in the form of Integration patterns.

Integration patterns allow us to tie together multiple business patterns to solve a problem. When the same combination of Business and Integration patterns has been identified in the marketplace we refer to the combination as a Composite pattern. Several common uses of Business and Integration patterns have been identified and formalized into Composite patterns.

Once the business pattern is identified, the next step is to define the high-level logical components that make up the solution and how these components interact. This is known as the Application pattern. A Business pattern will usually have multiple application patterns identified that describe the possible logical components and their interactions. For example, an Application pattern may have logical components that describe a presentation tier for interacting with users, a Web application tier, and a back-end application tier.

The Application pattern is further refined and broken down into one or more Runtime patterns. Runtime patterns define functional nodes that represent middleware functions that must be performed. The Application pattern exists as an abstract representation of high-level functions, whereas the Runtime pattern is a more concrete representation of the functions that must be performed, the network structure to be used, and the systems management features, such as load balancing and security.

Once a Runtime pattern has been identified, the next logical step is to determine the actual product and platform to use for each node. The Patterns for e-business have Runtime Product mappings that correlate to Runtime patterns, describing actual products that have been used to build an e-business solution for this situation.

Finally, guidelines assist you in creating the application using best practices that have been identified through experience.

## **1.3 Self-service business pattern**

Businesses have traditionally invested a lot of resources into making information available to customers, vendors, and employees. These resources took the form of call-centers, mailings, etc. They have also maintained information about their customers in the form of customer profiles. Updates to these profiles were usually handled over the phone or by mail.

The concept of self-service makes this puts this information at the fingertips of the customers through a user interface, whether that interface be a Web site, a PDA, or some other client interface. An e-business application makes the information accessible to the right audience in an easy-to-access manner, thus reducing the need for human interaction and increasing user satisfaction.

Key elements of an application that provides self-service for a customer would include elements that provide clear navigational directions, extended search capabilities, and useful links. A popular aspect is to provide a direct link to the online representatives that can answer questions and provide a human interface if needed.

The following are examples of self-service applications:

- ▶ An insurance company makes policy information available to users and allows them to apply for a policy online.
- ▶ A mortgage company publishes information about its loan policies and load rates online. Customers can view their current mortgage information, change their payment options, or apply for a mortgage online.
- ▶ A scientific organization makes research papers available to interested users by putting it online.
- ▶ A bank allows customers to access their accounts and pay bills online.
- ▶ A well-known and respected group of technical writers make their work available online. They recruit technical participants for their projects by listing the upcoming projects online and allowing possible participants to apply online.
- ▶ A company allows its employees to view current human resource policies online. Employees can change their medical plan, tax withholding information, stock purchase plan, etc. online without having to call the HR office.

### 1.3.1 Self-service application patterns

As you can see, the Self-Service business pattern covers a wide range of uses. Applications of this pattern can range from the very simple function of allowing users to view data built explicitly for one purpose, to taking requests from users, decomposing them into multiple requests to be sent to multiple, disparate data sources, personalizing the information, and recomposing it into a response for the user. For this reason, there are currently seven defined application patterns that fit this range of function.

1. **Stand-alone single channel pattern:** Provides for stand-alone applications that have no need for integration with existing applications or data. It assumes one delivery channel, most likely a Web client, although it could be something else. It consists of a presentation tier that handles all aspects of the user interface, and an application tier that contains the business logic to access data from a local database. The communication between the two tiers is synchronous. The presentation tier passes a request from the user to the business logic in the Web application tier. The request is handled and a response sent back to the presentation tier for delivery to the user.
2. **Directly integrated single channel pattern:** Provides point-to-point connectivity between the user and existing back-end applications. As with the Stand-alone Single Channel, it assumes one delivery channel and the user interface is handled by the presentation tier. The business logic can reside in the Web application tier and in the back-end application. The Web application

tier has access to local data that exists primarily as a result of this application, or example, customer profile information or cached data. It is also responsible for accessing one or more back-end applications. The back-end applications contain business logic and are responsible for accessing the existing back-end data. The communication between the presentation tier and Web application tier is synchronous. The communication between the Web application tier and the back-end can be either synchronous or asynchronous, depending on the characteristics and capabilities of the back-end application.

3. **As-is host pattern:** Provides simple direct access to existing host applications. The application is unchanged, but the user access is translated from green-screen type access to Web browser-based access. This is very quickly implemented but does nothing to change the appearance of the application to the user. The business logic and presentation are both handled by the back-end host. Because the interface is still host driven, this is more suited to an intranet solution where employees are familiar with the application.
4. **Customized presentation to host pattern:** This is one step up from the As-is Host pattern. The back-end host application remains unchanged, but a Web application now translates the presentation from the back-end host application into a more user-friendly, graphical view. The back-end host application is not aware of this translation.
5. **Router pattern:** The router pattern provides intelligent routing from multiple channels to multiple back-end applications using a hub-and-spoke architecture. The interaction between the user and the back-end application is a one-to-one relation, meaning the user interacts with applications one at a time. The router maintains the connections to the back-end applications and pools connections when appropriate, but there is no true integration of the applications themselves. The router can use a read-only database, most probably to look up routing information. The primary business logic still resides in the back-end application tier.

This pattern assumes that the users are accessing the applications from a variety of client types such as Web browsers, VRUs, or kiosks. The router provides a common interface for accessing multiple back-end applications and acts as an intermediary between them and the delivery channels. In doing this, the application pattern may use elements of the Integration patterns.

6. **Decomposition pattern:** The decomposition pattern expands on the router pattern, providing all the features and functions of that pattern and adding recomposition/decomposition capability. It provides the ability to take a user request and decompose it into multiple requests to be routed to multiple back-end applications. Responses are recomposed into a single response for the user. This moves some of the business logic into the decomposition tier, but the primary business logic still resides in the back-end application tier.

7. **Agent pattern:** The Agent pattern includes the functions of the decomposition tier, plus it incorporates personalization into the application to provide a customer-centric view. The agent tier collects information about the user, either from monitoring their habits or from information stored in a CRM. It uses this information to customize the view presented to the user and can perform cross-selling functions by pushing offers to the user.

## 1.4 Self-service pattern and the Trade application

Trade is an end-to-end Web application modeled after an online brokerage. Trade leverages J2EE components such as servlets, JSPs, EJBs, and JDBC to provide a set of user services such as login/logout, stock quotes, buy, sell, account details, etc., through a standards based HTTP protocol. This is a typical self-service application where the customer end-user interacts with a presentation tier that communicates with business logic on an application tier. The business logic on the application tier manages the required data access.

In its simplest form, the Trade application fits the Stand-alone Single Channel application pattern. It is easy to envision a Trade system that accesses existing data on backend tiers and interacts with backend applications; thus, becoming an application that matches the Directly Integrated Single Channel application pattern.

Chapter 2, “Trade application overview” on page 9 introduces the Trade application and subsequent chapters discuss the development and enhancement of elements of the Trade application using the capabilities of Versata business logic automation for WebSphere.



## Trade application overview

As an example of the Self-service business pattern, a stand-alone single Channel application pattern gives us a starting point for understanding the Trade application.

**Tip:** An IBM created benchmark is available at:

[http://www.ibm.com/software/webservers/appserv/wpbs\\_download.html](http://www.ibm.com/software/webservers/appserv/wpbs_download.html)

The business logic in Trade will be automated using rules as a part of the activities for this redbook.

Trade is an end-to-end Web application modeled after an online brokerage. Trade leverages J2EE components such as servlets, JSPs, EJBs, and JDBC to provide a set of user services such as login/logout, stock quotes, buy, sell, account details, and so forth, through a standards based HTTP protocol.

## 2.1 Trade application functionality

Figure 2-1 illustrates the Home Page. Trade provides HTML buttons to initiate the following functions:

- ▶ Register a new user to establish an account and profile.
- ▶ Login to the application. The user name and password are validated from a database table and a session is created.
- ▶ View the Home Page. The page is displayed with the user's current balance and current market conditions.
- ▶ Review and update an account. This allows the user to view and modify his profile.
- ▶ Obtain security quotes and purchase stock. Each purchase establishes a new portfolio "holding" for the user.
- ▶ View the portfolio. The page is displayed with all of the user's holdings. Holdings may be sold from the portfolio. Holdings must be sold in their entirety, for example, you can't sell 10 shares from a 100 share holding.
- ▶ Logoff from the Trade application and close the user session.

The screenshot shows the Trade application's home page. At the top, there is a blue header with the IBM logo on the left and the text 'WebSphere Performance Benchmark Sample' in the center. Below the header, a navigation sidebar on the left lists several links: Overview, Technical Documentation, Benchmarking, Configuration, Go Trade!, Web Primitives, and Setup Instructions. The main content area is titled 'Trade' and includes a 'Trade Home' button. A status message indicates a successful login for user 'uid:1'. Below this, a table shows 'Current Market Conditions' for Dow Jones Industrial (10,000 (+25)) and Nasdaq Composite (4,400 (+23)). A welcome message for 'uid:1' is followed by the current account balance of \$239556.0. A prompt asks the user to select from available services, with a 'Go Trade!' link. A row of buttons includes Home, Account, Portfolio, Quote/Buy (with a text input field containing 's:1'), and Log Off. The footer contains the text 'Powered By WebSphere e-business' and 'Created with IBM WebSphere Advanced, Visual Age for Java and WebSphere Studio Copyright 2000, IBM Corporation'.

Figure 2-1 Trade user's home page



Although the Trade application was primarily designed to test specific aspects of WebSphere performance, it has a well-designed Model-View-Controller architecture that is very useful as a case study of J2EE programming techniques.

## 2.1.1 Trade client design using MVC

As described in the Self-service Pattern Redbook, the Model-View-Controller (MVC) architecture, illustrated in Figure 2-2, is a common way of partitioning applications for maximum flexibility, maintainability, and reuse.

In MVC, a Model component represents an application object that implements the application data and business logic. A View component is responsible for formatting the application results and dynamic page construction. The Controller component is responsible for receiving client requests, invoking the appropriate business logic, and based on the results, selecting the appropriate view to be presented to the user.

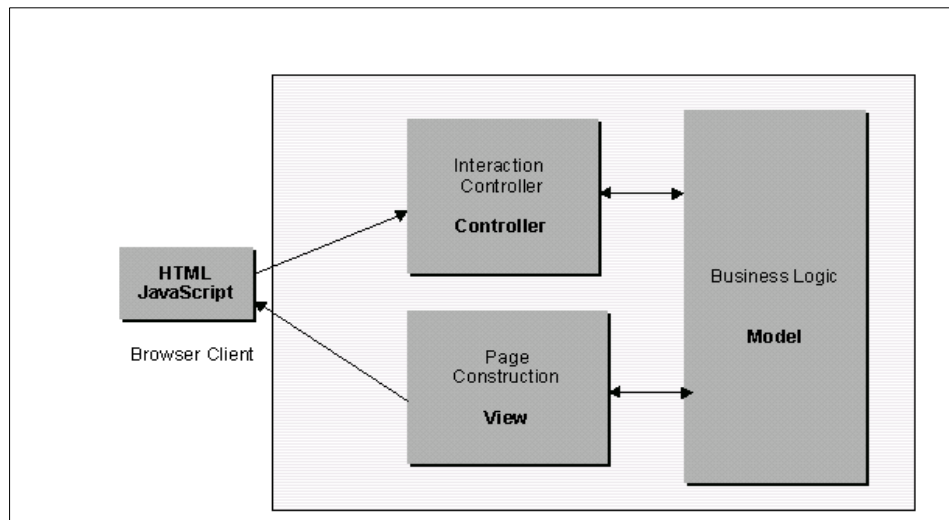


Figure 2-2 Model-View-Controller architecture

In the Trade application, the Controller is implemented as a servlet and several Java classes. The main controller servlet, TradeAppServlet, provides a standard Web interface to Trade users. It maps user input, such as requests to “Buy” or “Update Account”, to actions handled by an event handler class, TradeServletAction.

When an action is sent to TradeServletAction, it invokes the appropriate method from an implementation class, TradeAction, and matches the output of the action to its specific JSP. The controller is illustrated in Figure 2-3.

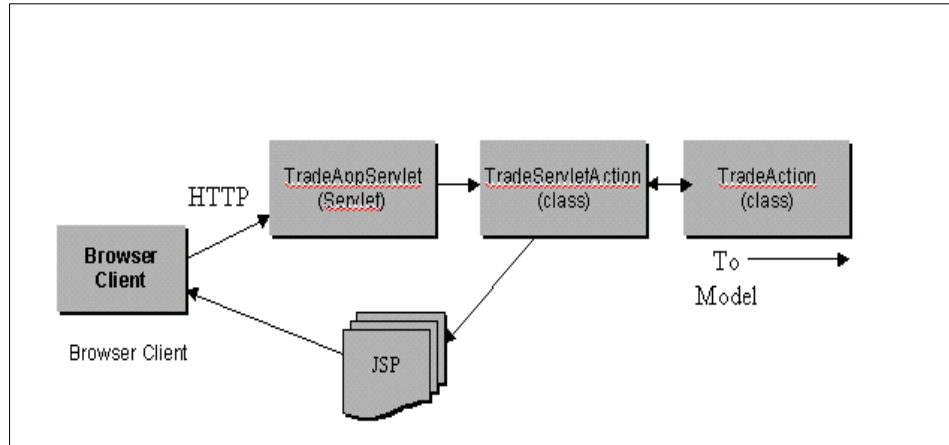


Figure 2-3 Trade application controller implementation

In the Trade client, JSPs provide the Views. The TradeAction class also interfaces to the Model, which, as we shall see below, can be configured in a number of ways. This four-step process, while seeming a little more complex than normal, allows for maximum flexibility when configuring the runtime behavior of the Trade application.

## 2.1.2 Multiple runtime modes

One of the most interesting features of Trade is the ability to choose, through set-up parameters, a runtime “mode” of the application. One mode of operation implements business logic and transactions (the Model in the MVC architecture) in the client-tier using simple Java classes. In this mode, data is accessed directly from the database through JDBC. This is illustrated in Figure 2-4.

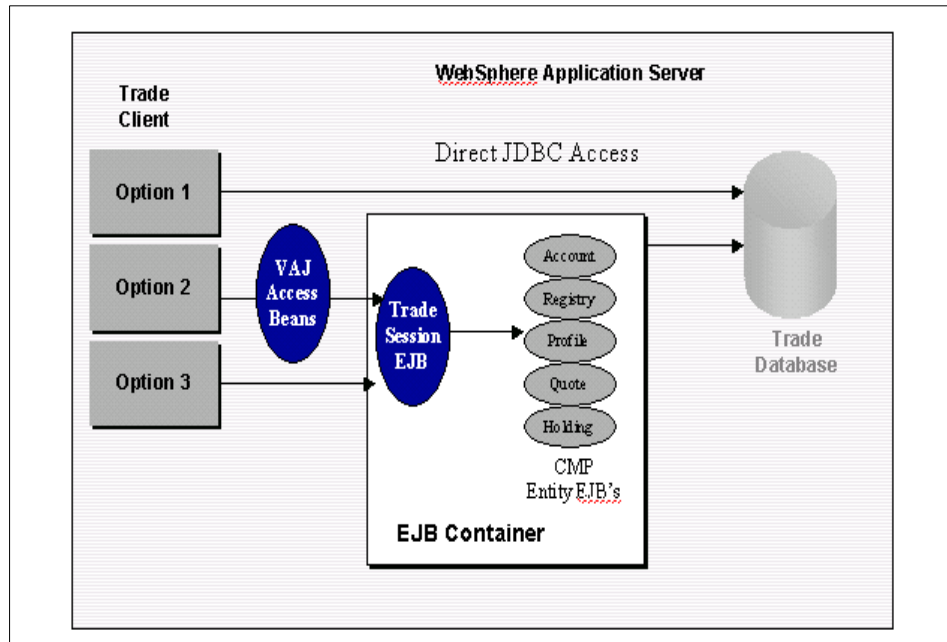


Figure 2-4 Multiple runtime modes

Two other modes implement business logic and transactions in the EJB-tier, using a stateless session EJB, TradeBean, to implement most of the business methods. Database access is implemented using an entity bean for each of the database tables. Persistence is managed through the WebSphere EJB container using container managed persistence (CMP). The two EJB modes differ in the way that they access the Trade session and entity EJBs. One mode uses optimized EJB access beans, generated by VisualAge for Java. The other uses direct, remote, EJB access from the client.

The purpose of multiple Trade modes is to compare the performance of JDBC versus both forms of EJB access. Readers can also examine each implementation to judge its relative ease-of-use.

In the application documentation, Trade developers discuss the tradeoffs between JDBC and EJB access. They note that well-written JDBC code will generally outperform the EJB equivalent, while JDBC may be significantly more complex to develop and maintain. Most of the JDBC complexity comes from the need to explicitly manage transactions and difficulty with logic re-use when business logic is placed in the client-tier.

In summary, data for the Trade application is kept in a relational database and Trade provides three, distinct, paths to the data:

- ▶ JDBC to database access using direct JDBC calls from the Web tier (no EJBs)
- ▶ EJB to DB access using EJB container managed persistence by leveraging IBM VisualAge for Java EJB Access Beans for EJB access.
- ▶ EJB to DB access using direct EJB access by leveraging EJB Access Beans

## 2.2 Versata: a new option

The reason for choosing the Trade application as a starting point for this Redbook is the flexibility provided by the MVC architecture. Because of this, we can easily implement a fourth and fifth option — accessing the database through business objects managed by the Versata Logic Server.

As we shall see in Chapter 4, “Architecture of the Versata Logic Server within WebSphere” on page 39, these objects contain business logic designed through declarative rules. The rule-enabled objects are installed into a WebSphere EJB container where they execute using runtime services provided by the Versata Logic Server, as shown in Figure 2-5.

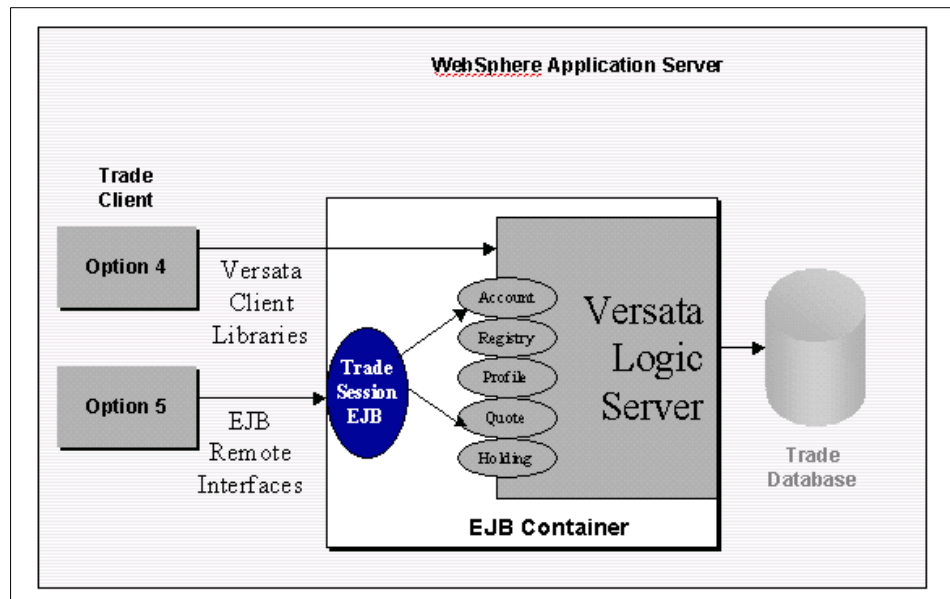


Figure 2-5 Versata Logic Server

Versata rule-enabled EJBs can be accessed from the Trade client, through their EJB interfaces, and through the Versata Logic Server client libraries. The client library option is most closely examined for its potential to bridge the EJB-JDBC gap, encapsulating business logic in the EJB tier while providing higher-speed access from client applications.

## 2.3 Details of the Trade EJB implementation

We conclude this chapter by examining the Trade EJB implementation in more detail. This will allow us to understand the Versata implementation more easily.

### 2.3.1 Database schema

The database schema to support Trade is straightforward, as illustrated in Figure 2-6.

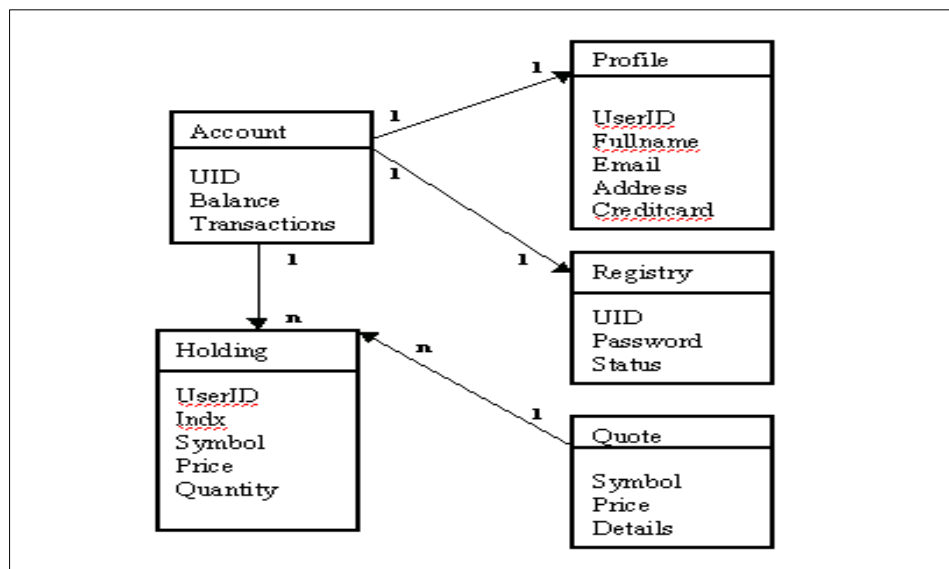


Figure 2-6 Database scheme

There are five database tables:

- ▶ **Account:** This holds the user ID and current balance.
- ▶ **Profile:** This provides more information about the user, such as address and e-mail.
- ▶ **Quote:** This holds stock symbols and descriptions.
- ▶ **Holding:** This holds the result of a Buy operation. A holding row contains the user ID, the stock symbol, the quantity purchased, and the purchase price of the stock (per share).
- ▶ **Registry:** This holds the user ID, password, and status. Authentication and authorization in Trade is done through the application. This is a departure from a normal Topology 1 design that relies on WebSphere to authenticate users.

### 2.3.2 Container managed EJBs

There are five container managed entity EJBs that correspond to each of the Trade database tables. Each EJB implements the get and set methods for its own attributes. In addition, each entity EJB provides a `findByPrimaryKey()` method that allows the database row corresponding to the object to be retrieved by its unique index.

EJBs that need to support more complex queries use “Finder Helper” classes for each potential query structure. An example is the `HoldingBeanFinderHelper` class, which implements methods to find holdings by user ID, find the maximum holding index number, and so on.

Each of the five entity EJBs has a corresponding Access Bean “wrapper”, generated by VisualAge for Java. When accessing an entity EJB directly from the client, these access bean wrappers reduce the complexity of remote access through a simplified client API and improve EJB performance through a number of techniques.

In general, however, entity EJBs are not directly manipulated by the Trade client, even when operating in EJB mode. Instead, operations go through the session bean, `TradeBean`, through its access bean wrapper, the `TradeAccessBean`.

An example of end-to-end processing is illustrated in Figure 2-7.

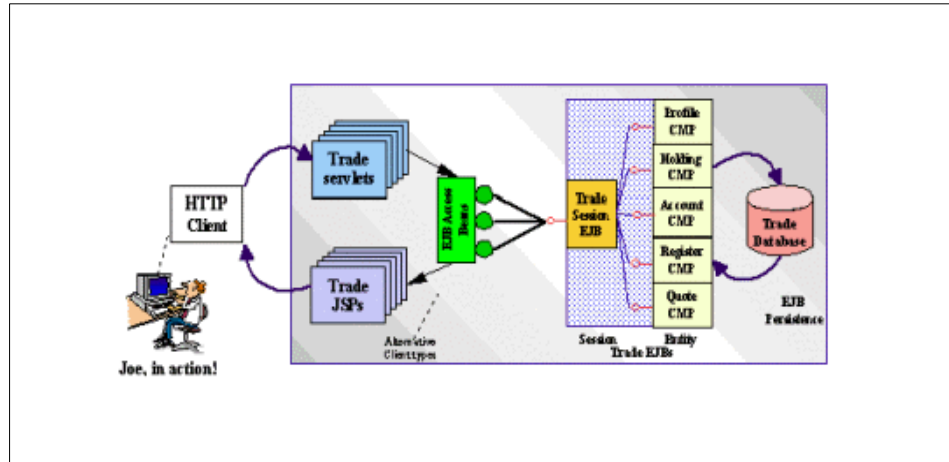


Figure 2-7 End-to-end processing

### 2.3.3 Use of copy helper access beans

Some business functions, however, bypass the TradeBean. Business functions that update several items of client data at the same time need a more efficient way of setting EJB attributes. This is the case, for instance, when a user updates his personal profile.

Profiles are maintained by the `doAccountUpdate()` method. To optimize `doAccountUpdate()`, the TradeAction class directly calls the ProfileAccessBean. The ProfileAccessBean is a VisualAge for Java copy helper access bean. This bean provides optimizations designed specifically for updates. It allows individual attributes of an object to be updated in the client tier before committing the entire updated row in a single remote call.

To optimize performance, Trade designers have employed a sophisticated mix of session EJBs, entity EJBs, plus wrapper and helper classes for their business logic. This is in addition to the various JSP, servlets, classes, and other components used for the MVC architecture of the client tier.

### 2.3.4 Basic business logic in Trade

Compared to the operations of a real life brokerage application, the business logic in the basic Trade application is minimal. Of course, this is because Trade was designed for performance testing, and not as a fully operational business application.

If we consider that business logic is the data processing that must occur to carry out organizational functions and policies, we can summarize the main, transactional business logic in Trade.

For the buy() method, whose signature is “public double buy(String userID, String symbol, double quantity)”

- ▶ Find the account for the current userID; do an error check if there is no account or non-existing account.
- ▶ Find the quote price for the stock symbol; do an error check if there is no symbol or non-existing symbol.
- ▶ Multiply the quote price by the quantity.
- ▶ Create a holding object with for the specified user with the amount, stock, and quantity.
- ▶ Within the same transaction, debit the user's account balance by the amount of the transaction. (The debit function will need to be added to the Account EJB to supplement the default get and set methods.)

For the sell method, the logic is similar but reversed, although more sophisticated coding may be needed to find the correct holding to sell.

## 2.4 Potential enhancements to Trade business logic

In the remainder of this redbook we examine how the initial business functionality in Trade can be automated using the Versata Logic Server. We also consider how the application can be enhanced to meet real-life requirements such as:

- ▶ Allowing users to sell partial holdings. This includes adding a transaction table to save the buy and sell operations for each holding.
- ▶ Controlling “margin selling”. This includes checking balances and margin rules before sell operations.
- ▶ Much more extensive checking and validations in the EJB tier. The current Trade application implements most data validation and error checking in the client tier (in the TradeAction class). Placing field length, data type, and other validations in the business logic will allow it to be shared among multiple applications.
- ▶ Adding a flexible commission system that can maintained by business managers.

We will also examine the ease of automating this functionality with high level declarative rules.





## Business logic automation using rules

The Versata Logic Suite is a business logic automation engine and design studio that captures business logic rules and executes them as J2EE components running within WebSphere. It is designed to simplify WebSphere application development and to make J2EE systems more general, flexible, understandable, and reusable.

Versata defines the process of creating and executing systems directly from their high level rule specifications as automation. Automation allows the developer to specify “what” the logic should do, not “how” it is to be implemented.

Because Versata rules are easier to understand, write, change, and debug than the hand-coded programs they replace, software systems can be built faster and with more consistent quality. In addition, the time and quality benefits should increase over the entire software life cycle, since new requirements can usually be handled by simple rule changes and additions. These changes are applied consistently during re-automation by Versata and typically do not introduce new bugs or errors — the downfall of most software maintenance efforts.

### 3.1 Scenarios for automating WebSphere applications

There are two scenarios for automating WebSphere applications using the Versata Logic Suite.

In the first scenario, the Versata Logic Suite is used to automate the business logic of a WebSphere system and is also used to create the Versata-automated client application. This scenario is examined in Chapter 6, “Designing an HTML client application” on page 73, which introduces the Versata Presentation Server (part of the Versata Logic Suite).

In the second scenario, the Versata Logic Suite is used to create the transactional business logic (EJB tier) of a WebSphere application, while non-Versata components are integrated to provide the presentation logic (Web tier). This scenario is examined in Chapter 10, “Integrating Versata Logic Suite with WebSphere Studio Application Developer” on page 167, where the existing Trade client, built with WebSphere Studio and VisualAge for Java, is integrated with Versata's rule based business logic.

An illustration of both scenarios is shown in Figure 3-1.

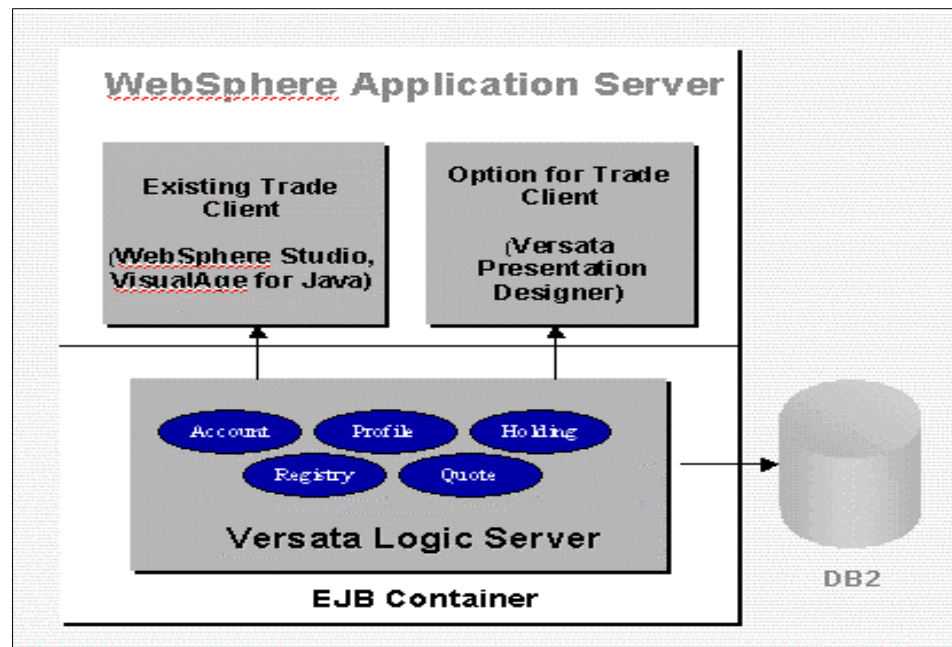


Figure 3-1 Two scenarios for automating WebSphere applications

**Note:** In either client scenario, VisualAge for Java can have an important role in a Versata-automated system. For instance to, integrate Versata business logic to enterprise resources, Versata rules can call Visual Age for Java components such as Enterprise Access Beans. This ability allows new, rule-based logic to interact with legacy applications, MQSeries middleware, and other specialized EJBs. In addition, VisualAge for Java can be used as the Java debugging environment for the Versata Logic Server. These, and other potential integration points, are explored in Chapter 10, “Integrating Versata Logic Suite with WebSphere Studio Application Developer” on page 167 and Chapter 11, “Developing with UML and rules” on page 193.

## 3.2 Design and runtime environment

The Versata Logic Suite includes a design environment, the Versata Logic Studio, and a runtime environment, the Versata Logic Server.

The Versata Logic Studio includes a transaction rules designer to specify business logic, and an optional presentation designer to specify complete HTML or Java client applications. The Versata Logic Studio runs as a desktop application.

The Versata Logic Server includes a transaction rules engine to execute business logic, and an optional presentation engine to execute client application logic. The Versata Logic Server installs into an EJB container provided by the WebSphere application server. A development copy of the Logic Server is provided with the Versata Logic Studio for testing.

These are illustrated in Figure 3-2.

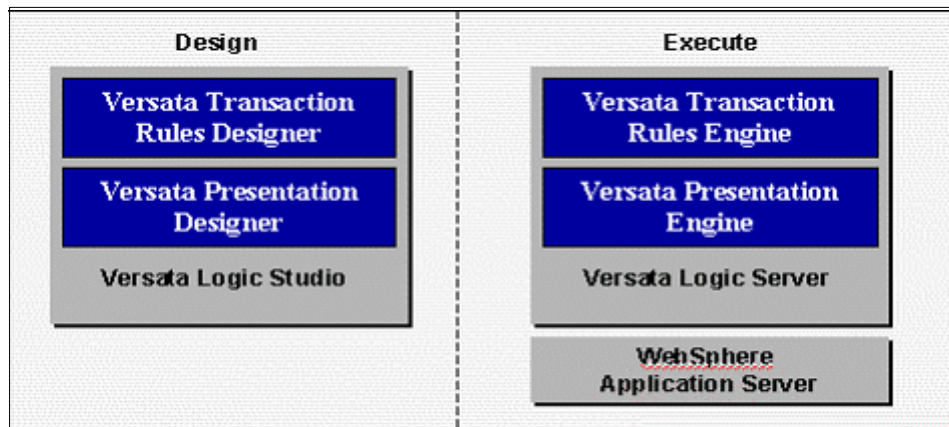


Figure 3-2 Versata design and execution environments

As noted previously, the Versata Transaction Rules Designer and Engine can be used with or without the Versata Presentation Designer and Engine. Both use a high level development approach.

The remainder of this chapter examines some characteristics of Versata rules created in the Transaction Rules Designer.

## 3.3 Transaction rules for automated business logic

In the software industry, the term “rule” has been used to describe several technologies. Within messaging systems such as IBM MQSeries, rules are used to direct the routing of messages. For Web personalization, such as that provided by modules in the WebSphere Application Server, rules are used to control the presentation of content. Finally, within Artificial Intelligence (AI)-like inference engines, rules are used to leverage “expert knowledge” to solve a specific business problem.

For clarity, we will now distinguish Versata Logic Server rules from the other rule technologies.

**Important:** Versata Logic Server rules are high level, unordered assertions about data used to formulate and direct the transactions within a J2EE application server.

Unlike messaging systems, they do actual logic execution. Unlike personalization add-ons, they carry out the core business functions of a WebSphere application. And, unlike inference engines, they are not situated “outside” of the system to provide inferred conclusions.

### 3.3.1 Example of a rule

Having said that Versata rules are assertions about data, Example 3-1 is an assertion:

*Example 3-1 Assertion example*

---

For an object Account,  
the value of the attribute ActiveHolding  
is  
the number of Holding objects associated with this Account where the  
QuantityOnHand of the Holding is greater than zero.

---

Figure 3-2 shows a typical rule definition within the Versata Studio.

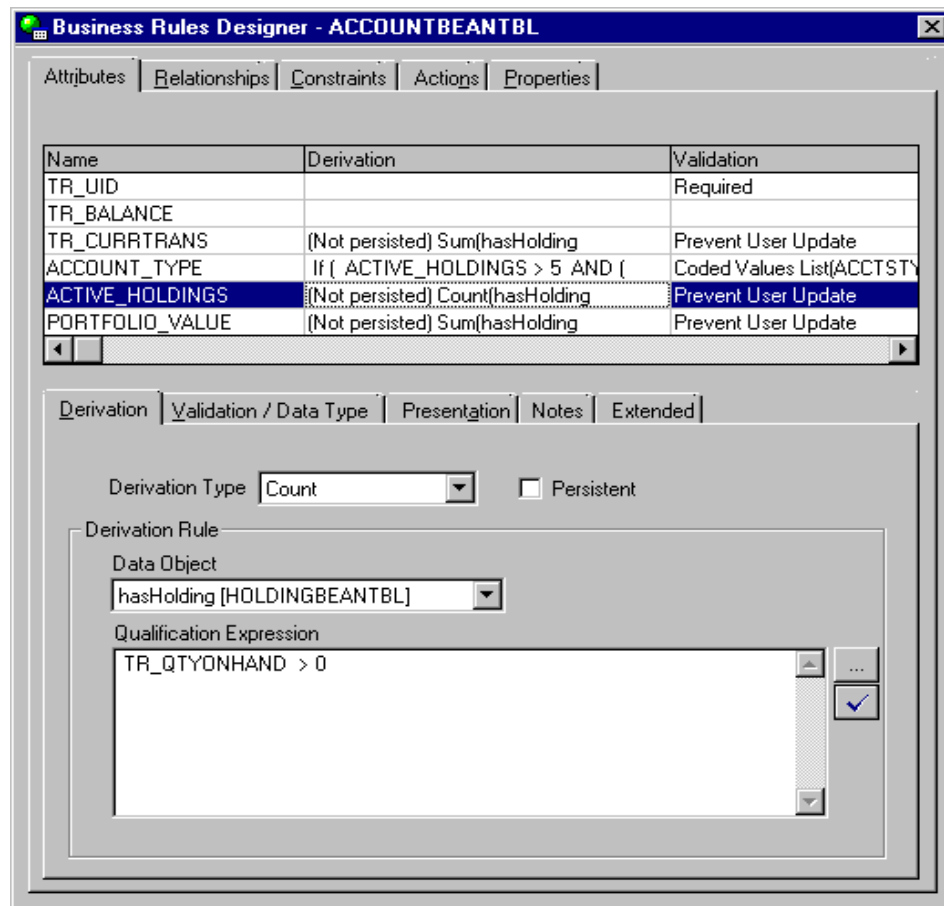


Figure 3-3 The rule specifying the value of ACTIVE\_HOLDING

The transactional effect of this assertion is that, when the QuantityOnHand of a Holding is changed (when buying or selling a stock, for instance), the ActiveHoldings count in the Account object will be automatically examined. If a new Holding is added, or if the shares of a Holding drop to zero, it will be adjusted up or down as needed.

**Note:** The number of ActiveHoldings will be used as part of a personalized commission calculation in the rule-enhanced Trade application.

### 3.3.2 Characteristics of rules

Here are some important points to notice about the rule:

- ▶ **It implements logic for multiple objects.** From what we have seen, the rule affects at least two entities — Account and Holding. In fact, the rule will also involve a third entity — a new Transaction object. Logic in the third object will be automatically created because Holding has a rule to derive its QuantityOnHand from the Transaction entity.
- ▶ **It is declarative.** This means that the designer did not specify how the system is to get this result. The developer did not need to decide whether the “count” function resides in the Holding EJB, or the Account EJB or a third, session EJB. He did not need to implement arrays of Holdings, counters, or cross-object get and set methods.
- ▶ **It is transaction independent.** There are several transactions that affect the on-hand quantity of a stock in a holding. A buy transaction will increase the quantity: A sell transaction will decrease it. In addition, new transactions may be added in the future (to convert options to stock, for instance.) These all may affect the on-hand quantity of a stock. The important thing is that the rule to re-calculate ActiveHoldings will be applied automatically, to any transaction that requires it. No developer analysis is needed to determine when to apply the rule.
- ▶ **It is unordered.** Although a specific sequence of operations will be implemented (Holding will be updated before Account), the developer need not worry about the ordering of operations. A rules compiler in the Versata Logic Server will unravel the dependency between objects and sequence operations correctly.
- ▶ **It will be automatically optimized for runtime.** As we saw in the original Trade application, performance is frequently a concern when utilizing EJB remote interfaces and container managed persistence. The Versata Logic Server provides services within the WebSphere Application Server to overcome many of these concerns.

For instance, the Versata Logic Server maintains a transaction cache for cross-entity interactions inside of WebSphere. This means that the appropriate Transaction, Holding and Account objects will be brought into the Logic Server cache in a single read from the database. The entire rule chain will execute from this cache, speeding execution while guaranteeing data integrity.

In addition, all the interactions between Transactions, Holdings, and Accounts will be done with simple Java method calls, rather than with expensive EJB access. Current EJB containers (those adhering to the EJB 1.1 specification) impose significant overhead on EJB calls, even between beans using the same Virtual Machine.

This overhead includes RMI call-by-value semantics, permission checks, and transaction control. To avoid this overhead, the Versata Logic Server implements business object rules in lightweight Java class “helpers” — one for each entity EJB. Thus, all Versata rule processing will use simple Java method calls, greatly improving the performance of rule processing versus hand-coded EJB transactions.

**Note:** EJB 2.0 compliant servers are expected to adopt this approach to local object access.

Chapter 4, “Architecture of the Versata Logic Server within WebSphere” on page 39, details the runtime architecture of the Versata Logic Server within WebSphere and explains these performance features in more detail.

### 3.3.3 Rules and EJB domains

From the example above, we can make these additional observations about Versata rules.

The first observation is that rules apply to data, specifically they apply to data when it changes. From this we can see that, in applications where persistent data does not change, this type of rule will not be very useful.

**Note:** This characteristic is one of the main differences between “inference rule engines” and “transactional rule engines”. Inference engines collect their data and make decisions independent of transactions. The result of the decision may or may not be used in a transaction. With most inference engines, the transaction will be designed and coded by the developer using traditional coding methods.

Transactional rules engines, on the other hand, drive transaction logic. Automated transactions proceed (or don't proceed) because of rules. These automated transactions may be customized with hand-coded logic, but the bulk of the calculations, validations and evaluations come directly from the executing rule.

The second thing we observe about Versata logic rules is that they govern the interactions of a set of business objects (and their attributes) in a J2EE application. Here we will coin a new phrase - this set can be thought of as a domain of EJBs, where the domain is a collection of entity-type EJBs that will be needed to carry out a series of common business functions.

The specification of this domain usually begins with an object model (or a database schema.). In fact, the Versata Logic Suite, has utilities to import both object models (from Rational Rose, for instance) and schemas (from relational databases). These jump-start the definition of the domain. The remainder of the domain specification is done through rules.

**Note:** For those familiar with software engineering domain analysis, a repository of Versata business objects and rules capture both the commonalities and variabilities of a set of applications to be deployed in the enterprise.

The complete specification for all of the domain objects and their interactions is stored as “metadata”. This metadata is kept in an XML-formatted repository and is available, at design time, to the Versata Logic Server and other applications accessing objects in the domain.

What remains then is to classify the types of rules that can control the behavior of a domain of EJBs.

### 3.3.4 Classification of declarative logic (rules)

In the business rules community, there has been considerable analysis of the types of rules needed to specify the logic of business functions. As early as 1995, the GUIDE Business Rule Project established a vocabulary and taxonomy (classification) of rules on which Versata's definitions are largely based.

**Tip:** A paper describing this is available at:

[http://www.businessrulesgroup.org/first\\_paper/](http://www.businessrulesgroup.org/first_paper/)

**Note:** Because the Project did not tie their vocabulary to a specific technology, we have narrowed their definitions to make them more relevant to Java developers. For instance, a business “term” is narrowed to mean a Java class, entity, or business object (all synonymous) and its fields or attributes (also synonymous).



Figure 3-4 illustrates the various rule types based on the GUIDE Business Rule Project.

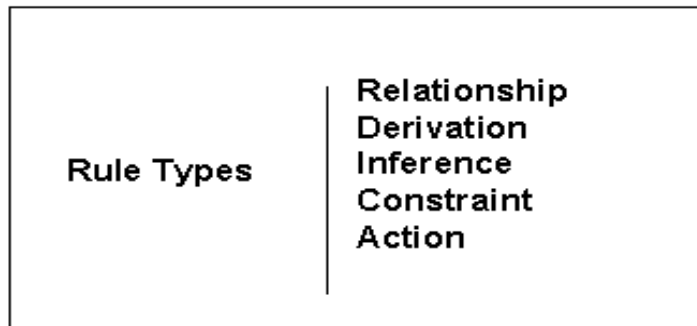


Figure 3-4 Based on GUIDE Business Rule Project

The rule types based on the GUIDE Business Rule Project are as follows:

- ▶ Relationship rules specify the association between entities. Rather than controlling transaction behavior itself, they *enable* other rules to be designed and enforced.
- ▶ Derivation rules are algorithms that derive attributes from other attributes. In Versata, derived attributes can be persisted or non-persisted (in which case they represent runtime only values).
- ▶ Inference rules use one or more truths to arrive at a new truth. The new truth usually derives an attribute.
- ▶ Constraint rules specify the policies of a business. They govern under what conditions operations can proceed.
- ▶ Action rules initiate another business event, message, or activity based on some condition.

In the following sections we provide some examples from the Trade application.

### Relationship example

Relationships typically imply a parent-child association between entities. Through the use of intermediate entities, they can also be used for more complex relationships. An example of a parent-child association in the Trade application is a rule that says, “An account has holdings”.

As we will see, the Versata Logic Server automates can almost all operations between related entities. For instance, an Account can automatically count and “sum” all of its Holdings. Holdings can automatically check for sufficient funds in the related Account.

Furthermore, as they are entered, Holdings can automatically check to ensure they are associated with valid Accounts. These are only some object behaviors enabled by relationships

## Derivation example

Derivations are *computational*, which means that the value of that attribute is arrive at by a formula. An example is a rule that says, for a Holding:

“QuantityOnHand = QuantityPurchased - QuantitySold”.

In addition, computational derivations can specify sophisticated qualifications and can navigate to other objects to use their values in computations. An example of this is a rule that says, for a sales Transaction:

“The Price used to calculate the Amount of the Transaction is the Price (from the Quote Object) of the associated Stock (specified in the Holding Object)”

From the Versata Logic Studio, the rule is shown in Figure 3-5.

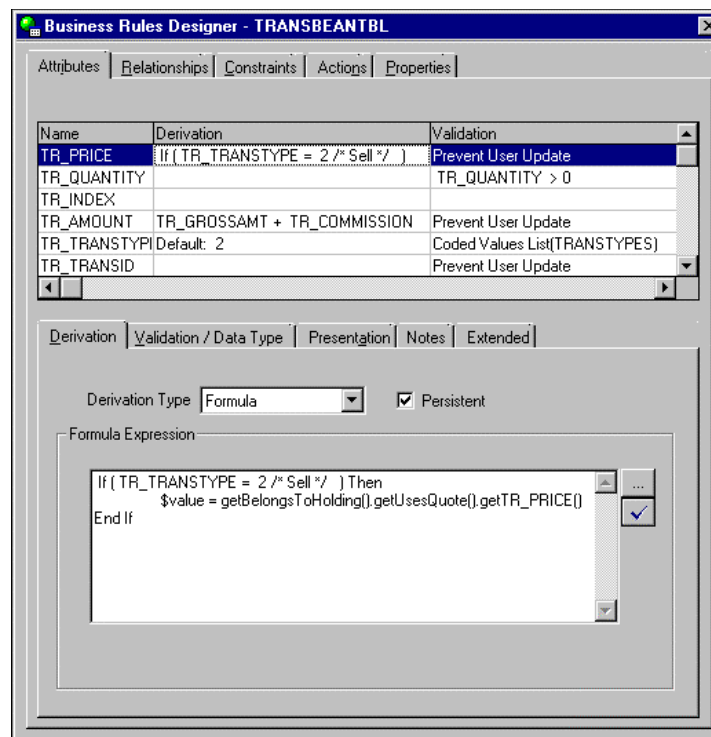


Figure 3-5 Rule example

## Inference example

Inference rules can be thought of as a specialized type of derivation, where an attribute is derived from a truth. An example is a rule that says, for an Account:

“If Balance is greater than \$500K, then the AccountType is 'wholesale'.”

Like computational derivation, inference derivations can chain together evaluations and calculations from several entities. It is possible, for instance, for a rule to say that, for an Account:

“An AccountType is 'wholesale' if the Account Balance is greater than \$100K [Account entity], AND the number of ActiveHoldings is greater than 20 [Holding entity], AND the average TransactionSize is > \$10K [Transaction Entity].”

## Constraint example

Constraints define legal states in the system. For transactional systems they define legal values of data that are allowed to exist. A rule to restrict (constraint) margin selling in the Trade application is one such state. It says, on Account, “Balance can't be less than zero”.

This simple-sounding constraint rule will automatically included in any rule chain that can potentially affect the Account Balance. For instance, here's a “Buy” rule chain, seen in more detail in Chapter 9, which will be constrained by the margin rule above.

When buying a stock, the following happens:

1. First, the Logic Server begins to create a new Holding for this Symbol, Date and Quantity.
2. It next finds the stock's price from the associated Quote.
3. It then begins to insert an initial “Buy” Transaction object.
4. Next, it calculates the cost of the stock [multiply Price times Quantity].
5. Then it finds the Commission [by checking the Account entity to see whether the AccountType is a wholesale or retail account, checking the current CommissionRate for that Account Type and multiplying the CommissionRate times the TransactionAmount].
6. Then it calculates the total Transaction Amount.
7. Next, it begins to Update the Transaction Count for this Holding.
8. Then it begins to debit the Account Balance with the Transaction Amount.
9. Finally, it encounters the constraint.
10. If the constraint is violated (if the new Balance will be less than zero).
11. It unwinds the entire operation [rolls back changes to the Holding, Transaction, and Account objects.

This example reinforces three important characteristics about rules.

First, notice that we did not have to write a “Buy” method to perform this chain of events. A simple change [insertion] to a Holding object began the process. The Logic Server understood all of the objects involved. It understood all of the rules and all of the cross-logic between the object attributes. It understood all of the dependencies, including which was the “inner-most” and “outer-most” object. It understood how to optimize the chain of events, including caching the Account, Holding, Transaction and Quote objects during the transaction because their attributes were needed several times. Finally, it understood where to put transaction boundaries and how to deploy the transaction into the WebSphere application server.

Second, notice that the rule is not tied to any particular transaction. Instead, it applies to the “state” of the system. The rule would apply also if the user tried to withdraw money from his account - his would be prevented from withdrawing more funds than he had. The Logic Server is responsible for maintaining the declared state and the Logic Server is responsible for identifying all the possible transactions that affect that state.

Third, notice that the declaration of rules was completely unordered. We actually created this rule to constrain margin selling before we even knew how the Balance would be derived. Similarly, declared how Commission was derived before worrying about how the stock price was going to be found.

This is “what” not “how” — terms often used when categorizing declarative logic rules.

Another type of constraint rules — a transition constraint — is possible:

- ▶ Transition constraints define legal transitions, or changes from one state to another. An example is a rule that says:

“Old QuantitySold cannot be more than New QuantitySold”

(In other words, in the Trade application, it is not possible to “un-sell” a stock)

During rules processing, the Logic Server maintains old and new values of all attributes. This allows rules to easily check and constrain transitions between object states.

### **Action example**

The final type of rule consists of action rules, more completely referred to as an Event/Condition/Action. The *Event* is the operation and entity being watched by the Logic Server. The *Condition* must be met in order to proceed. The *Action* is the side-effect that should result from a condition being met.

An example of this is a rule that says:

“When adding a new Transaction, if the TransactionType is “sell”, credit the related Holding's related Account Balance”.

The Logic Server will take an action when all other rules leading up this event/condition have evaluated as “true”.

## 3.4 Business uses of rules

It is also helpful to classify the business use of rules. Although there are many more, here are six business uses of transactional rules:

1. Rules can automate data validation. Data and transactions are more accurate. Validation coding is eliminated.
2. Rules can preserve the association between related objects when they change. System consistency is assured. Coding to check and maintain relationships is eliminated.
3. Rules can automatically synchronize related attributes in different objects. Complex transactions are always implemented correctly. Performance is optimized.
4. Rules can enforce operational policies and respond to change when policies change. Logic is defined in well-understood declarations. Policies are enforced uniformly.
5. Rules can identify interesting data. Flagged data can be used for personalization or cross-marketing.
6. Rules can initiate asynchronous events. Events integrate the rule-based system with external applications. Events can also initiate exception processing.

In the following topics we supply details and examples of the EJB hand-coding that is replaced by rules:

### **1. Rules can automate data validation. Data and transactions are more accurate. Validation coding is eliminated.**

A fundamental use of rules in the Versata Logic Server is for data validation. Validation rules are a type of constraint.

Although validation is frequently overlooked when estimating the amount of business logic in an application, its design and development consumes costly programming resources. Moreover, validation (and related error handling) code is usually sprinkled throughout various client-tier and logic-tier components. This makes logic difficult to re-use and maintain.

The Trade application, for example, checks user input first in the TradeAppServlet, validates the parameters in the TradeAction class, and finally confirms them in the Trade session EJB. If the Trade application were to do more extensive validations, or if the data types in the entity EJBs were to change, many components would require maintenance.

Versata rules allow data validations to be attached to objects directly, as part of their metadata. This is illustrated in Figure 3-6, where the Account Type (wholesale, retail or new) is validated from a list of values. (To optimize performance, Versata Logic services can cache this list in the Web-tier of the application server and share it among users.)

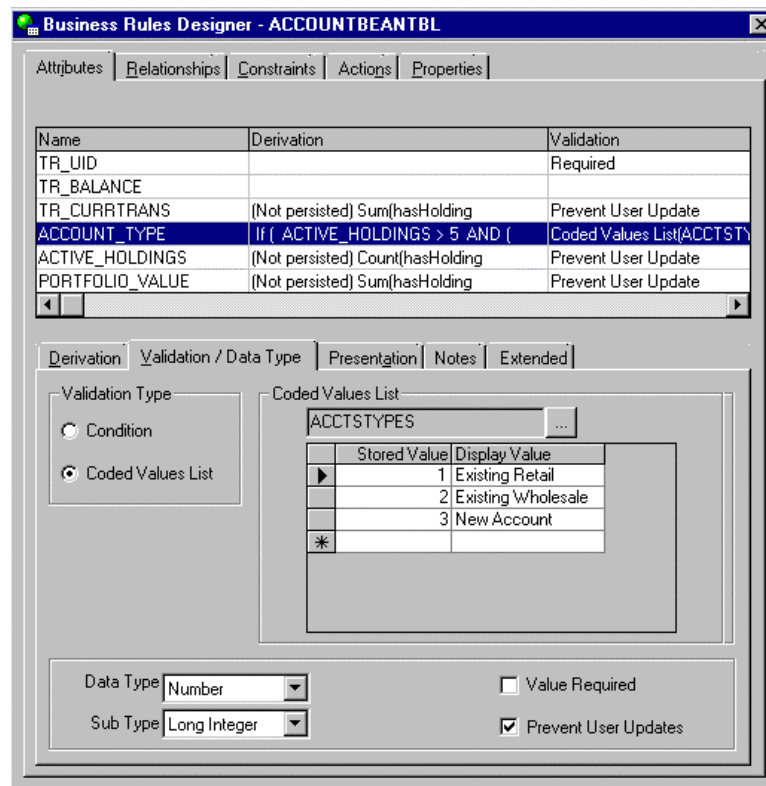


Figure 3-6 Validation from a cached list of values

These are some other validations for the Account\_Type attribute shown here:

- ▶ The User must enter a value.
- ▶ The value will be translated, using the validation list, to a Java long integer (this is stored in the database).

Using rules to specify attribute validations with rules has several benefits. First, validations, like all rules, will be applied consistently across all applications. This increases system integrity.

In addition, validation metadata, like other rule-based metadata, can be accessed through object methods. This makes it possible for a client components to derive their behavior from business entity metadata. This extends the use of the “Model” in the Model-View-Controller architecture and could allow client components to adapt automatically to changes in business logic.

**2. Rules can preserve the association between related objects when they change. System consistency is assured. Coding to check and maintain relationships is eliminated.**

Referential integrity refers to the need to maintain consistency between one business object and all of the other objects that refer to it. For example, in the Trade application, where there is a relationship between an Account and its Holdings, it would not be good business policy to allow an Account to be deleted if it had active Holdings. Similarly, the business undoubtedly has a policy ensuring that Holdings are created only if the user has a valid Account.

In applications where all data is kept in the same relational database, enforcing referential integrity can be left to the database management system. If, however, there is the potential for data to come from more than one source, integrity must be assured through application code. In J2EE applications, the place for this code is in EJBs (Whether it should be placed in the parent entity EJB; or the child entity EJB; or in a third, session EBJ — is a question frequently debated.)

Versata business logic rules provide an easy and consistent way to enforce relationships and integrity, even between objects in different databases or legacy applications.

Figure 3-7 shows a Trade referential integrity rule specified in the Versata Studio. Here, the Account entity has two relationships defined: one to the Holding entity and one to the Profile entity. Versata rules specify how the relationship between Account and Holding should be formed (by userID) and specifies the types of referential integrity that will be enforced.

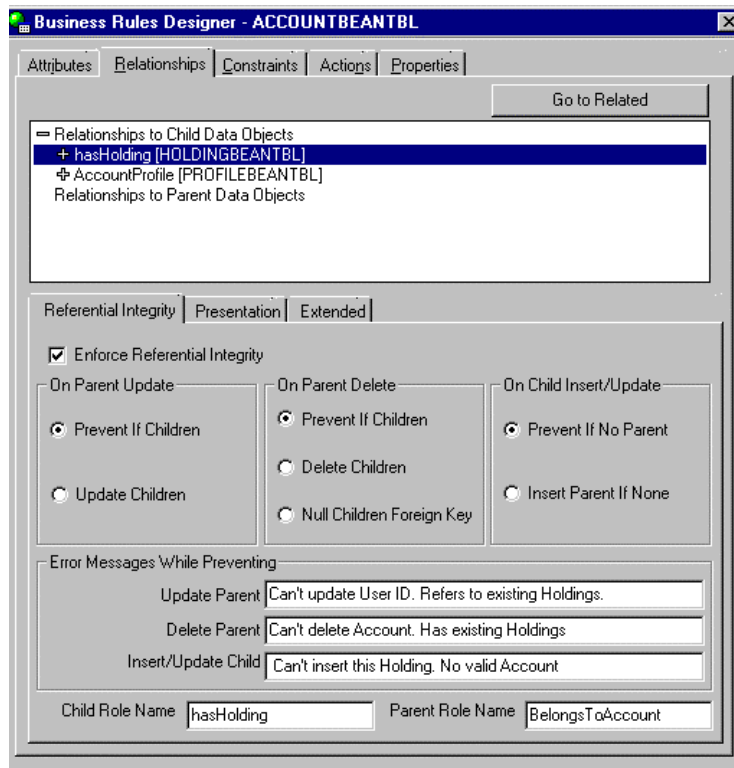


Figure 3-7 Account relationships and referential integrity

Versata rules can enforce very sophisticated integrity. For instance, to perform automatic clean-up, a rule could be defined that, when deleting an Account, the user's Profile should be deleted as well.

Enforcing referential integrity in EJBs greatly increases integrity of data entered. Automating this behavior with rules improves the functionality of systems without tedious hand-coding.

**3. Rules can automatically synchronize related attributes in different objects. Complex transactions are always implemented correctly. Performance is optimized.**

One of the most common patterns coded into EJB logic is getting and setting the attributes in related objects within the scope of transactions. The IBM-version of the Trade application "Buy" function is a simple example. Given a UserID, stock Symbol and Quantity, the Buy operation:



- ▶ Finds the related stock from the Quote entity (the developer first defines a find method to do this)
- ▶ Checks that the Quote is valid and gets its price
- ▶ Finds the related account for that UserID (the developer first defines a find method to do this)
- ▶ Checks that the account is valid and gets its balance
- ▶ Creates a holding and checks to see that this succeeds
- ▶ Debits the account balance (the developer first defines a debit method to do this)

(The Trade code notes that the logic to check that there are sufficient funds to buy the holding has not been implemented.)

As we see in Chapter 6, “Designing an HTML client application” on page 73, rules can be used to automate this transaction. The relationship rule between the Holding and Quote entity allow us to automatically get the price of a stock. The relationship between the Holding and Account allow us to automatically get and update the Account balance. And as we saw earlier, logic to check for sufficient funds can also be implemented with a simple constraint.

One of the most effective uses of Versata Logic Server rules is to automatically implement complex, cross-entity transaction logic. Rules assure that the logic is implemented correctly, and that it can be enhance with simple rule modifications.

#### **4. Rules can enforce operational policies and respond to change when policies change. Logic is defined in well-understood declarations. Policies are enforced uniformly.**

Often, organizations begin to look at rule-based systems when they need comply with government or industry regulations, especially when those regulations change. Constraint rules are useful for this purpose.

In the rule-based implementation of the Trade application, we will add a rule to control margin accounts. For each user, a margin limit will be calculated based on the current SEC regulation for that account type. Buy operations will be permitted or rejected by consulting the current SEC cut-off.

#### **5. Rules can identify interesting data. Flagged data can be used for personalization or cross-marketing.**

Although transactional rules are not primarily used for personalization or data mining, such rules do “watch” transactions as they flow through the J2EE application server.

For example, with a rule on the Account entity, a sales representative can be automatically notified when a new user, with an account balance greater than \$100K, is added to his territory. Or a user may set a flag on his holdings, to notify him when a stock drops more than 25%

Since rules watch for changing data, any behavior initiated by a change in state can be a candidate for a rule.

**6. Rules can initiate asynchronous events. Events integrate the rule-based system with external applications. Events initiate exception processing.**

Although Trade is an entirely synchronous application (transactions are committed or rolled back immediately), most enterprise systems have some asynchronous operations.

For example, when a user updates his account information, a rule could create an XML formatted message and place it on a message queue to be picked up by the corporate CRM system. Or the system could start a workflow process, and advise an investment representative to contact this customer who had just deposited \$100K in his account.

Rules watch data changes and can initiate synchronous transactions or asynchronous events based on those changes.

## **3.5 What the Versata Logic Suite is not**

We conclude this chapter with a discussion of what transactional rules are not. This may help to clarify when and when not to use transactional rules for WebSphere applications.

As we discussed, the Versata Logic Server is not an inference rules engine. Typically, inference (or decision support) engines sit outside of the transactional system. They may be used to build expert systems or produce input to transactional components, but they do not directly implement the transactions contained in components such as EJBs.

The Versata Logic Studio is also not a Case tool. Case tools produce models and code “stubs” which are then implemented and integrated. Versata rules are executable and they need no further development (although they can be customized, as we see in Chapter 9, “Integrating the IBM Trade2 client” on page 151.)

The Versata Logic Suite is not a “4GL”. Fourth-generation languages shortcut procedural programming, but they are not declarative. A 4GL function will generally map, one-to-one, to a coded procedure. These functions do not unravel dependencies, sequence complex chains of operations, and map to logic across many entities.

Finally, although the Versata Logic Suite does construct Java components, it is much more than a code generator. It uses high level specifications to create and directly execute applications. In each case, a specification (the “what”) is input to the Versata Design Studio and is stored as exchangeable XML. From the specification, the system automatically parses, analyzes and creates the desired applications utilizing highly performing enterprise Java frameworks (the “how”.)

As we see during the next chapters, which detail the rules-enhanced Trade application, the Versata approach is particularly well-suited to WebSphere applications with substantial business logic, where application requirements are evolving or where project time, costs or EJB development skills may be an issue.





# Architecture of the Versata Logic Server within WebSphere

In Chapter 2, “Trade application overview” on page 9 and Chapter 3, “Business logic automation using rules” on page 19, we explained that the Versata Logic Server installs into the WebSphere Application Server to provide runtime services for rule-enabled business logic.

In this chapter we answer common questions about the Logic Server Architecture within WebSphere, its utilization of WebSphere services, and the exact nature of the business objects it creates and executes.

**Note:** At the time of this writing the Versata Logic Server was generally available on WebSphere version 3.5. A port was underway to WebSphere version 4.0. Terminology may differ slightly for WebSphere version 4.0.

## 4.1 What the Versata Logic Server is, and how it works

The most frequently asked question about the Versata Logic Server is simply, “What is it really?” In terms of J2EE components, this means “What is it comprised of, and how does it work?”

The Versata Logic Server is a loose term for a small set of sophisticated EJBs that run inside of the WebSphere Application Server. The EJBs provide libraries of Java classes (services) used by the business objects and applications created in the Versata Studio.

The two primary EJBs are the VLContext, used by the Versata Transaction Rules Engine, and the PLContext, used by the Versata Presentation Engine. When the Logic Server is installed, these stateful session EJBs are deployed automatically into a WebSphere EJB container. This is illustrated in Figure 4-1.

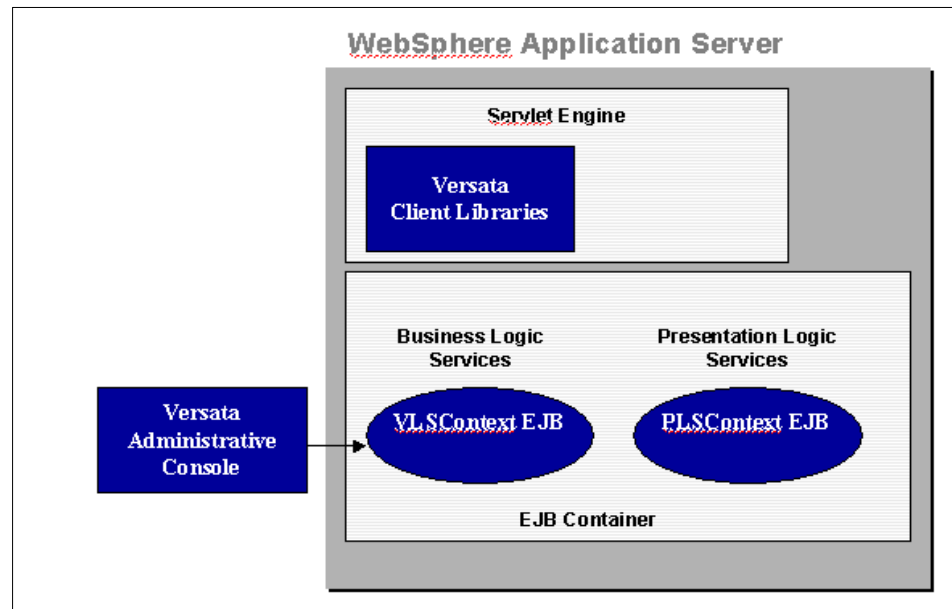


Figure 4-1 Versata Logic Server components created by installation

To assist in managing the collection of Versata resources, an instance of the WebSphere Application Server, called VERSATA, is created to house the EJB container. A servlet engine is also configured to execute Versata application servlets.

When a user connects to the Logic Server, a VLContext instance is created and is available for the duration of the session. When a user executes a Versata constructed application, a PLContext instance is also created.

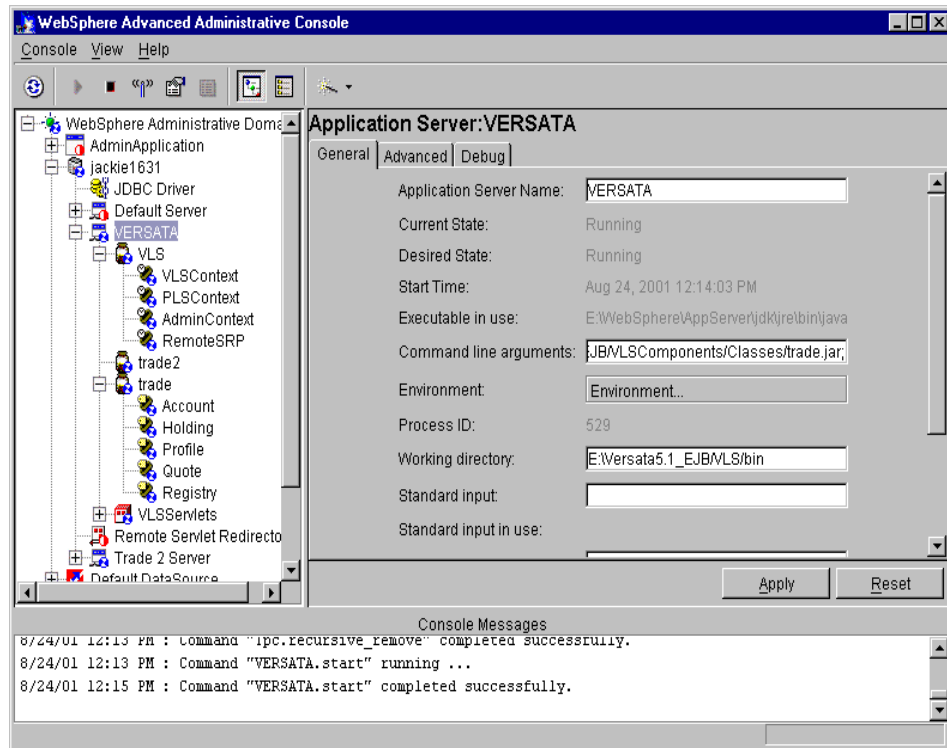


Figure 4-2 Versata Logic Server EJBs and servlet engine in WebSphere

## 4.2 Managing the Versata Logic Server in WebSphere

It is important to note that the Versata Logic Server has been designed to take advantage of the management and scalability strengths of the WebSphere Application Server. Here are some of the management characteristics of the Logic Server running within WebSphere:

- ▶ It is implemented with workload managed session EJBs. This allows multiple Versata Logic Servers to be replicated (cloned) and allows WebSphere to transparently distribute user load between Logic Servers, increasing throughput. Cloned copies of the Logic Server can be placed on the same physical system (vertical scaling), or on multiple distributed systems (horizontal scaling).

- ▶ It can use WebSphere security mechanisms. As we see in the next chapters, the Versata Logic Server controls access to its business objects using role-based authorization. Versata can use any WebSphere authentication mechanism to obtain validated user and role information. In this way, an organization can maintain a single directory of user data. Similarly, the Versata Logic Server can use sign-on information, passed from other applications, to grant access to business objects. This facilitates single user sign-on.
- ▶ It can be configured to provide redundancy. When configured with workload management software and WebSphere servlet redirection, service requests to a failed system can be directed to a Logic Server clone on a backup system. When the WebSphere administrative database and Versata role information is also replicated, this configuration eliminates a single point of failure.
- ▶ It can be configured to run behind a firewall. With WebSphere servlet redirection, the Versata Logic Server can be placed behind a firewall, as shown in Figure 4-3.

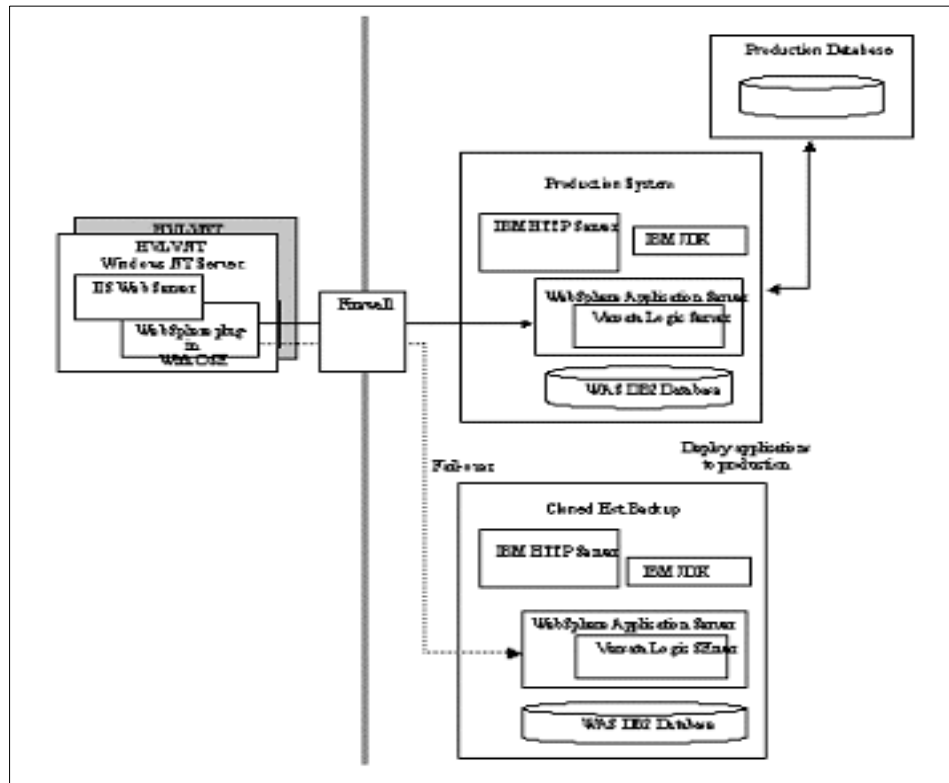


Figure 4-3 Firewall and hot backup configuration



## 4.3 Versata business objects

Before beginning development, it is also useful to understand the construction of the business objects managed by the Versata Logic Server. As we explained in Chapter 3, “Business logic automation using rules” on page 19, rules attach to data. Specifically, rules attach to business objects and their attributes.

Business objects in Versata are application-independent components that represent data and encapsulate the logic, or rules, need to carry out business processes. There are two types of business objects used in Versata:

- ▶ Data objects map to entities physically persisted to disk. A data object contains the set of attributes (both persistent and derived virtual attributes) to which rules are attached. Within WebSphere, data objects are deployed as entity EJBs as illustrated in Figure 4-4.

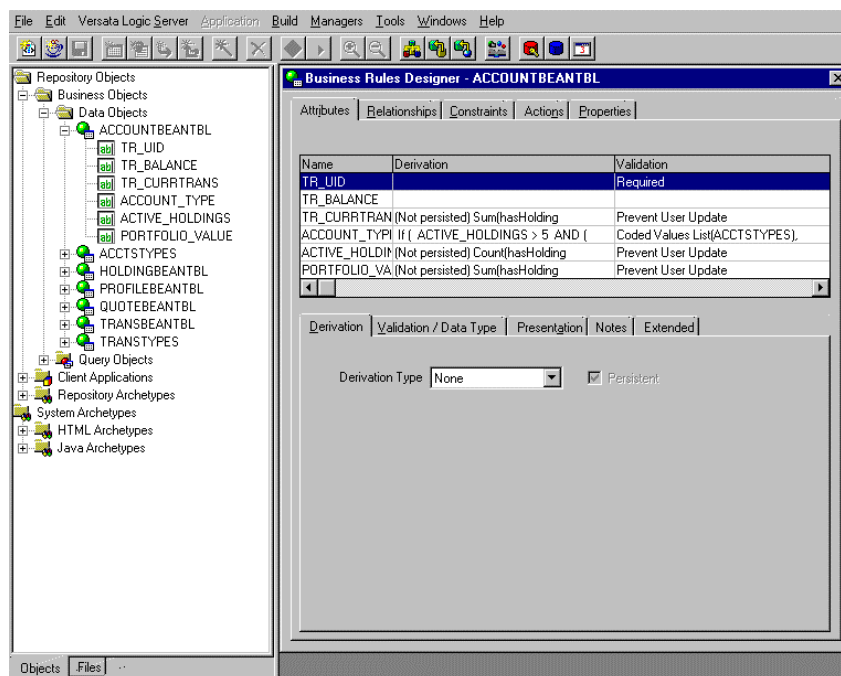


Figure 4-4 Data objects (left) and account attributes and rules (right)

- ▶ Query objects represent “views” of joined or restricted data objects. A query object provides an abstracted, reusable view of one or more data objects that protect client components from changes to the underlying data objects. Query objects can also have “virtual” attributes calculated at runtime. Within WebSphere, query objects are deployed as session EJBs and implements the J2EE pattern of aggregate or compound entities as shown in Figure 4-5.

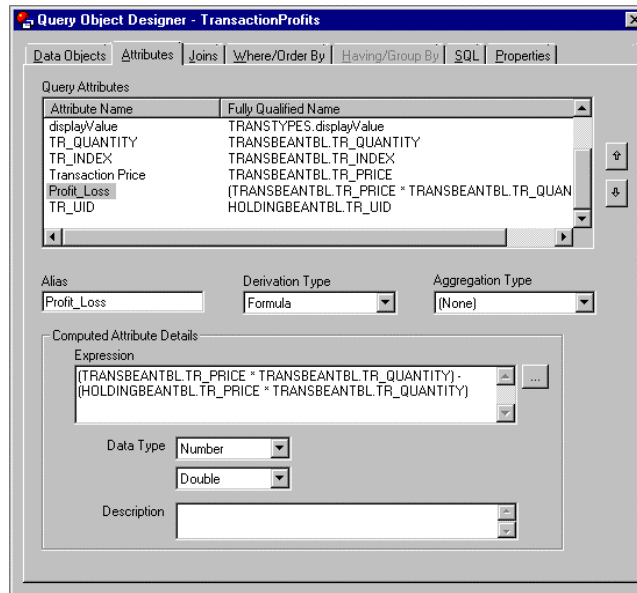


Figure 4-5 Query object in Trade shows Profit\_Loss attribute

## 4.4 Versata Logic Server classes

The Versata Logic Server class libraries provides both the construction framework and the runtime environment for business objects. Each business object will be created from a subclass of either a Versata DataObject or QueryObject Java class. In addition to normal EJB methods — create(), findByPrimaryKey() and so on — these objects will inherit extensive methods to:

- ▶ Execute ad-hoc queries
- ▶ Communicate with related objects
- ▶ Listen for and process rule events
- ▶ Manage the transaction cache
- ▶ Form transactions
- ▶ Communicate with the Versata Logic Server persistence layer

The mechanism for business object transactions and persistence is particularly interesting, as we now explain.

#### 4.4.1 Persistence as a layer in the server MVC

The Versata Logic Server has a layered architecture that can be thought of as its own, server-side “Model-View-Controller” as illustrated in Figure 4-6. The purpose of the Logic Server's MVC is to abstract business logic (the Controller) from the physical source of data (the Model), and to produce optimized “packets” of serialized data for display by the client-tier (View). This combination provides performance, extensibility, and portability.

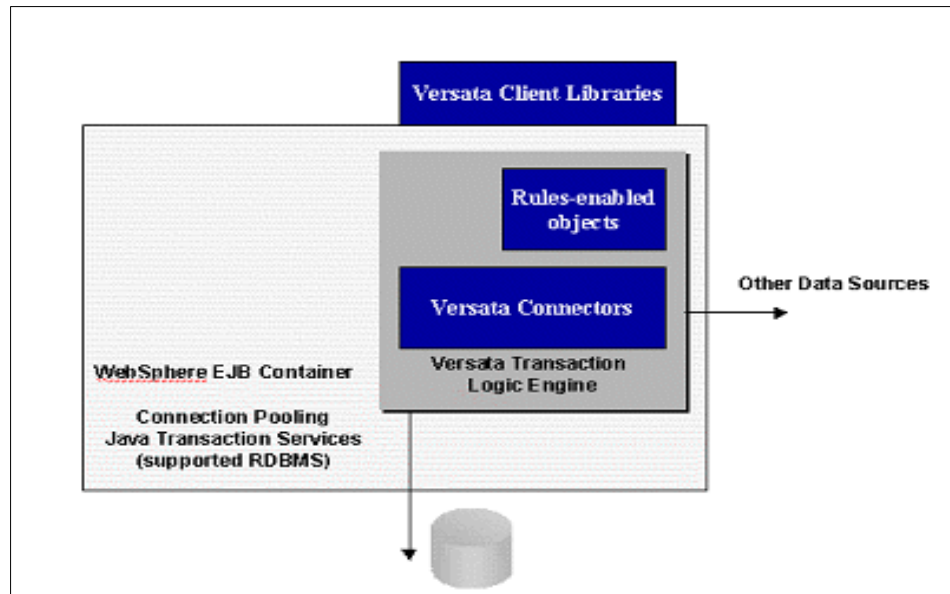


Figure 4-6 The MVC architecture of the Versata Logic Server

The Logic Server's controller layer consists of the rule-enabled objects, the View layer consists of the Versata Client APIs, and the Model layer consists of Versata Connectors. The capabilities of Versata's Client APIs and Connectors are explained later in this chapter. The capabilities of rule-enabled objects are covered in the following section.

## 4.4.2 Rule-enabled objects as WebSphere components

Up to this point, you might have noticed that we use the terms object, entity, and EJB almost interchangeably when referring to business objects. Here is a clarification on the types of WebSphere components used for rules-based business objects.

As we mentioned, Versata business objects are sub-classed from Java classes. One Java class is created for each object and contains its complete rule-defined behavior. These objects are under the management of the VLSEContext EJB (the business logic service EJB in the Versata Logic Server.)

When the set of business objects is deployed from the Versata Studio, the entire set of Java classes is deployed (as a JAR) file, and will be added to the CLASSPATH of the VLSEContext bean. During rule-execution, the Versata Logic Server uses fast, local calls to these class objects as shown in Figure 4-7.

In addition to being implemented as Java classes, business objects can also be deployed with EJB “faces”. Like a common J2EE blueprint from JavaSoft, Versata business object classes and their EJB faces implement an Entity Bean — Dependent Object pattern. Developers can automatically deploy these EJBs for any object whose methods may need to be referenced from an external component (such as a non-Versata EJB.)

**Note:** Object data can be accessed by external components through the client libraries, even if they are not deployed as EJBs.

The choice to create an EJB for an object is made on an object “property sheet” in the Versata Studio. For these objects, Versata constructs both the Home methods to obtain the objects (read from disk, instantiate, and return remote handle) and update them.

It is interesting to note that there is no rule or persistence performance penalty for simply deploying objects as EJBs, since the Logic Server will always execute the local implementation of the method, regardless of how it is invoked. There is however, the typical look-up and EJB-instantiation overhead if objects are accessed by their remote interfaces. For this reason, many Versata developers prefer to use the client libraries.

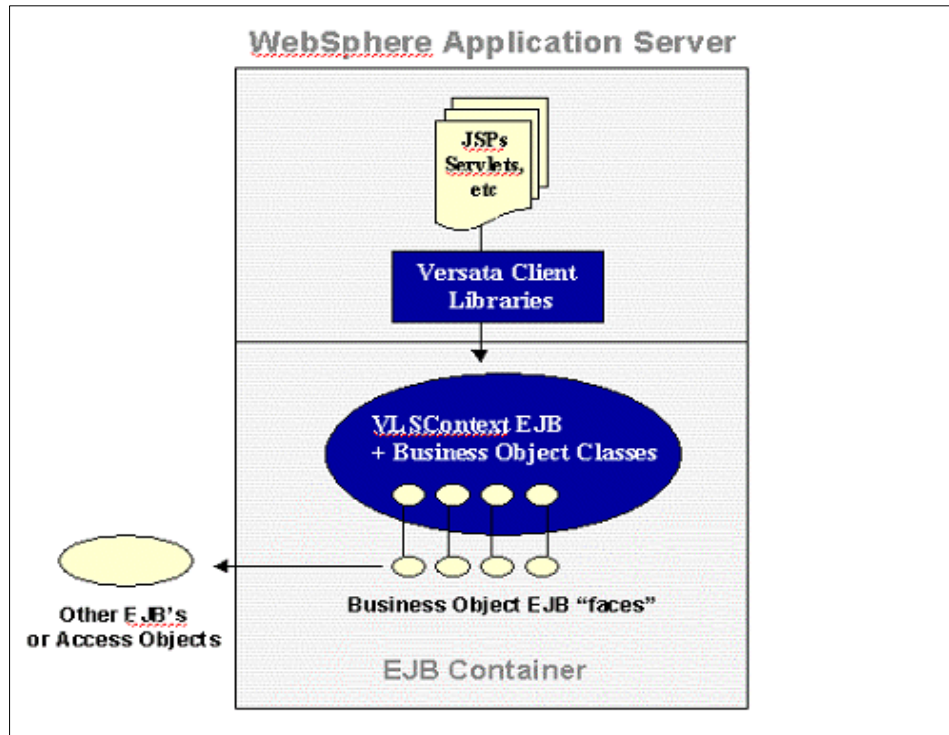


Figure 4-7 Business object deployed as local classes with EJB faces

#### 4.4.3 ResultSet access and just-in-time object instantiation

As we diagramed in the MVC discussion, Versata provides another method to access the business objects under its control — the Versata Client Libraries.

The Versata Client Libraries provide fast, ResultSet-based access to “rows” and “sets” of data objects. As with local rules processing, this avoids the overhead of remote object references. In addition, the Logic Server minimizes the use of shared server resources (memory) providing for “Just-in-time” instantiation of the objects.

## ResultSets of objects

Versata client libraries access the Versata Logic Server much like JDBC libraries access a database. A client establishes a “session” with the Logic Server, “connects” to a collection of business objects and issues “query” commands on either a Data Object or a Query Object. Commands can be used to retrieve, update, insert and save object changes.

Unlike JDBC, however, the commands and queries supported by the Versata libraries are independent of any database or physical storage. Instead, the libraries communicate to the Logic Server, who manages details of database or legacy data access through the “Connector tier” (explained below.)

There are several performance features built into Versata ResultSets. First, serialized values, instead of objects or handles, are returned to the client. Like the J2EE “ValueObject” pattern, this allows the client to access multiple attributes with a single call, copy the attributes to a local object, and operate on it without remote reference.

Unlike ValueObjects, however, ResultSets can return groups of serialized objects, further reducing overhead. This is similar to the “ValueObjectList” pattern, where the Logic Server acts as a general purpose “ValueObjectAssembler” for all of the business objects in the system. In addition, Versata ResultSets are updatable, extending the benefit of ValueObjects to all data accessed by the client.

To support this functionality, the Versata client library provides an optimized execution framework. The framework retrieves ResultSets into scrollable buffers. The size of the buffer is tunable (10 rows or 50 row buffers, for instance.) Clients can loop through ResultSets using first, last, next and previous methods. In addition, clients can insert rows and modify values in the ResultSet. When ResultSet modifications are complete, the set can be “saved” to send it to the Logic Server for processing. (A tunable optimistic locking mechanism preserves data source concurrency.) An illustration is shown in Figure 4-8.

ResultSet access to EJB-tier business objects is designed to provide the efficiency of JDBC with the logic encapsulation and reuse of EJBs.

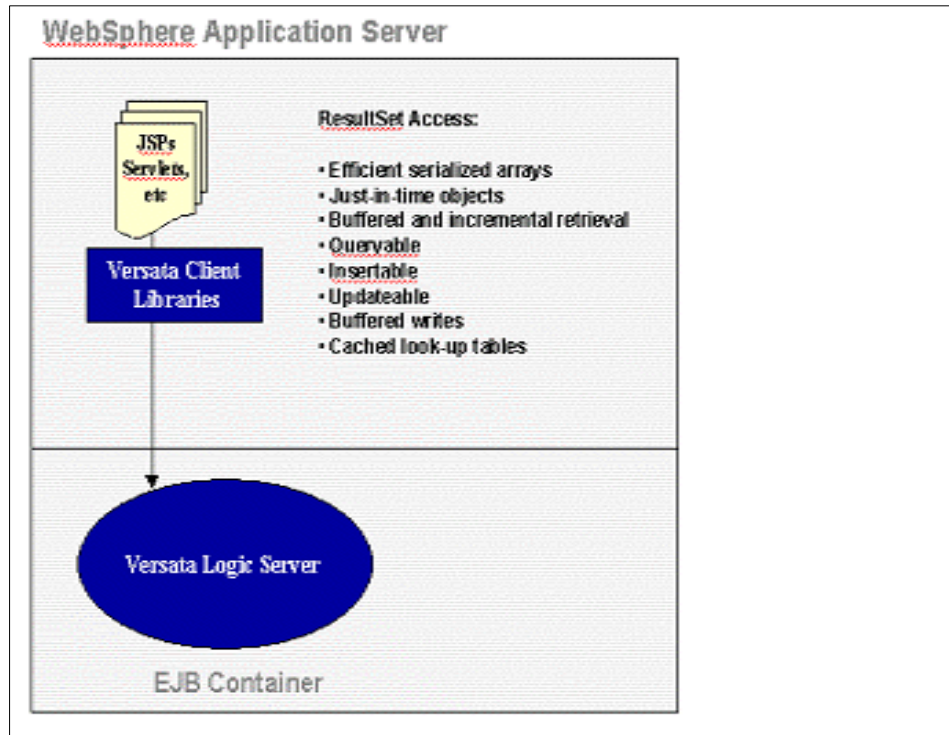


Figure 4-8 Value-based access through Versata client libraries

### Just-in-time objects

To conserve system resources, the Versata Logic Server does not create server objects for ResultSets. Instead, ResultSets are retrieved as serialized arrays directly through the Versata Connector.

Business objects in the Logic Server are instantiated at the just-in-time moment that the client completes its changes and issues a “save” method. At that time the Logic Server instantiates objects for all data that will be affected by the change. This technique is similar to the Java Pattern for “lazy object instantiation”, although the Logic Server framework for execution is more extensive.

As explained in our discussion of rules, all of the objects changed by the client or changed by a chain of rules triggered by the client operation are brought into a transaction cache to reduce database i/o. Rules ripple through the cache, evaluating and updating data as needed. A single object may be updated several times in the cache. When all rule operations have completed, the objects are written back through the Versata Connector to the data source.

Like sophisticated DBMS, the Logic Server supports implicit single-update transactions or Begin/Commit verbs to bundle several changes into a larger transaction. Even in the case of a single changed row there may be multiple object updates initiated by rules. One of the most productive features of the Logic Server is that such updates are automatically bound into the transaction, thus eliminating transaction bugs.

## Connector layer

As mentioned above, Client ResultSets are retrieved directly through the Connector Layer and transactions were written through Connectors. Here is an explanation of the Logic Server persistence mechanism that uses this layer.

One of the design goals of the Logic Server is to decouple rule-constructed transactions from any concern for the data's physical persistence mechanism. Since rules operate on data from relational databases (which may be supported by EJB Container Managed Persistence) and non-relational data sources (which are not currently supported by CMP), persistence must be managed directly by the Logic Server.

Recall that the Logic Server framework is implemented as an EJB, and that business objects, as Java classes, are managed by this EJB. The Logic Server manages the persistence of these Java classes through the VLContext bean. It does this by calling the Connector mapped to the business object in the Versata Console. (The Console is an administration interface for a running Logic Server.)

The connector implements object persistence for each class of objects, such as objects persisted to DB2, or to Oracle, or through MQSeries. Where possible, the connector uses the transaction and connection pooling services provided by the EJB container (using the Java Transaction Services API, JTS, from WebSphere, for instance). For data sources not supported by the EJB container, the connector implements its own, similar functionality.

Similarly, where WebSphere supports two-phase-commit protocol (2PC), the connector will also support 2PC. This is done using the JTA ability to assure that a single call is made for starting transactions (rather than using a separate call for each connection, which would introduce the opportunity for errors). This API requires that the EJB code “register” each connection on behalf of the user thread. This enables the BeginTrans call to communicate with each connection. With the Versata Logic Server, this occurs as database connections from the WebSphere pool are obtained for a specified thread.

As with transaction management, 2PC can occur automatically where the WebSphere application server supports it.

**Note:** For WebSphere 3.5 this includes DB2 and Oracle.



The connector architecture of the Logic Server offers a high-degree of convenience and flexibility. As with container managed persistence, developers do not need to program transaction boundaries or database details. In addition, developers do not need to specify transaction attributes or database details in EJB deployment descriptors. The connector used by a data object can be modified during runtime, without re-deploying the object.

The Versata Logic server is shipped with a number of out-of-the box Connectors, including one for each of the major relational databases. In addition, Versata provides a standard interface definition and a set of transactional APIs for use in developing customized Connectors.

## 4.5 Looking to the future: EJB 2.0 and JCA

We conclude the discussion of persistence and connectors with a look at two upcoming Java standards — the EJB 2.0 specification and the Java Connection Architecture (JCA). Both proposed standards overlap some of the functionality now provided by the Versata Logic Server.

One of the first things we notice is that the Versata architecture has many things in common with EJB 2.0 and JCA. This is probably not a coincidence, since both Versata and the new J2EE specifications were designed to solve the same problems surrounding distributed, heterogeneous applications. The similarities should make it straightforward for Versata to adapt the Logic Server to use new EJB and JCA functionality. In fact, Versata has announced support for both standards as they become available for WebSphere.

It must be especially noted that when migrating between EJB 1.1 and EJB 2.0, Versata customers may have a big advantage over most Java developers, because rules-based automation abstracts away the implementation of EJB relationships and persistence — the two big areas addressed in the new specification. This may turn out to be one of the most significant advantages of developing with a high level framework such as Versata.

To leverage new standards in hand coded components, development teams often face fundamental re-writes of their applications. For instance, EJB 2.0 EJBs and EJB 1.1 EJBs may not be mixed in the same container. When using a higher level approach such as Versata's, however, an existing set of business rules can be used to automatically produce a completely new set of integrated components that comply with the changed specification. This transfers the migration burden from the developer to the vendor. It also removes the possibility of introducing new bugs into existing code.

Here we look at EJB 2.0 and JCA to understand their overlap and migration with Versata.

### **4.5.1 EJB 2.0: Container Managed Relationships (CMR)**

The biggest difference between the EJB 1.1 and 2.0 specification is the significant change to Container Managed Persistence, especially support for Container Managed Relationships (CMR) and faster EJB access (through local interfaces).

Container Managed Relationships allow an EJB container to maintain associations between container-managed entity beans. The relationships are defined in the XML-descriptors of the EJBs and are implemented within the bean with coordinating get and set methods for each logical field. The get method on the “many” side of a one-to-many relationship is implemented with a Java collection and iterated over when traversing the relationship.

The EJB container maintains basic referential integrity. For instance, in the case of an Account with many Holdings, the container can automatically delete the Holdings for a deleted account, and can ensure that a related Account exists before inserting a Holding.

There are some differences between the new CMR and Versata's current relationship rules. For instance, Versata relationships are automatically bi-directional. In addition, there are several more enforcement options: child objects can be automatically changed when parent objects change (their primary and foreign keys will be automatically updated), parent deletions can be prevented (rather than just cascaded), and so on.

The biggest value that Versata may offer when it migrates to the new CMR scheme is in automatically coordinating the data model, the EJB implementation and the deployment descriptors. CMR requires a high-degree of synchronization between the logic implemented in an EJB (names and parameters of abstract methods, for instance), and that placed in deployment descriptors (which must be tied to EJB logical fields). These must be further coordinated with the exact implementation of the get methods on each side of the relationship which differ depending on whether a single object (one-to-one relationship) or a collection of objects (one-to-many relationship) is being defined.

This degree of synchronization between data modeling, EJB development and sophisticated deployment makes a strong case for automating the production of these artifacts as a result of higher level rules — the approach taken by Versata.

## 4.5.2 EJB 2.0: local interfaces

Another significant change in EJB 2.0 is the introduction of local interfaces. Local interfaces allow EJBs within the same JVM to communicate with simple Java calls and pass data by reference, rather than by value. This is designed to address a major performance shortcoming in EJB 1.1 and is almost identical to the approach now used by Versata (although the APIs differ and will be adjusted by Versata).

In EJB 2.0, developers have the choice of either developing local EJB interfaces (which inherit from a new `EJBLocalHome`) or providing remote interfaces (which continue to inherit from `EJBHome`). A developer who wants to optimize local access while also allowing for remote access, will need to create and coordinate two sets of interfaces. Versata simplifies this process by creating local and remote interfaces automatically. In addition, it automatically synchronizes changes in an object's remote interface with changes to the local object.

## 4.5.3 Java Connector Architecture (JCA)

The other area of overlap between the current Versata Logic Server and the up-coming Java specifications is the Java Connector Architecture (JCA). JCA provides a Common Client Interface (CCI) that provides access from J2EE clients, such as enterprise beans, JavaServer Pages, and servlets, to an underlying enterprise information systems.

When implemented through application servers such as WebSphere, JCA will take over part of the role now performed by the Versata Data Access layer. Specifically, it will allow Versata applications to utilize adapters provided by 3rd parties, instead of calling hand-coded Connectors written to Versata's Data Connector API.

Versata is eager to expand connectivity using the JCA and says that future versions of the Logic Server will fully exploit the standard. We do note, however, that the current JCA specification (version 1.0) lacks support for metadata, XML, and asynchronous communication. (The JCA 2.0 draft specification is working to address these.) Until then, Versata support for these features will still be useful.

## 4.5.4 Other J2EE standards used by the logic server

Throughout this chapter we have alluded to a number of other places where J2EE standards apply to the Versata Logic Server. For instance, the WebSphere servlet redirector “finds” a Logic Server (Java Naming and Directory Service, or JNDI). Also, the remote servlet “communicates” with the Logic Server (RMI over IIOP).

Figure 4-9 shows a complete picture of the interaction of the Logic Server with WebSphere and other components.

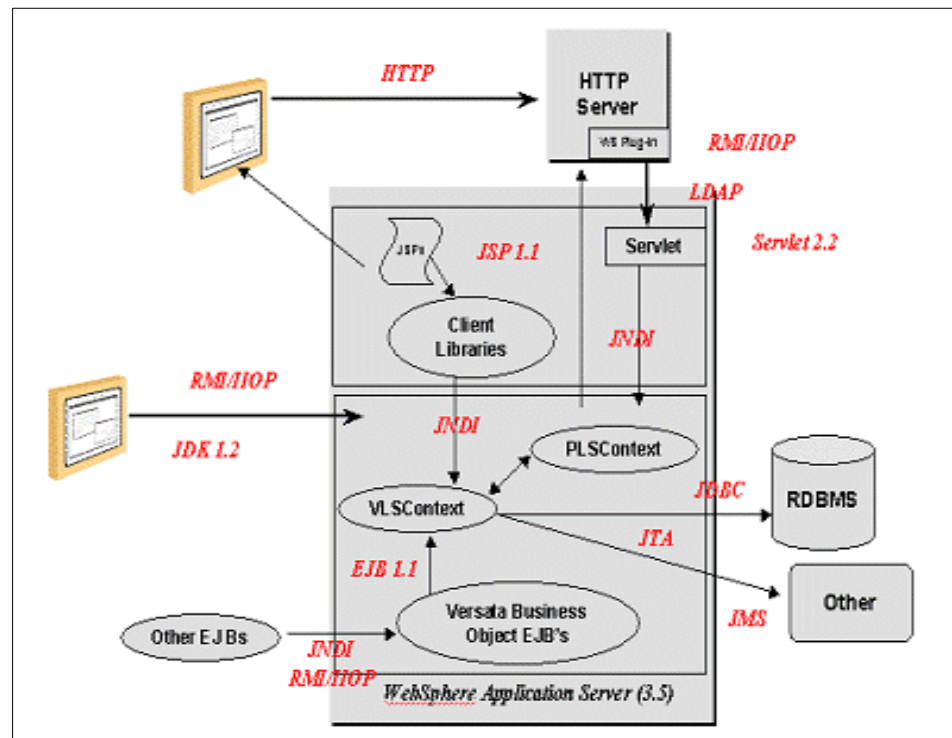


Figure 4-9 Standards used for WebSphere version 3.5

#### 4.5.5 Recap of the Versata logic services

This chapter concludes with a review of the services provided by the Versata Logic Server, specifically those provided by the Transaction Rules Engine, separate from any specific client tier. They are generally higher level services than those provided by the J2EE application server and can be thought of in the following way:

J2EE services free the developer from routine infrastructure programming to concentrate on business logic, while Versata services free the developer from routine business logic programming to concentrate on just those issues, like connectivity to legacy systems, that are unique to his environment.

Most of these services have already been mentioned. Here we will highlight them before moving on to develop the Trade application.

## Four service objectives

Versata services can be grouped around four objectives:

1. Executing high-level, declarative assertions about a domain of business entities, where those assertions (rules) were defined without regard to sequence or dependencies
2. Providing a framework that allows the developer to customize and extend constructed logic and preserves his extensions through repeated development iterations
3. Enabling fast, convenient access by client applications for all business functions (ad-hoc query as well as object update.)
4. Optimizing the performance and persistence of inter-object logic, regardless of the data source

The Versata Libraries, used by the VLSCContext EJB at runtime, provide most of these services. Detailing the Versata Libraries is beyond the scope of this redbook, however, we can examine some of the classes for examples of their capabilities.

### ***For the logic server EJB itself:***

- ▶ Establish and destroy sessions to the Logic Server, maintain session context
- ▶ Limit number of connections per logic server (before redirecting to another server)
- ▶ Maintain and report statistics on connections
- ▶ Establish and maintain connection pools to data sources
- ▶ Trace and report rules execution
- ▶ Grant fine-grained authorization to business objects, including distinct authorizations to update (versus insert), and to read specific attributes.
- ▶ Manage the transaction cache

### ***For data objects:***

- ▶ Create, destroy data objects
- ▶ Process queries using a datasource-neutral syntax (that is, masks whether the source is SQL or not). Queries can use any attribute or combination of attributes and may join objects
- ▶ Explicitly get and set any attribute of any object
- ▶ Find related objects
- ▶ Get and set attribute of any related object by relationship navigation (no find method required)

- ▶ Listen to and respond to client or server “save” events
- ▶ Maintain before and after data values for use in rules
- ▶ Listen and respond to rule events
- ▶ Apply defaults, performs calculations, adjust values, check constraint and guarantee referential integrity
- ▶ Listen and respond to user defined events
- ▶ Call external methods
- ▶ Formulate commit and rollback transactions
- ▶ Maintain object metadata (attribute data types, lengths, defaults, formatting, captions, validations, and so forth)
- ▶ Provide object metadata to any function that calls for it.

***For query objects:***

- ▶ Map attributes to underlying data objects (may be from multiple sources)
- ▶ Derive virtual attributes
- ▶ Overload names, captions and other metadata
- ▶ Coordinate save operations to multiple data objects (manage parent and child data objects)

***For connectors:***

- ▶ Marshall data from data source to serialized arrays used by the client
- ▶ Translate datasource-neutral commands to the syntax required by the data source
- ▶ Create the database or other connection needed to access the datasource

***For client libraries:***

- ▶ Provide ResultSets as arrays to client applications
- ▶ Manage buffers, providing first, next, previous and last behavior
- ▶ Incrementally retrieve rows as buffers are emptied
- ▶ Maintain Old and New values of attributes
- ▶ Provide cached data validation lists to client applications
- ▶ Allow index array access to rows and attributes in the ResultSet
- ▶ Support insert, delete of rows in the ResultSet
- ▶ Access business object metadata (captions, update permissions, formatting, and so forth)
- ▶ Save changed ResultSets to the Logic Server



## Rule-based development

In this chapter we cover the development of the business logic for an application with the same basic functionality as the Trade Application we outlined in Chapter 2, “Trade application overview” on page 9. We call this new application TradeX (Trade eXtended with Rules). TradeX logic supports:

- ▶ Logging a user onto the system
- ▶ Updating a user profile
- ▶ Viewing a portfolio (a set of holdings)
- ▶ Finding the quoted price for a stock
- ▶ Buying and selling holdings

## 5.1 Introducing Versata Studio Business Logic Designer

In regard to the original Trade application, our current implementation has two main points of deviation:

- ▶ First, the Trade application includes several functions to configure its multiple runtime modes (EJB, JDBC, etc.). No special functions are built into the TradeX application to support these modes. We assume two basic modes of access:
  - Versata-constructed clients, which need no special set up
  - Existing Trade client using Versata client-libraries, which will be explicitly setup.
- ▶ Second, the Trade application provides for simple user login with minimal security. User IDs are kept in the Trade database and passed as parameters to servlet operations. Default Versata security, on the other hand, authenticates users from an encrypted directory of users (or from other authentication services supported by WebSphere). The Logic Server grants attribute-level access for operations such as “read”, “update”, and “insert” to authenticated users by their role. For TradeX, in the next several chapters, we will use Versata's default security mechanism. When porting the Trade client to Versata we will resume using Trade's original login scheme.

The Versata Studio is a high-level design environment for specifying, building, and deploying business logic components (using the Versata Business Logic Designer) and, optionally, HTML and Java applications (using the Versata Application Designer).

This chapter covers the Business Logic Designer. Chapter 6 covers the Application Designer.

### 5.1.1 Project approaches and roles

Although this redbook presents the development of the business logic tier and the client presentation tier as two distinct processes, this is often not the case in practice.

Versata projects sometimes use a rapid, iterative development approach (RID). The ability to quickly specify and deploy partial business logic, and the ability to incrementally add to that logic through the addition of new rules, allows an approach where GUI development can proceed in parallel with business logic development. The system is reviewed frequently by users to refine requirements.



In addition, Versata projects often include business users and analysts as development team members. When business users understand the rule-based approach they can produce requirements that match Versata rules.

In addition to business analysts, Versata projects usually involve:

- ▶ **Application developers:** These are familiar with Java, COBOL or a 4GL, but not necessarily to the system level. In addition to developing with rules, Versata provides an event handler model for most customizations. This model should be easy-to-use by anyone familiar with a higher-level tool or language. Programmers familiar with VisualBasic, JSPs, JavaScript, or other client-tier “scriptlets” should find it easy to customize Versata-automated systems.
- ▶ **Web technology specialists:** These will usually assist in setting up Web servers, firewalls, etc. In addition, depending on the method of client-tier development, Versata-automated or developed with other Web development tools, these developers may use a Web development environment such as WebSphere Studio. Alternatively, they may integrate other content management systems such as Vignette.
- ▶ **Systems Architects:** These will usually specify the end-to-end solution architecture and be responsible for technology choices. System architects ensure that new systems interoperate with existing software, comply with corporate computing standards. Architects frequently trade-off the real world practicalities of development with industry trends and best practices.
- ▶ **Systems Level programmers:** These are familiar with enterprise integration techniques and system level Java. Although Versata reduces the need for system-level Java, in most cases experienced Java developers can add enormous value to the team. Java developers can extend Versata Class Libraries to create new rule types. Extended rules, customized for the organizations' own business processes, can be leveraged by the entire development team. Examples of such rules may include rules that access legacy systems, rules that invoke external components, etc. In addition, Java programmers familiar with VisualAge for Java can assist in tracing and debugging the end-to-end execution of the system, through the servlets, JSPs, and Versata Client Libraries to the Logic Server, Connectors and custom components.

It is beyond the scope of this redbook to recommend a particular team development approach. Versata does, however, issue “Best Practice” guides and training courses which address pre-requisites recommended for team members, the use of third party source code control with Versata, and the integration of rules-based development into other well-known methodologies such as the Rational Unified Process.

## 5.1.2 Versata repository

The repository is the basic organizer of a Versata-designed system. The repository is a collection of eXtended Markup Language (XML) files that store the metadata for all business objects and associated rules for a specific object model. For team development, repositories are typically put under source-code control. Any file-based control system can be used.

The Versata Logic Server can service any number of repositories. Each will be deployed in a unique JAR file to the Logic Server.

**Note:** With a knowledge of Versata's XML rule-schema formats, it is possible to populate a repository (enter rules and exchange rules, etc.) without using the Studio Rules Designer. This use is beyond the scope of the Redbook.

A repository is initially opened as a named empty container. There are a several methods you can use to populate the repository with initial object definitions:

- ▶ Object definitions can be imported from a UML model (like Rational Rose).
- ▶ Object definitions can be imported from database schema.
- ▶ Objects can be defined using Versata Studio wizards.
- ▶ Objects can be imported from an existing XML file.

## 5.2 Step 1: Importing an object model

TradeX development begins by opening a new repository, as shown in Figure 5-1, and re-engineering the existing Trade DB2 database. After importing the DB2 schema, you can view the Data Objects created, as shown in Figure 5-2.

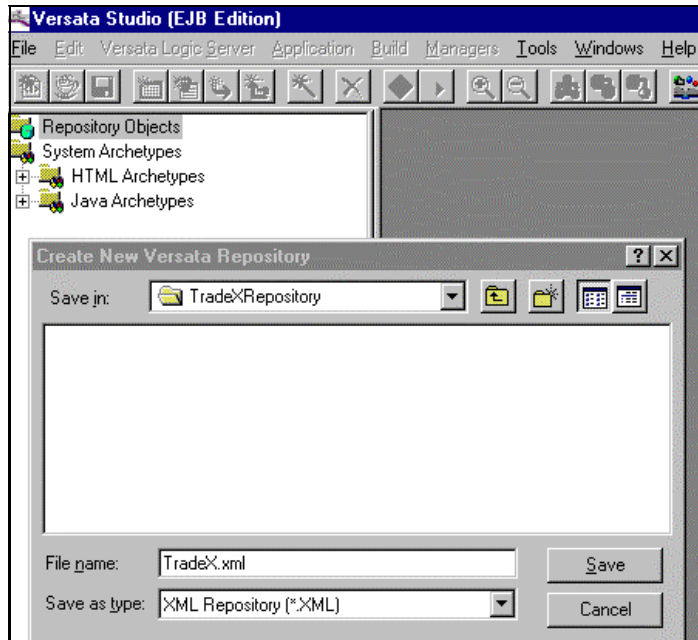


Figure 5-1 Create a new repository

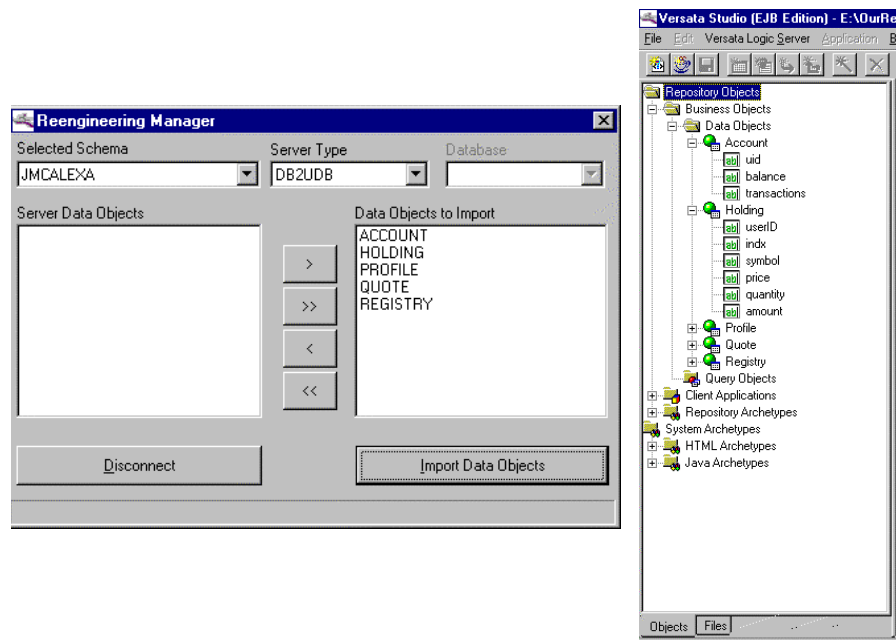


Figure 5-2 Import DB2 schema and view data objects and attributes

## 5.3 Step 2: Adding relationship rules

For TradeX, there are several potential relationships:

1. Accounts have Profiles (one to one)
2. Accounts have Holdings (one to many)
3. Holdings have Quotes (many to one)

Defining these relationships as rules will enable the Logic Server to easily navigate between related objects for rules processing.

A few decisions must be made when defining relationships; for example:

1. On which attributes will the objects be joined?
2. Which is the parent object (one) and which is the child (many)?
3. What referential integrity should be enforced?

The following steps take us through defining a relationship between an Account and its Holdings:

- First, add the relationship, define the parent object, and define attributes for their join, as shown in Figure 5-3.

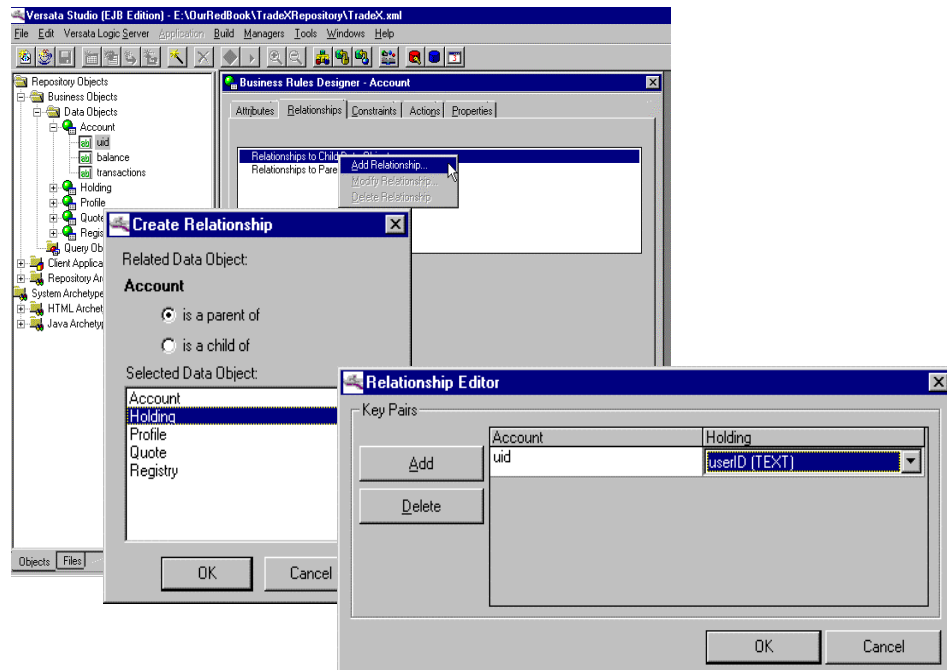


Figure 5-3 Creating the Account/Holding relationship

- ▶ Next, enter the details.

Here we chose to enforce the following integrity:

1. We cannot delete an Account object if it has associated Holdings.
2. We cannot insert a Holding object if it has no Account.
3. If the userID in the Account object is changed, then change its associated holdings to retain their relationship.

In addition, we have named the relationships. The system will use this name for Java methods that navigate the relationship as shown in Figure 5-4.

- ▶ Then, we create a standard error message if the relationship is violated. The Logic Server will provide this error message through the client libraries.

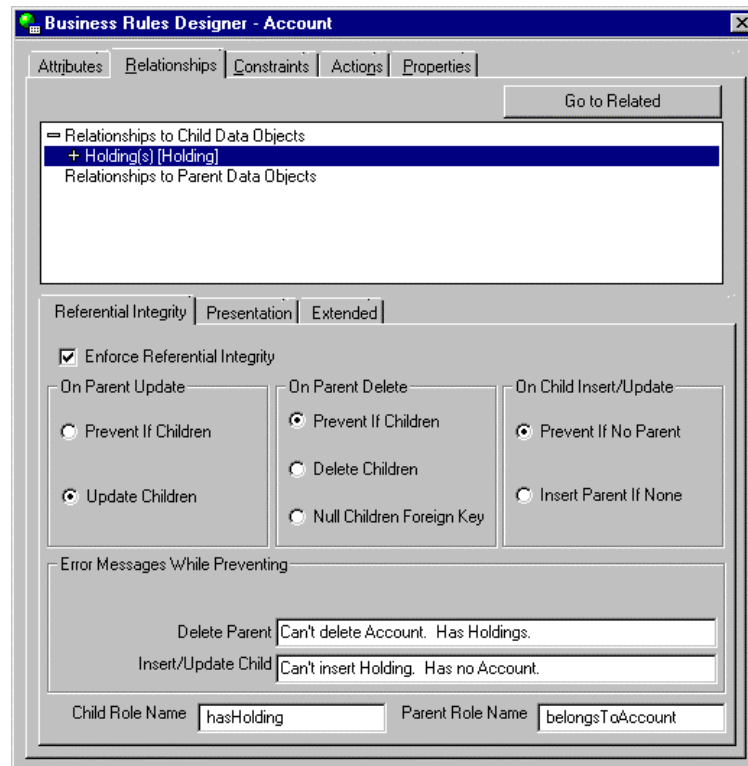


Figure 5-4 Completing the Relationship rule

- ▶ Finally, the remaining relationships are defined in the same steps.

## 5.4 Step 3: Identifying additional rules

Typically, identifying potential rules begins in the requirements analysis phase of a Versata project. Many development teams find it useful to extract potential rules from “use cases”. Because the Trade application already has well defined functionality, and because the initial business logic in those functions is rather limited, we will identify some potential rules here, just before they are created. We do this by mapping a functional requirement to its rule equivalent.

We will highlight some suggestions and return to the details when enhancing TradeX with additional functionality in Chapter 9, “Integrating the IBM Trade2 client” on page 151.

The first suggestion is to understand that there is a top-down progression:

- ▶ From Business functions — which Versata defines as transactions that may involve multiple steps
- ▶ To Requirements — defined as one of the steps in the transaction
- ▶ To Rules — the declarations about data at that step

Let's take the Trade business function: Buy a specified stock. The initiating action will be a call to insert a new Holding for this userID and stock symbol.

**Note:** The action may be initiated by the client through the Versata client Libraries or through a EJB interfaces. Versata-automated clients create the insert automatically as seen in Chapter 7, “Deploying the TradeX application” on page 103.

The functions are:

1. Obtain the current price for the stock (attribute: Holding.price)
2. Obtain a unique identifier (primary key) for the holding (attribute: Holding.indx)
3. Calculate the amount of the Transaction (Holding.price \* Holding.quantity)
4. Update the number of transactions for the Account (Account.transactions)
5. Debit the Account balance with the transaction amount (Account.balance)

Let us identify the rules as we defined them in the Rules Designer:

1. Specify how to get the current price. This is done through a derivation rule called a “parent replicate”. A Parent Replicate Rule allows us to go to a related parent object (Quote), get an attribute (price) and replicate it in this attribute (Holding.price) at runtime.

In the rules designer, the steps are illustrated in Figure 5-5.

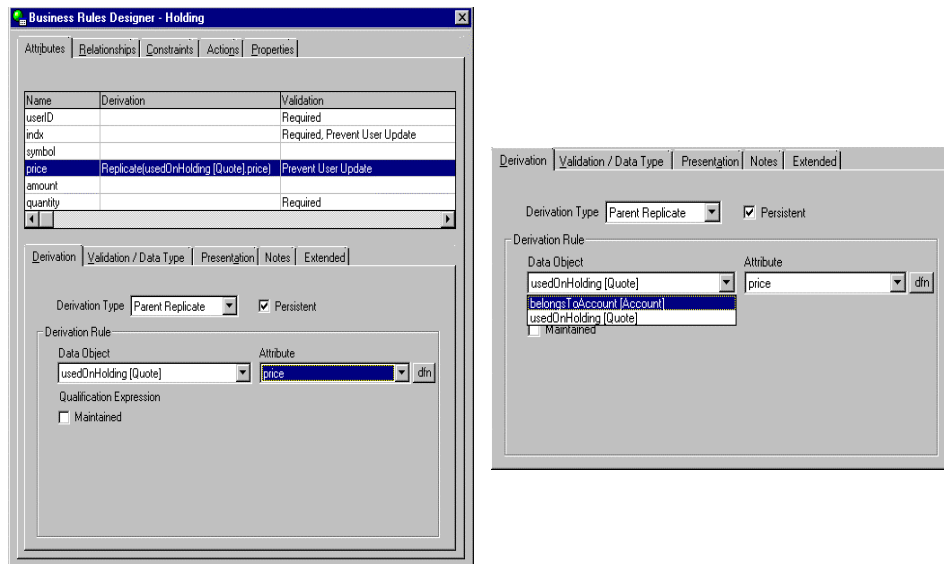


Figure 5-5 The rule to get price from quote object

## 2. Get a unique key for the Holding.

In the original Trade application, a cached block of serially numbered keys in the client-tier is created and a new one is passed to each Buy function. The key is used to create a unique identifier for a holding and is stored as the `indx` attribute. The last key used in a block is stored in an entity EJB.

In our first pass through TradeX, we will implement the serial key function differently. Instead of using Trade's EJB-to-cache function, we will use a built-in Versata capability — the autonumber datatype. By defining the `indx` attribute as autonumber, Versata Logic Server will be responsible for obtaining a unique, serial key for the `indx` attribute. It does this using the underlying database's serial datatype.

Figure 5-6 shows how an attribute's data type is assigned. Note that the `indx` attribute is mandatory. If the system cannot assign a key, an error will be produced. We have supplied an error message that the Logic Server will give to the client application.

In addition, we have marked that the attribute is required and cannot be updated by the user. This metadata will also be passed to the client. When Versata constructs a client (using the Versata Presentation Designer), HTML forms will be constructed using this metadata. If the developer provides his own client, this metadata can be used to drive client behavior.

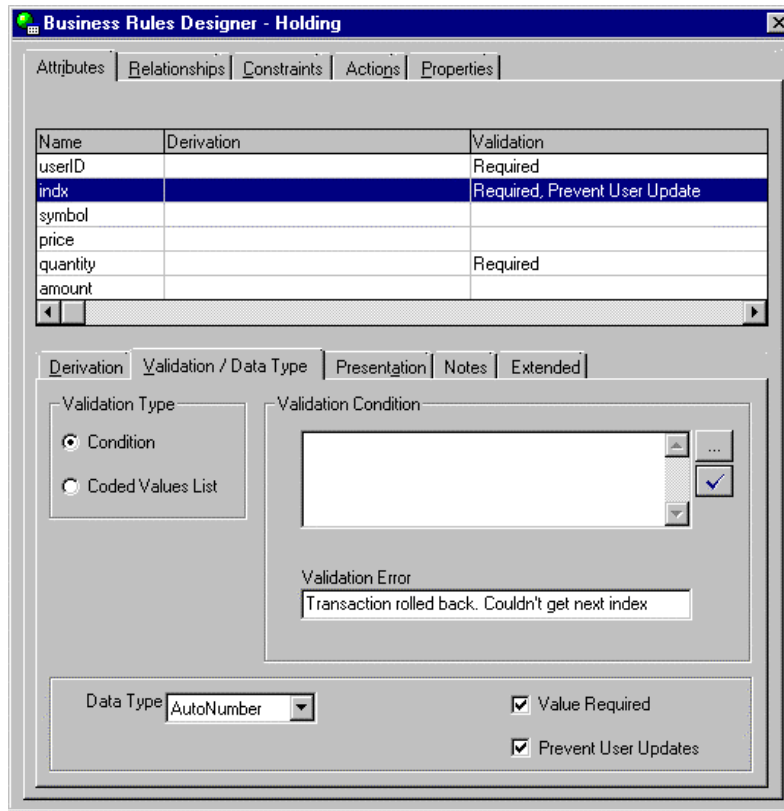


Figure 5-6 Unique key autonumbered

3. Calculate the transaction amount. The Trade database does not have an attribute for the transaction amount. Amount is calculated at runtime and not persisted.

Versata supports non-persisted variables. We simply create a new attribute and unmark its "Persisted" property.

Rules can be defined for the runtime attributes just like persisted attributes. Here we define a "formula" derivation rule to calculate the transaction amount.

The logic to calculate amount is: normally, the amount of the holding equals the quantity purchased \* the price of a share (which when the holding is inserted, is replicated from the Quote object.) However, when selling (deleting) a holding, the current share price must be retrieved from the quote object.



Therefore, the rule to calculate amount says:

```
If ( Deleting )
Then
    $value = (getusedOnHolding().getprice()) * quantity
Else
    $value = price * quantity
End If
```

This is illustrated in Figure 5-7.

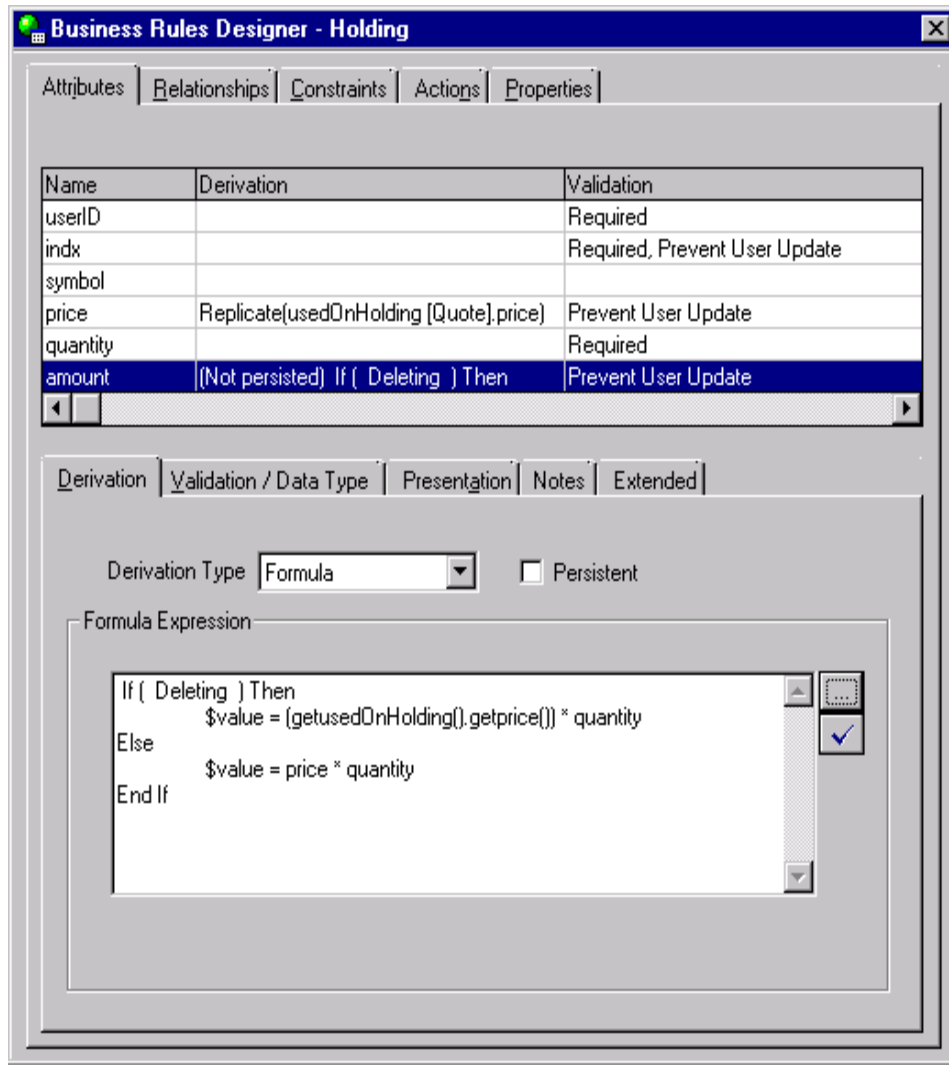


Figure 5-7 Creating a non-persisted attribute for the transaction amount

4. Update the number of transactions for the Account (Account.transactions).

This rule relies on the Logic Server to understand that there are transactions flowing through the system with the potential to update an Account object. The business rules compiler will understand this potential. (The change will be triggered by our next rule that updates the Account balance.)

This rule also relies on the Logic Server to maintain old and new values of attributes before transactions are committed.

The rule in Figure 5-8 says, if this is a new account (account is being inserted), then the transaction count is zero. Else, if the Balance is changing, increment the transaction count.

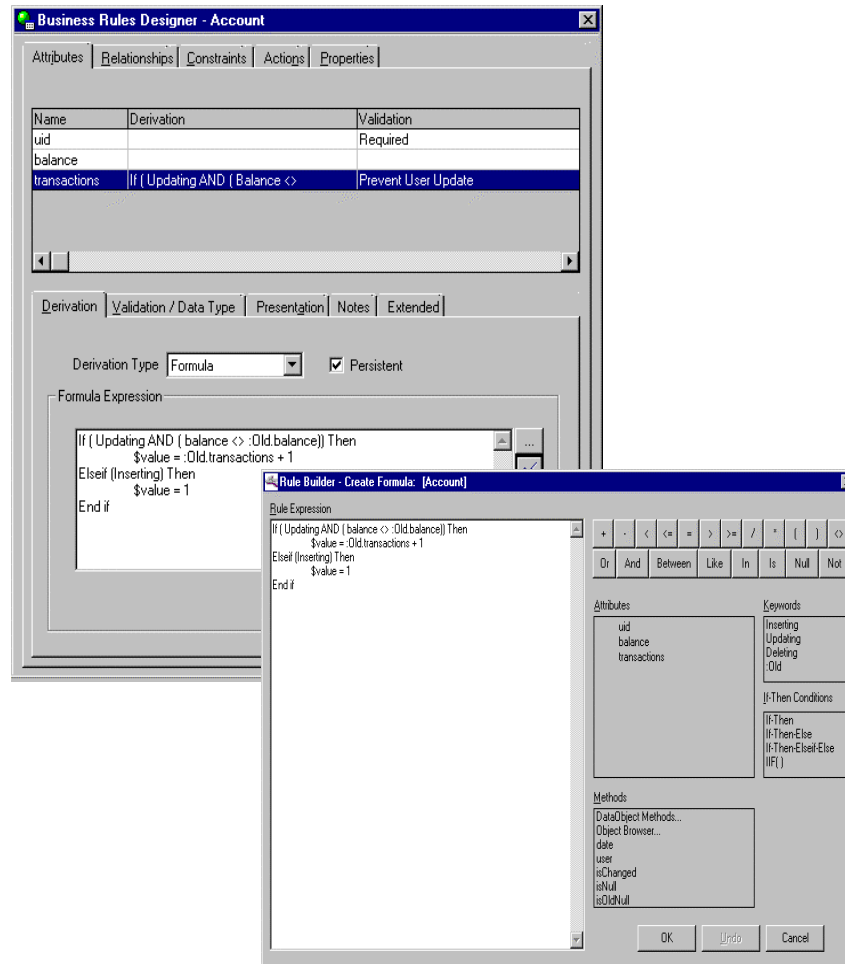


Figure 5-8 Rule builder is used to "point-and-click" the rule

5. Debit the Account balance with the transaction amount.

There are several rule patterns for explicitly updating related object “when” another an event occurs. A completely non-procedural option is to create a non-persisted transaction amount in the Account object that copies (sums) the transaction amount from Holding and adjusts the balance with it automatically.

We will show a more procedural approach that is better understood by Java programmers. It involves creating a “debit” method for the Account entity to allow it to debit its balance attribute. We will call this method from an event/condition/action rule on the Holding entity as shown in Figure 5-9.

The first step in creating a method is to open the Java class that has been created by the system from the rules already entered. This is done from the “Files” tab of the Versata Studio. The main file is the AccountImpl.java file. (You will notice that the EJB artifacts Account.java and AccountHome.java for the Remote and Home interfaces have also been created.

The Versata Studio contains a small Java editor for entering most business object customizations. (Developers familiar with VisualAge for Java may want to import objects into that environment for extensive customizations.)

The Versata Studio Code Editor prevents changes to the “protected” part of the object's source. Here we add the debit method to the end of the file. (At the same time, we enter a credit method to use when selling stocks.)

Customizations to an object's source code are preserved, even when the object is re-created. (Recreation happens when the business rules compiler is called.) Each new repository build will extract customizations, rebuild changed objects and replace the customizations automatically. This enables rapid and error-free development iterations.

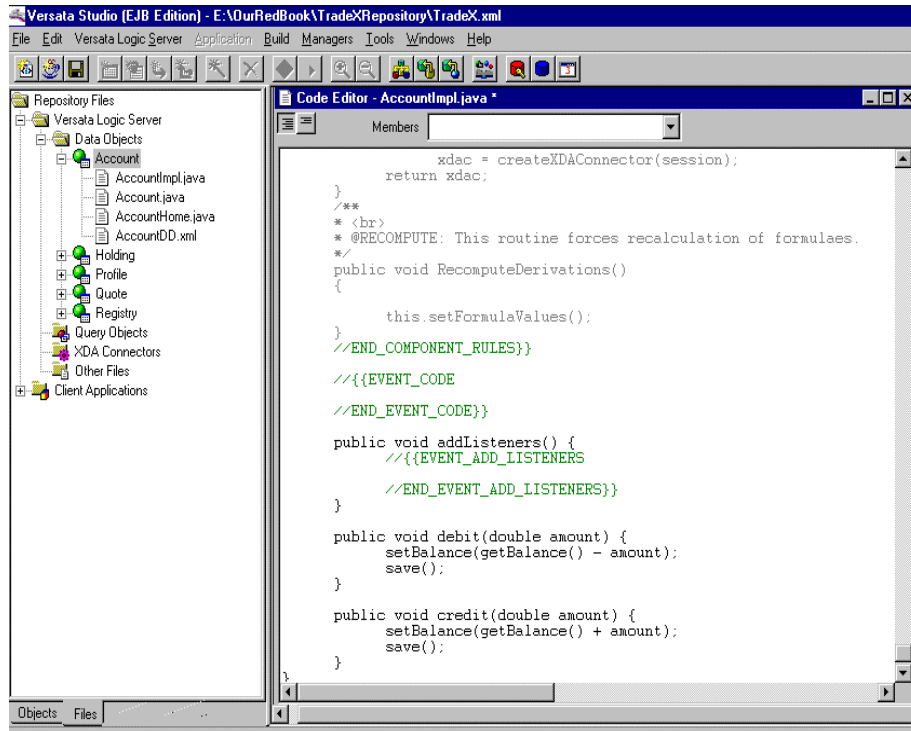


Figure 5-9 Debit and credit methods are added to the Account Java source

The final step in updating the account balance is the event/condition/action rule on the Holding entity. There is one rule for “Buy” which occurs when holdings are inserted. There is another rule for “Sell” which occurs when holdings are deleted.

The Buy rule is, if inserting, get the related account (this is the relationship name we defined with the relationship rule between Holding and Account), then invoke that Account's debit method for this amount. Simple as that!

Figure 5-10 shows the window for defining event/condition/action rules. Note the pop-up that allows browsing for the relationship navigation method.

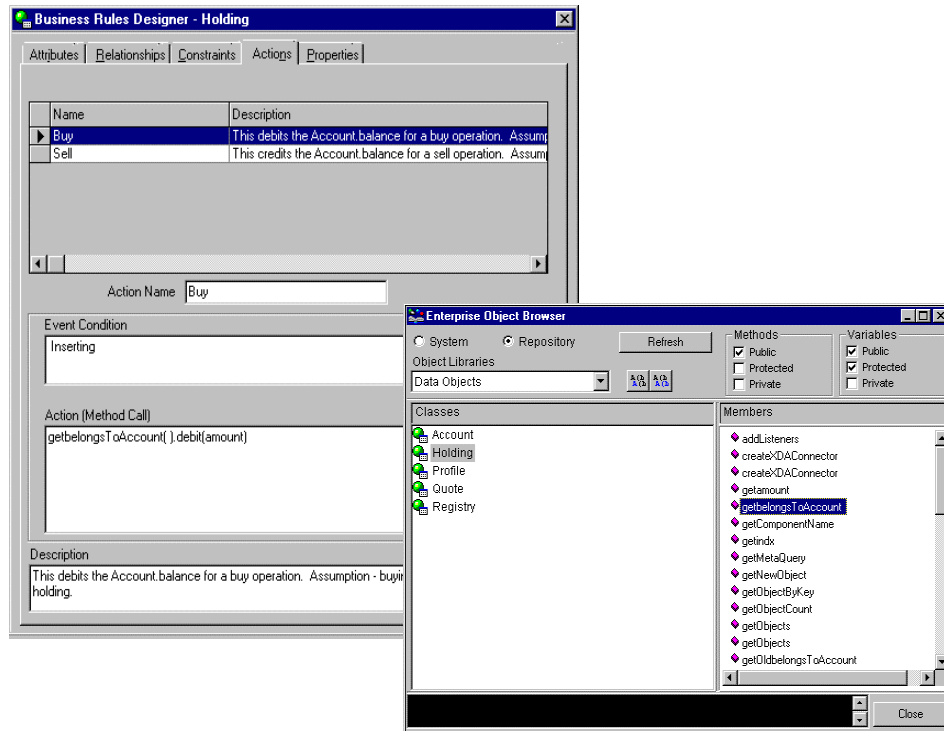


Figure 5-10 Defining the action to debit the user's Account

**Note:** The simple business logic of the original Trade application makes it possible to translate business functions into insertions and deletions from of Holdings. In Chapter 9, "Integrating the IBM Trade2 client" on page 151, we will implement a more likely business scenario that includes a separate transaction entity. Transactions will occur against holdings, allowing for the partial sale of a holding.

## 5.5 Review of the steps

With those three steps we have completed the EJB-tier of our application. Let's review the steps before moving on to deployment:

1. Import the initial data model from DB2.
2. Define relationships associating Holding to Quote and Holding to Account.
3. Identity requirements and enter their rules

The requirements, followed by their rule were:

**Requirement:** Obtain the current price for the stock (attribute: Holding.price).

**Rule:** Define a parent replicate rule on price, getting the price from the related Quote.

**Requirement:** Obtain a unique identifier for the Holding (attribute: Holding.indx).

**Rule:** Use the autonumber data type for the indx. The system will automatically create and assign a sequential number.

**Requirement:** Calculate the amount of the Transaction

**Rule:** Define a new, non-persisted attribute "amount" and derive it as price \* quantity.

**Requirement:** Update the count of transactions for the Account (Account.transactions)

**Rule:** Define a rule that when an account object is "touched" the system will evaluate whether the change is an insert or update operation. The object has been inserted operation (new account), the transaction count will be zero. If the object has been updated, and if the balance changed, then the count of transactions will be one more than it was.

**Requirement:** When buying, debit the Account balance with the transaction amount of the Holding. When selling, credit the Account balance with the transaction amount of the Holding.

**Rule:** Define the debit and credit method on the Account object. When inserting or deleting a holding, navigate to the associate account (by relationship) and debit or credit the balance.



## Designing an HTML client application

In Chapter 5, “Rule-based development” on page 57, we described the creation of the business logic tier of the TradeX application using the Versata Transaction Rules Designer.

In this chapter we explain the creation of an HTML application using the Versata Presentation Designer.

## 6.1 Versata Presentation Designer

The Versata Presentation Designer is a modeling environment where application pages, page elements, and transitions (links between pages) are specified. It can be used to specify both HTML and Java applets applications.

In the Designer, wizards are used to gather information about application objects, such as what data will be presented, how it is to be presented, and the flow of the application. A drag-and-drop pallet shows the overall application model, with the application flow diagram on the right, and the list of application pages and their elements on the left, as shown in Figure 6-1.

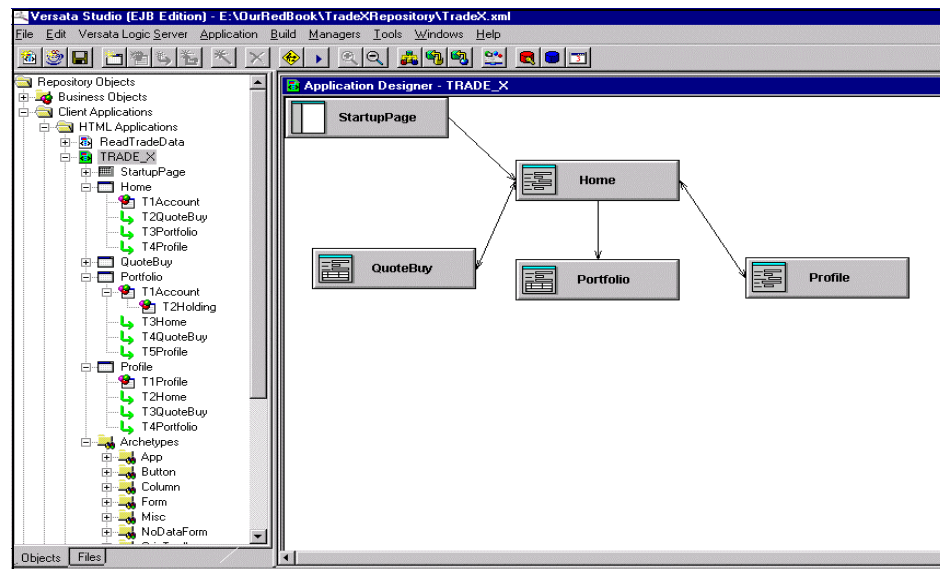


Figure 6-1 The TradeX application model

The Application Designer illustration in Figure 6-1 shows the completed model for the TradeX application. These are the application pages:

- ▶ **Home:** Shows basic account information.
- ▶ **QuoteBuy:** Allows the user to obtain a quote and buy stock.
- ▶ **Portfolio:** Shows the user's holdings and allows the user to sell holdings.
- ▶ **Profile:** Shows detailed information for the account.

The Presentation Designer generates application elements from “archetypes” (templates) which are part of the Versata system library. For HTML applications, archetypes are primarily composed of HTML code and specialized Versata tags (macro code) which allow the pages to be generated and processed by the Versata Presentation Engine.



Just as the Transaction Logic Designer creates a Java class for each business object defined to Versata, the Presentation Designer creates a Java class to process each HTML page defined to the system. These Java classes are included in the class path of the PLSContext, which, as we saw in Chapter 4, “Architecture of the Versata Logic Server within WebSphere” on page 39, is installed into WebSphere.

The Versata Presentation Designer automatically creates an architecture, Model-View-Controller, similar to the architecture we saw with the IBM Trade client. The HTML pages comprise the “view”. A combination of an application servlet plus the Java classes which drive the pages comprise the “controller”. The business logic tier comprises the “model”. This is shown in Figure 6-2.

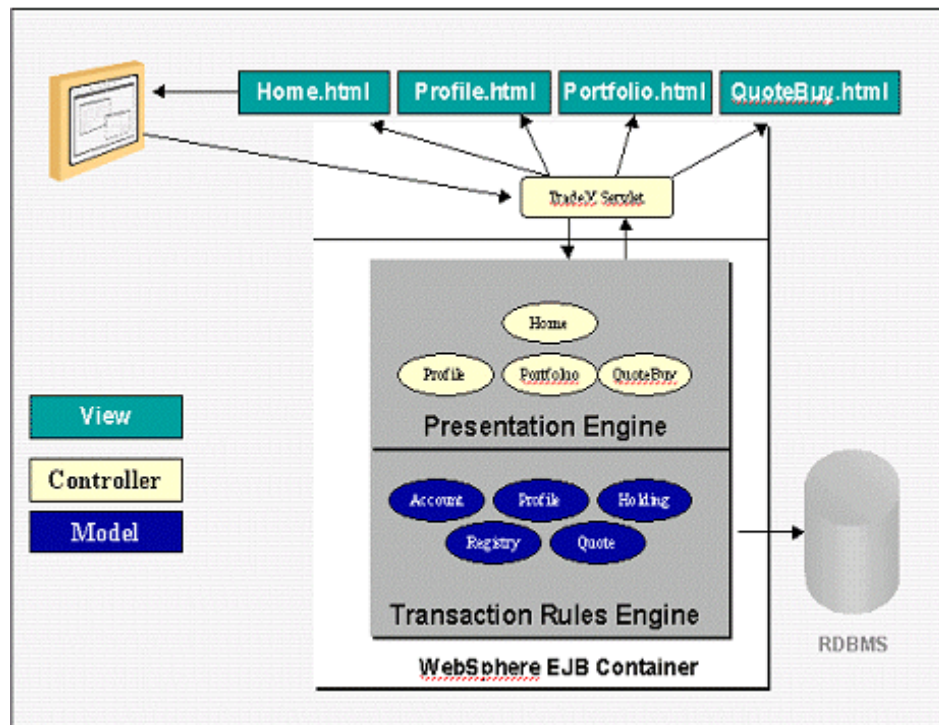


Figure 6-2 MVC architecture of the Presentation Engine

The primary difference between the IBM Trade client architecture, reviewed in Chapter 2, “Trade application overview” on page 9, and the Versata Presentation Engine architecture, is Versata's use of Java classes instead of Java Server Pages to control page behavior. (Versata's presentation architecture pre-dates the JSP specification.)

**Note:** Options for those who want to use JSPs will use the Versata JSP Toolkit or Versata Client-libraries instead of the Versata Presentation Designer and Engine.

## 6.2 Overview of the completed application

Before beginning to design the TradeX client tier, we review what the completed application will look like. Later in this chapter, as we design each application element, you will probably want to refer back to these completed pages.

### 6.2.1 Login page

The Login page is shown in Figure 6-3.

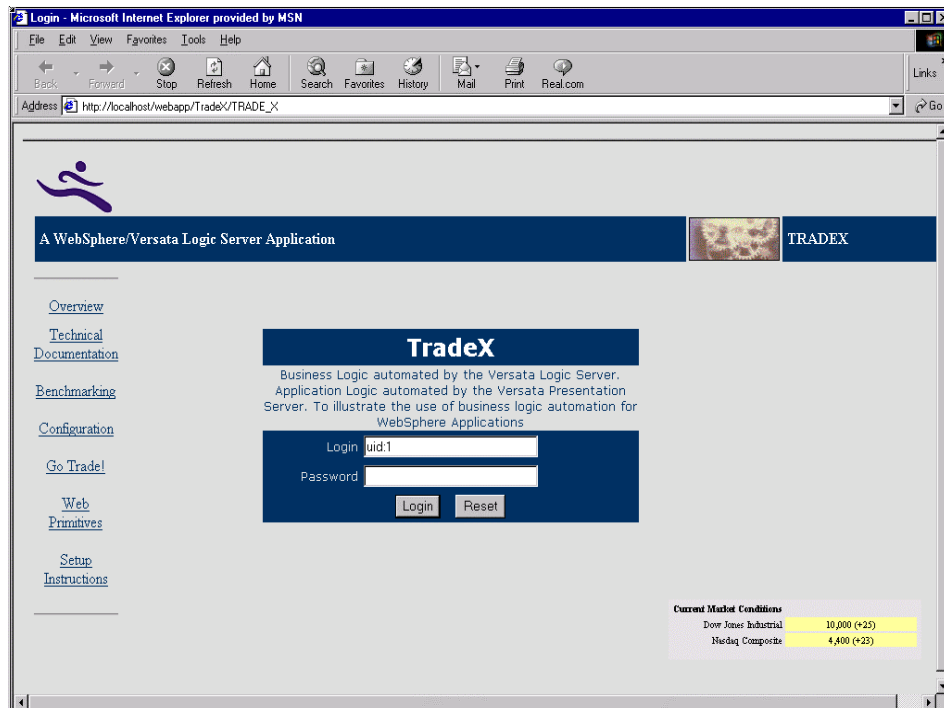


Figure 6-3 TradeX login page

By default, the application servlet will present a Login page for each new user session. The Login page connects a user, via his userID and password, to the Versata Logic Server.

As we have outlined, the Transaction Rules Engine within the Versata Logic Server controls a set of business objects defined to Versata. The TradeX application will use the Versata client libraries to establish a connection (session) with the Logic Server. Both the Login.html page and the application servlet are automatically generated by Versata.

The TradeX Login page includes a Versata standard login form, and top, left and bottom “banners” that have been customized in an HTML editor. The Login page is the only page that requires editing to display these banners. Subsequent pages in the TradeX application will be instructed to display these banners by setting page properties in the Presentation Design tool. (We will see this in the development process.)

**Note:** The mechanism for verifying the user name and password can be specified in the Versata Console. Choices include using any authentication mechanism supported by WebSphere (such as the local operating system or an LDAP service) or using Versata's default user registry. In this version of the TradeX application, we will use Versata's default security configuration.

## 6.2.2 Home page

The Home page is shown in Figure 6-4.

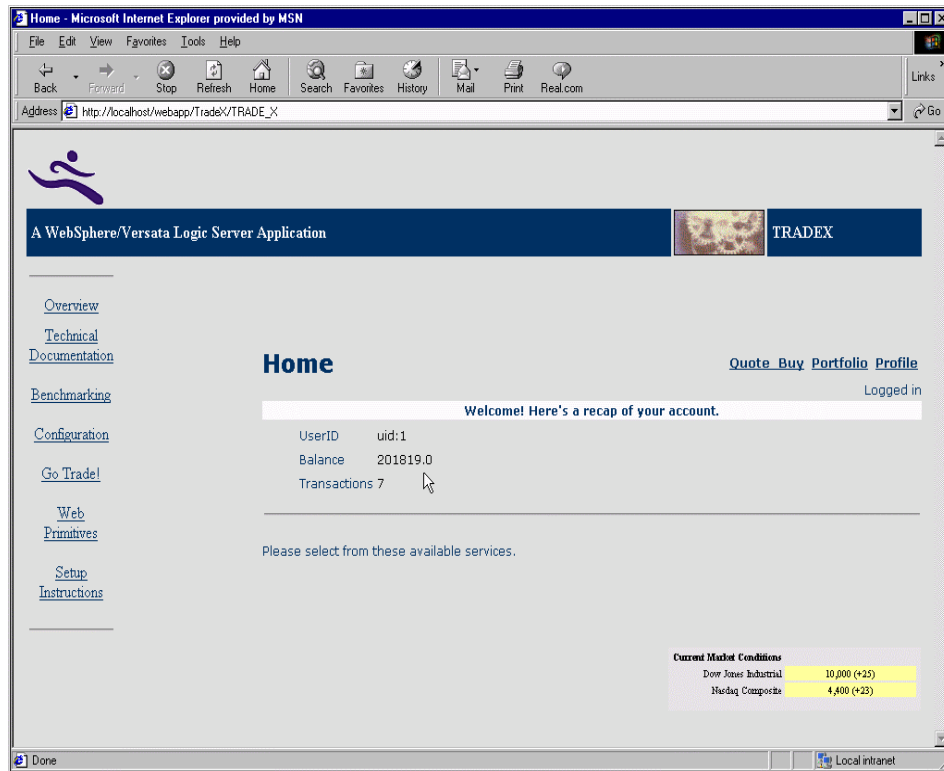


Figure 6-4 TradeX home page

The Home page contains information from the Account business object and the transitions (smart “links”) to the other pages: QuoteBuy, Portfolio and Profile. It includes the same banners as the Login page.

## 6.2.3 QuoteBuy page

The QuoteBuy page is shown in Figure 6-5.

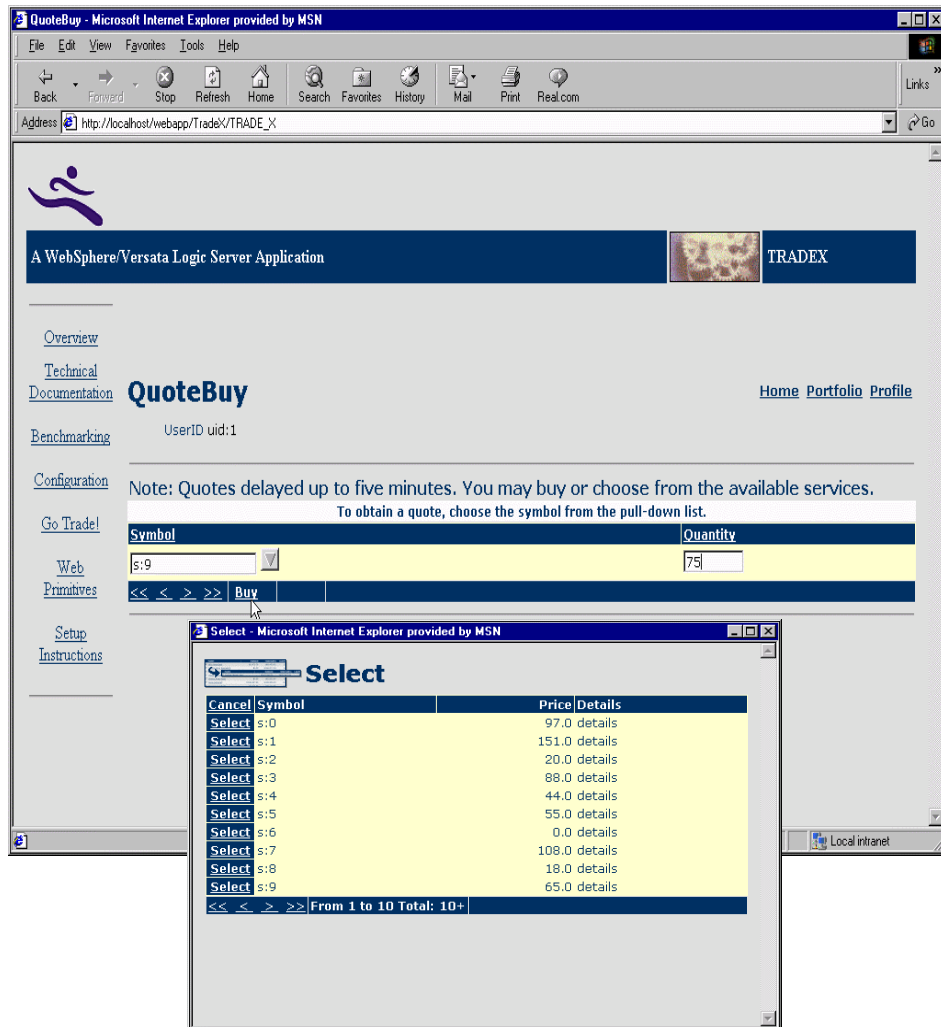


Figure 6-5 QuoteBuy page and Quote Pick List

The QuoteBuy page contains one attribute from the Account object (the UserID at the top) and one row from the Holding object (ready for insertion.)

Before inserting a holding, the user can click the “down-arrow” control next to the Symbol box. This will display a list of stock symbols with their prices. The “<” and “>” controls allow navigation forward and backward through the list of stocks.

From this list, the user can select a stock and the symbol will be copied into the QuoteBuy grid. To complete the transaction, the user enters a Quantity and clicks **Buy**.

## 6.2.4 Portfolio page

The Portfolio page is shown in Figure 6-6.



Figure 6-6 Portfolio page

The Portfolio page contains information from the Account object (the UserID at the top) and a grid (table) of Holdings associated with the Account. Some things to note about this default grid are:

- ▶ Rows may be selected using the radio button on the left of the grid.
- ▶ Clicking **Sell** will sell the chosen row.
- ▶ The grid automatically displays the number of rows found (“Total: 4”) and being displayed (“From 1 to 4”).

- ▶ The grid is sortable by any column. This is done by clicking the column heading (**Symbol**, **Price**, etc.)

## 6.2.5 Profile page

The Profile page is shown in Figure 6-7.

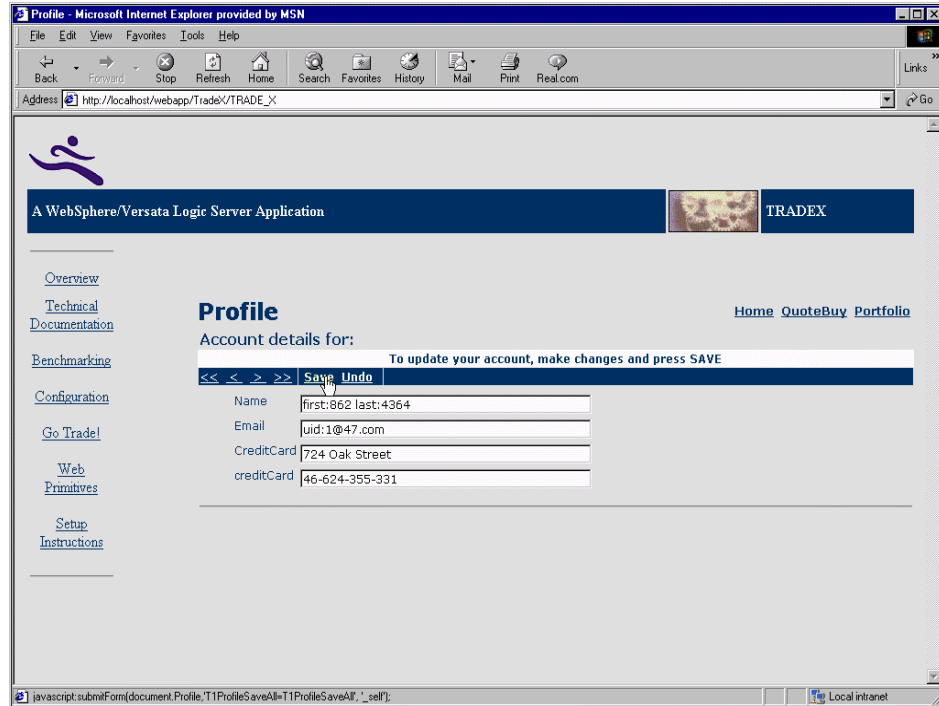


Figure 6-7 Profile page

The Profile page contains data from the Profile object. Users can change information and “Save” the changes. Because we chose the “buffered” save mode for this data source, the user is free to move around the fields to making several changes before they are sent, in a single statement, to the Versata Logic Server for processing. Before choosing to **Save** the changes, the user is free to back them out using the **Undo** button. Other Save modes for application data will be examined as we build the application.

## 6.3 Beginning application design

The definitions of applications are kept in the Versata Repository, along with the business object model and business logic rules. An unlimited number of Java and HTML applications can re-use the same business logic components.

### 6.3.1 Choosing the application style

A new application is begun by in the application designer by selecting the application **Style**. The style determines which set of application object archetypes will be used, as shown in Figure 6-8.

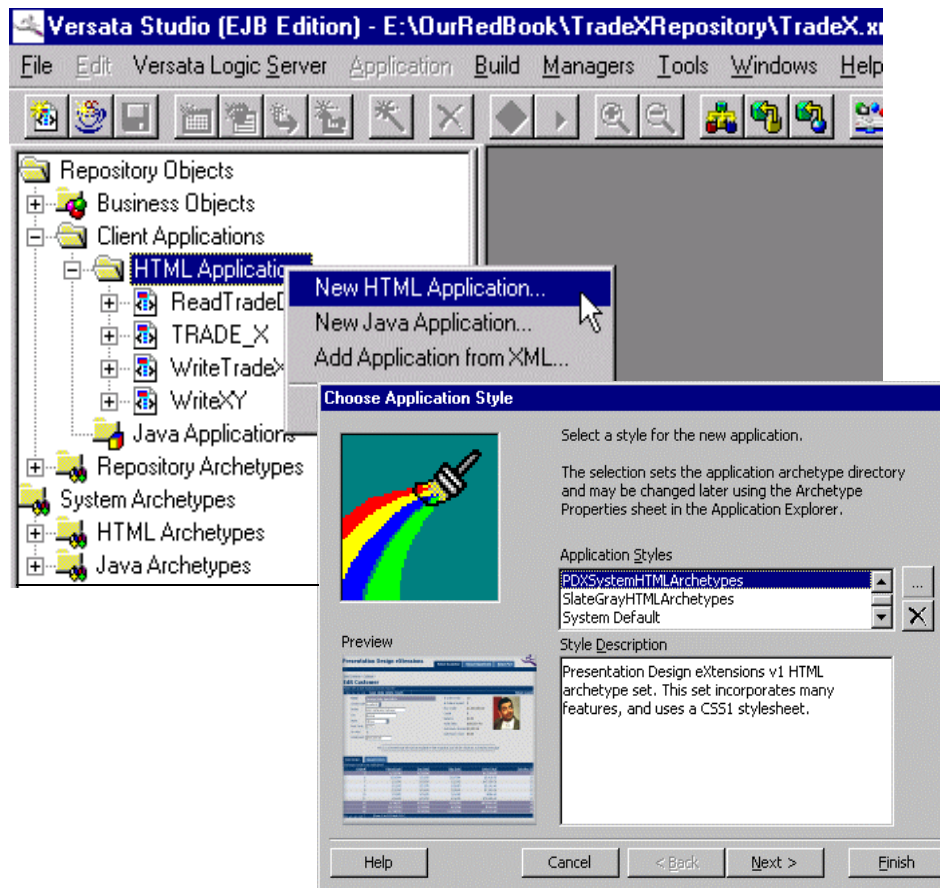


Figure 6-8 New application using a style



Automating application construction from a set of archetypes speeds development and ensures a consistent look and feel across an organization's applications.

Here we overview some of the main characteristics of archetypes in order to understand what is occurring behind some of our application design choices.

### **6.3.2 Archetypes: a brief overview**

Archetypes are smart, graphic templates for application objects such as pages, forms, tables, rows and buttons. Archetypes for HTML applications consist of standard HTML code to control object formatting and special tags (using a Versata macro language) to specify how the Presentation Designer will expand the archetype to create the object.

Several sets of archetypes are delivered with the Versata Presentation Designer, and Versata customers modify these to create object templates that reflect their corporate look-and-feel.

The TradeX application is build with the Presentation Design Extensions (PDX) archetype set. PDX archetypes are a relatively new addition to the Versata System and allow many object properties, such as page captions and branching conditions, to be set through properties, instead of customizing the HTML or macro code. The overall appearance of the archetypes is controlled by a Cascading Style Sheet. Master colors, fonts, and other formatting are set in the style sheet and automatically flow through to the pages constructed by Versata.

As we build the TradeX application, you will notice that the design wizards present many choices. For instance, when creating an HTML page, the wizard will present archetype choices that specify whether the page will be a modal or non-modal page, whether it has a toolbar, and so on.

While details of the archetype system and macro language are beyond the scope of this redbook, they are explained in The Versata Developer Guide downloadable from Versata.

### 6.3.3 Designing the Home page

HTML pages display information from one or more RecordSources. A RecordSource is the application view of a Versata business object. At design time, the developer chooses an archetype for the RecordSource and can set RecordSource properties to refine its default behavior.

As a new application is created, the application wizard will prompt for the business object to be used as the primary RecordSource on the initial page.

Since the Home page displays information about the user's Account, we choose the Account object, as shown in Figure 6-9.

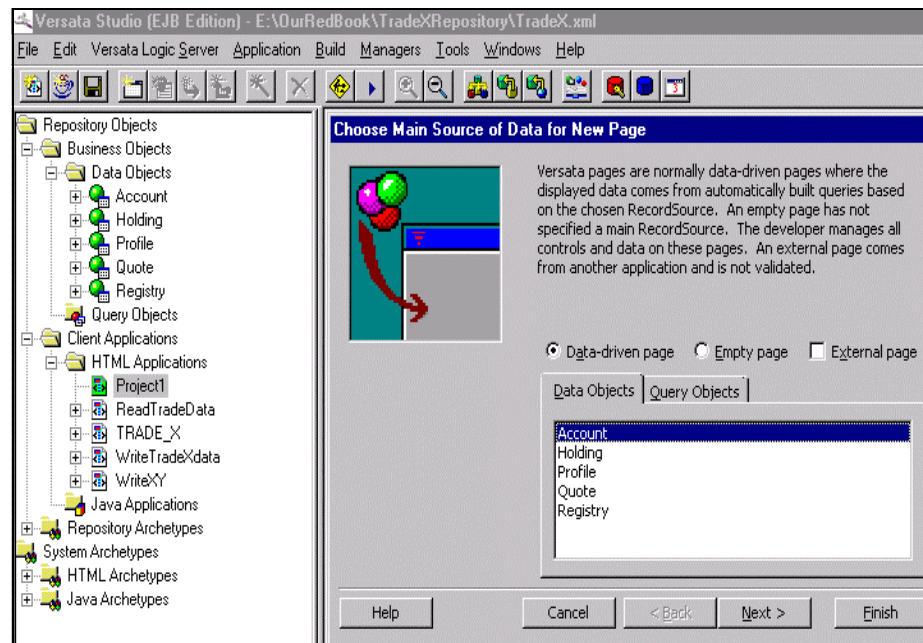


Figure 6-9 Account RecordSource on the home page

Figure 6-10 prompts for the archetype to be used for the Account RecordSource. The DisplayReadOnly choice will assure that the data is presented in a simple form (not a table) and cannot be updated.

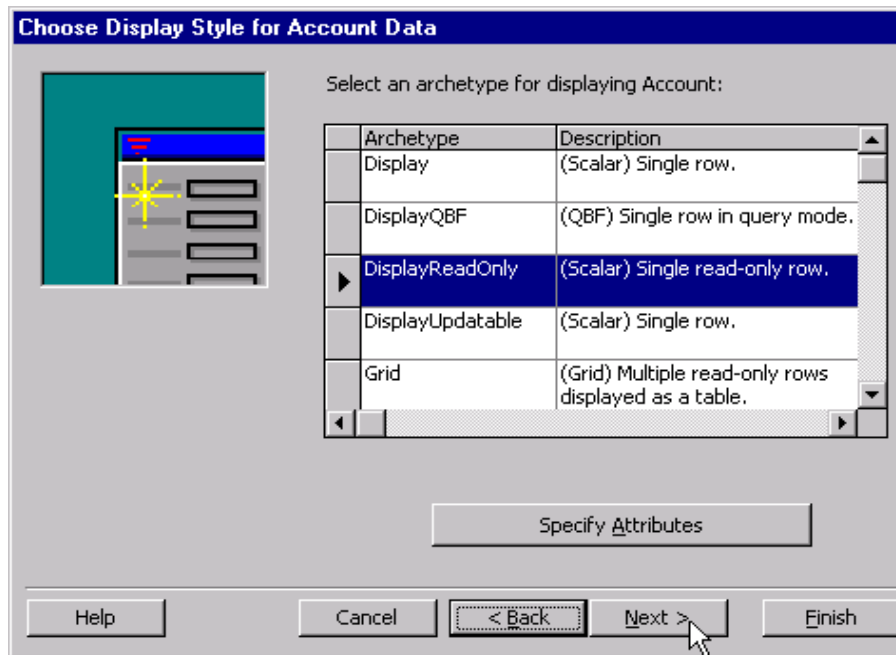


Figure 6-10 Choosing the archetype for the Account Record source

Figure 6-11 prompts for the archetype for the page. The Default choice provides us with a non-modal page that we can navigate using “transitions” (links to other pages). Versata builds these transitions from our design. In this panel the name of the object (used in constructing the page's Java page) and the Page Caption (used as the label in the page title bar) are also defined.

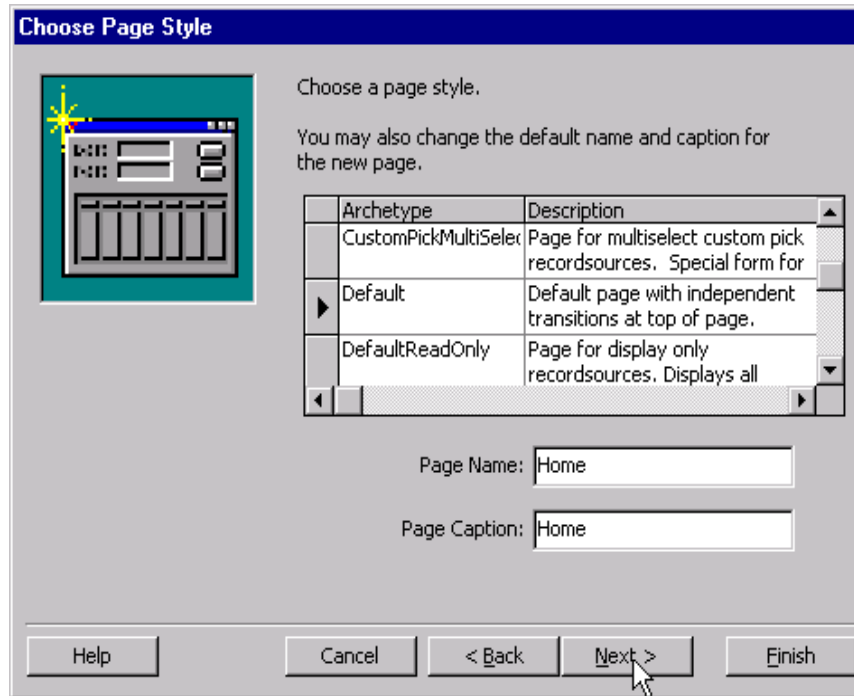


Figure 6-11 Choosing the archetype for the home page

When the user selects **Finish**, the page will be displayed on the right panel (the pallet) of the Application Designer. Notice that the Home page object is listed under the “Project 1” application in the left panel (Object View) of the Studio. (We have not named the application yet). The RecordSource “Account” is displayed under the page icon. As we continue to create the application, other pages and their objects will be shown. An illustration is given in Figure 6-12.

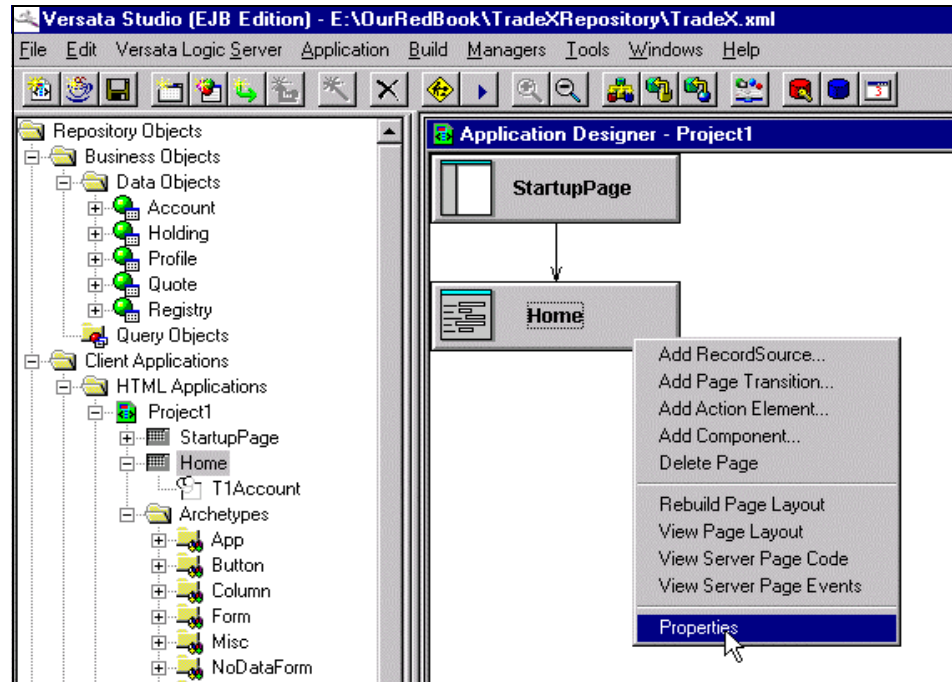


Figure 6-12 Refine the page properties

The final step in creating the Home page is setting the properties for the Page and the properties for the RecordSource, as shown in Figure 6-13.

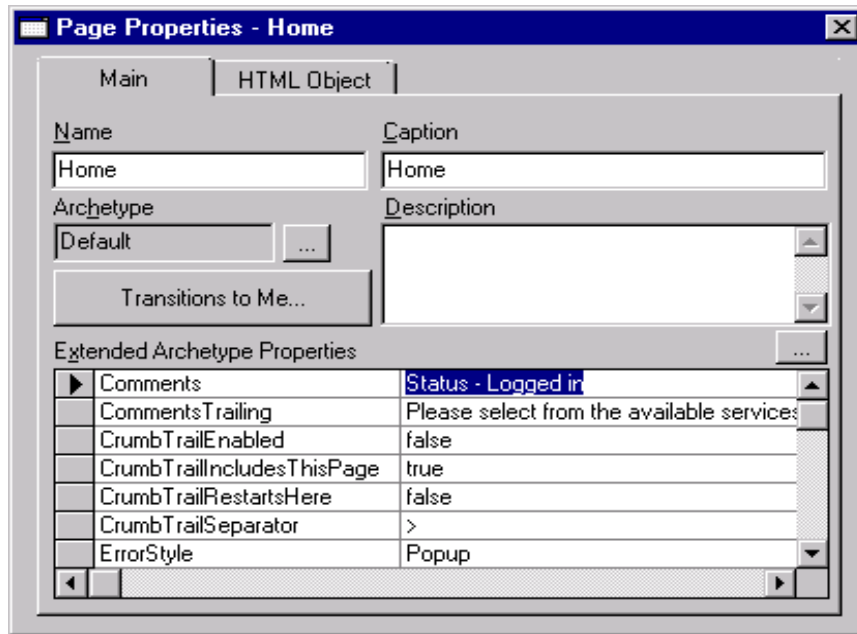


Figure 6-13 Properties for the home page

We set page properties for the Home page to include:

- ▶ **Comments:** This is text (and optional HTML formatting tags) to be included at the top of the page. Here we set the Welcome message.
- ▶ **Comments Trailing:** This is text to be included at the bottom of the page. Here we set the instructions regarding how to navigate the page.
- ▶ **CrumbTrail settings:** Here we indicate that previously viewed pages should not be displayed.
- ▶ **Borders:** Here we direct that files containing the static HTML for the top, side, and bottom banners should be used.

Next we set the properties for the Account RecordSource. Since a RecordSource has potentially more sophisticated behavior than the page, its property panel includes several tabs as shown in Figure 6-14.

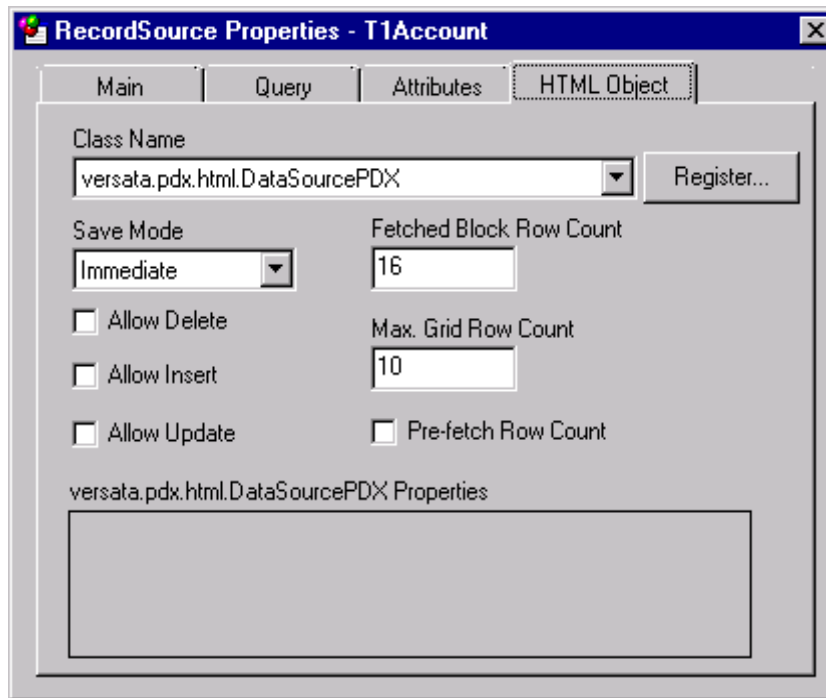


Figure 6-14 The Account Record source properties - HTML object tab

The HTML Object Tab shows:

- ▶ **Class Name:** This is the base Java class for the generated object. The component generated by the Versata System will be derived from this class and the component code will be automatically added to the Page's Java file.
- ▶ **Save mode:** For every RecordSource you can indicate whether changes to object should be saved immediately after the change is made by the user or only when the user clicks a **Save** button.
- ▶ **Fetched Block Row Count:** This sets the number of rows to be retrieved at a time. Subsequent rows are retrieved in this increment.
- ▶ **Grid Row Count:** This sets the number of rows to appear in a grid at runtime. Rows are scrolled in this increment.
- ▶ **Insert, Update and Delete boxes:** When checked, this indicates that these operations are permitted on the RecordSource.

Next we set the Attributes properties for the RecordSource. Here we have indicated that all of the attributes from the Account object should be displayed on the page, as shown in Figure 6-15.

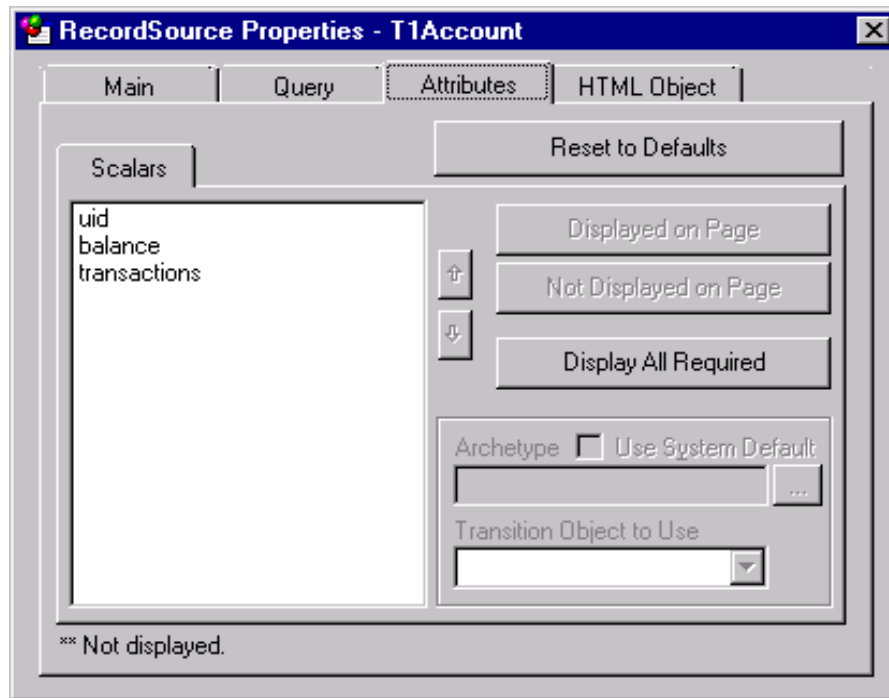


Figure 6-15 Choosing Account attributes for display

Finally, we set the Main properties for the Account RecordSource. (We demonstrate the Query tab in the next section). The Query to be used on this page has not been set yet. It will be set when we edit the transition from the login page to the home page, as shown in Figure 6-16.



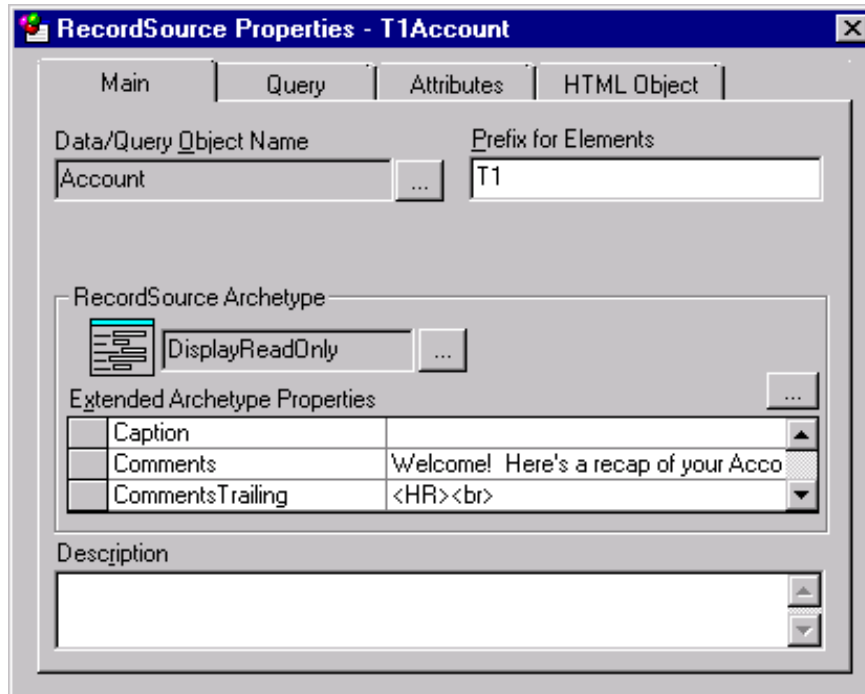


Figure 6-16 Main properties tab for the Account Record source

The main properties of the RecordSource include its archetype and the extended properties similar to those we saw for the Page. For the Account object they include:

- ▶ **Comments:** This is text displayed above the RecordSource.
- ▶ **Comments trailing:** This is text displayed after the RecordSource.
- ▶ **ShowRSAction flag:** This is set to false. By default, RecordSources are displayed with a default interface to scroll through records (first, last, next, previous), and insert, update, and delete records. Since the Account information cannot be changed on this page, we set this RecordSource option to false so that we don't display these buttons.

The remaining step, needed to complete the Home page, is to edit the link between the initial Startup page (the Login page in this case) and the Home page. This link is called a “transition” and it, along with the default Startup page, was created automatically by the New Application Wizard.

More than a simple HTML link, a transition coordinates navigation between the pages of an application or between different RecordSources on the same page. The transition contains the information required to display data on the target form or page. This information is usually a query that selects records in the target RecordSource. A user-selected button on the source form or hyperlink on the source page executes the query.

To edit the transition to the Home page, we click the transition (represented by the curved arrow to Home) in the Object View of the Studio and select its **Properties** as shown in Figure 6-17.

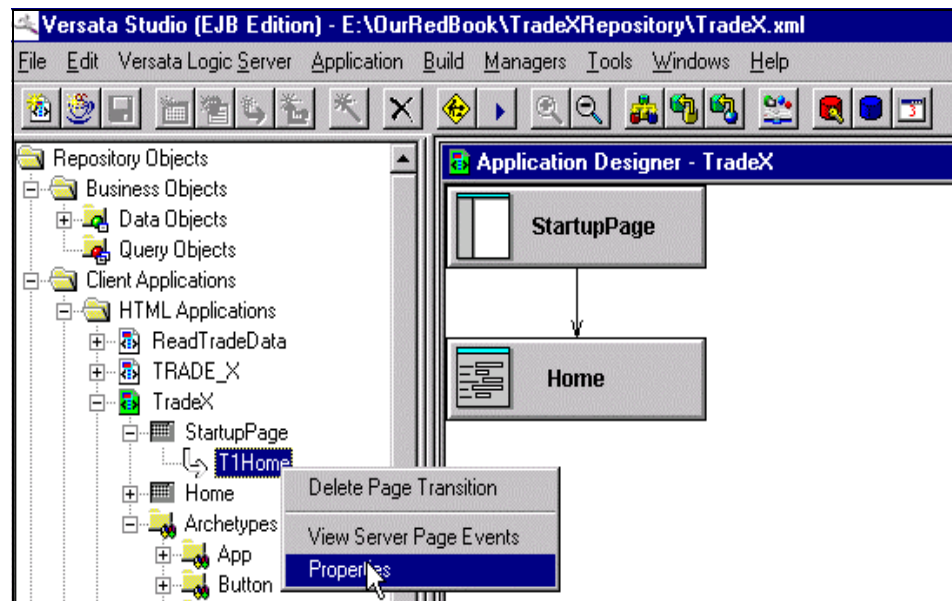


Figure 6-17 Editing transition properties

The property we need to edit on the Startup-Home transition is the Query property. This determines which Account record will be displayed to the user on the Home page. Versata makes available all of the attributes and methods of the Account object, along with a number of system methods, through the query “expression builder”. This makes it easy to instruct the Home page to display the Account object for the logged-on user, as shown in Figure 6-18.

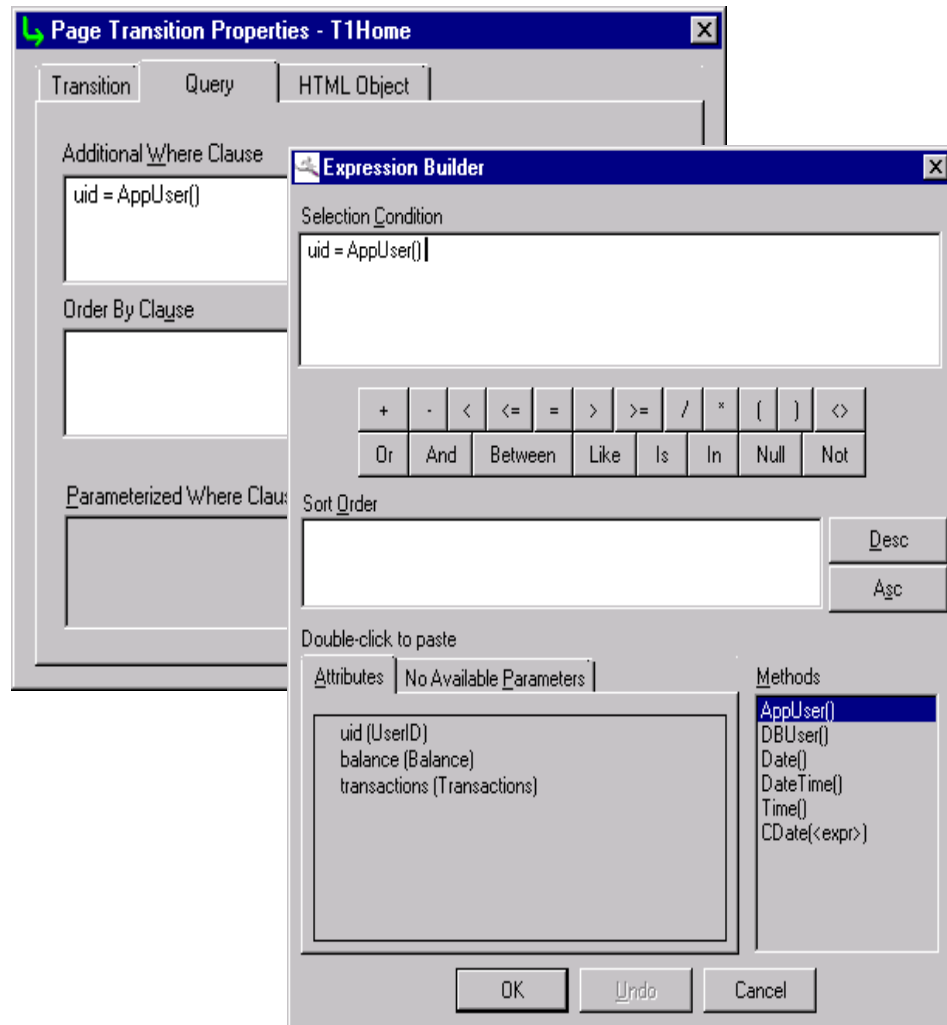


Figure 6-18 Building the query for the home page

### 6.3.4 Designing the QuoteBuy page

The next page we build is the QUOTE\_BUY page. This allows us to view stocks and prices, as well as purchase holdings.

The QuoteBuy page uses information from the Account object (the userID at the top of the form) and the Holding object (a blank grid ready to insert a Holding.) The Account object will be used as our primary RecordSource, the Holding object will be a dependent record source. A Versata page will have only one primary RecordSource, but can have any number of dependent sources. Versata can automatically coordinate RecordSources, so that once we find the Account for the user, only that Account's holdings will be displayed.

The QuoteBuy page is started by dragging the Account business object onto the pallet and using the Page Wizard to choose Page and Record source options such as its archetype.

For the Account object on the QuoteBuy page, we select a DisplayReadOnly archetype and select only the UID attribute for display, as shown in Figure 6-19.

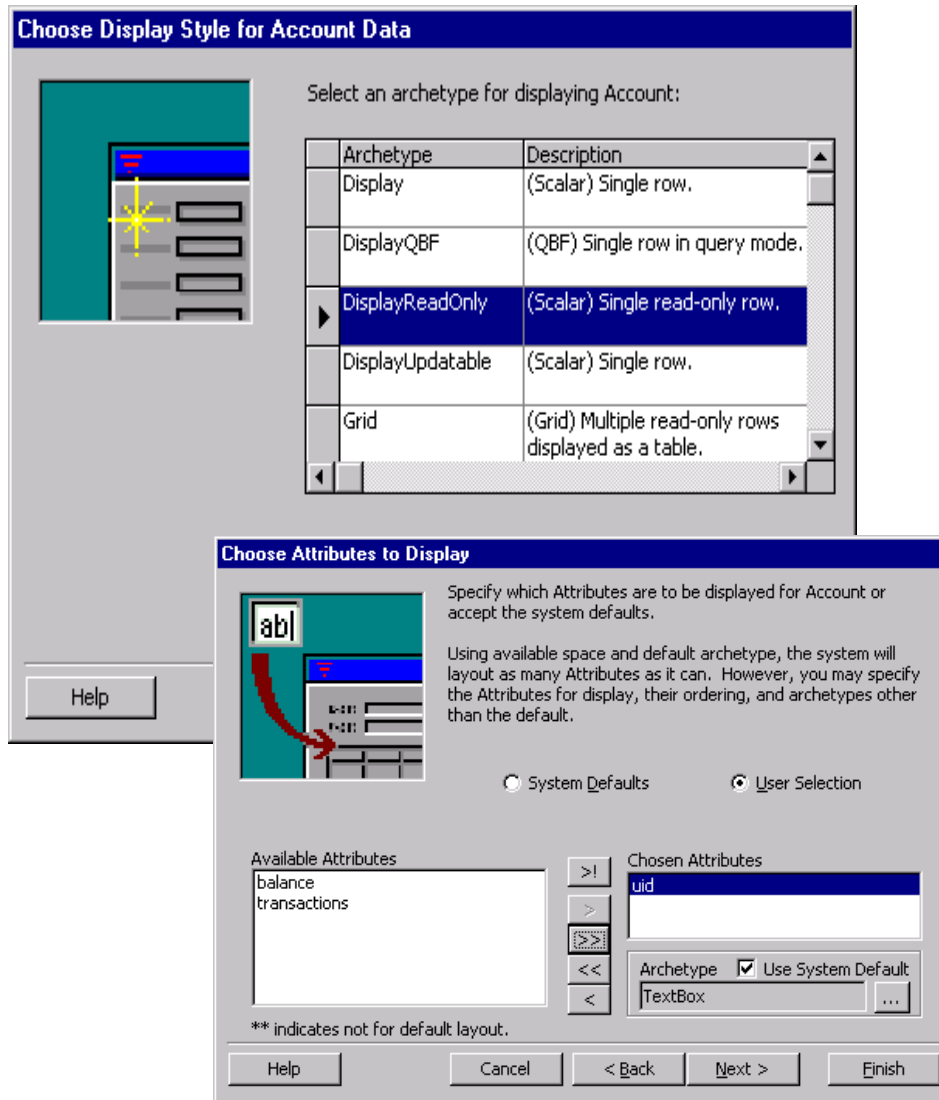


Figure 6-19 Specifying the Account Record source on the QuoteBuy page

To add other RecordSources to the Page, we can drag the required business object to the pallet. When dropped onto the page, a Wizard appears.

The first panel of the wizard asks whether the Holding RecordSource will be dependent to the Account RecordSource, for example, are they coordinated in a Master/Detail relationship on the page. Since a relationship rule was defined between the two business objects in the business logic tier, Versata can also use this relationship to coordinate client interactions, as shown in Figure 6-20.

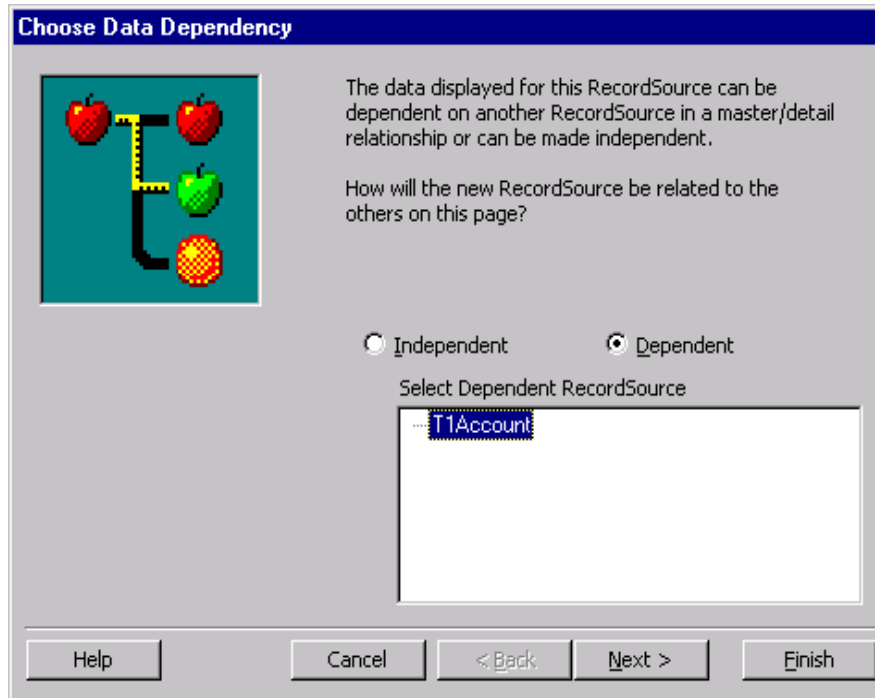


Figure 6-20 Choosing Account/Holding client coordination

Figure 6-21 asks for the archetype to be used for Holdings. Here we choose an UpdateableGrid, which is an HTML table with a variety of built in functions such as updateable cells, selectable rows and sortable columns.

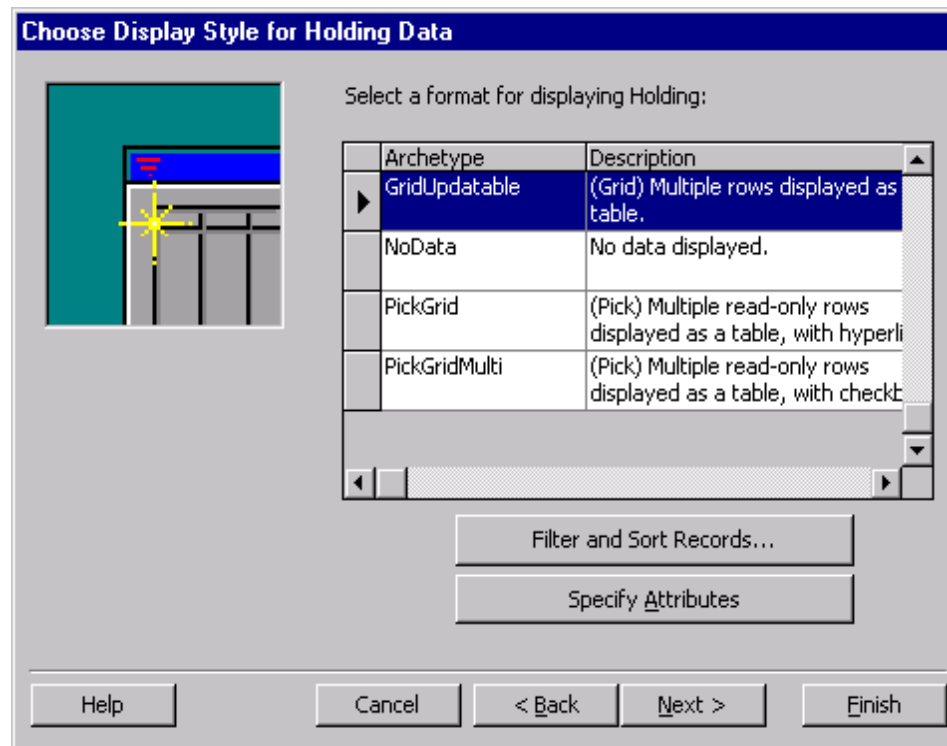


Figure 6-21 Choosing the archetype for Holdings

Figure 6-22 asks for the attributes from Holding to be displayed. We will display only the stock Symbol and Quantity, since these are the fields that can be entered by the user.

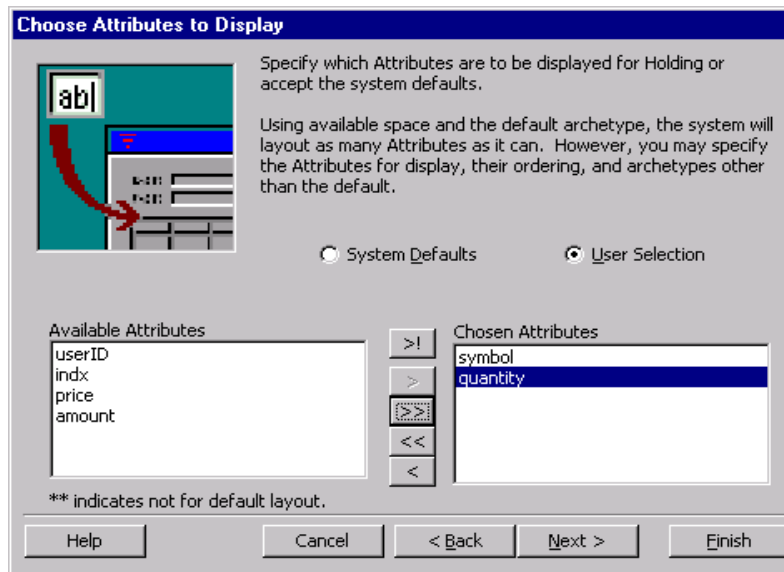


Figure 6-22 Choosing displayed attributes for Holding



The final Wizard panel, Figure 6-23, constructs the “pop-up” form that displays stock symbols and quotes (from the Quote object). In the Versata system, this is called a “Pick”. A Pick list pop-up can be specified for any attribute that participates in a parent-child between the business objects.

In the TradeX model, the Quote object is the parent of Holdings (one-to-many relationship) and they are joined on the Symbol primary and foreign keys. Therefore, the Versata-generated Holding.Symbol field on the QuoteBuy page can automatically present all of the Quotes as choices in a pop-up window. The Wizard prompts for which pop-ups to construct.

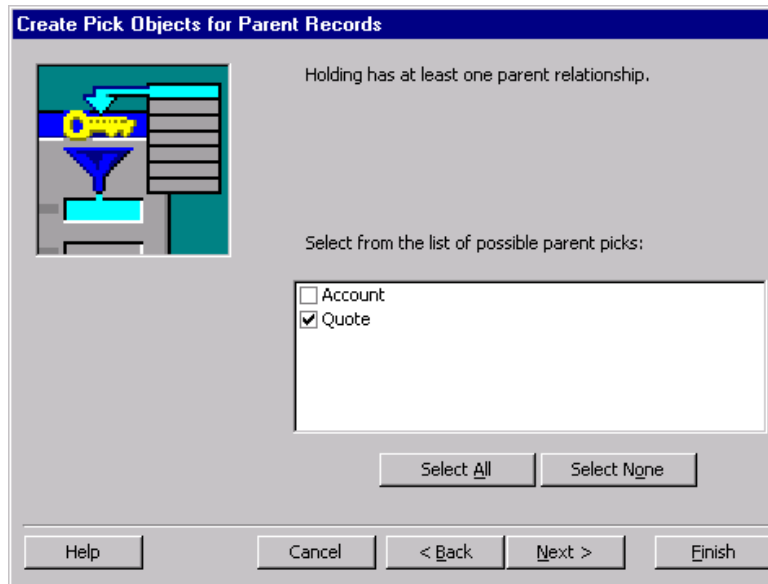


Figure 6-23 Specifying the pop-up for viewing Quotes

This completes the QuoteBuy page Wizard. When the Wizard completes, the Page properties can be edited using the same process we used to edit Main properties for the Home page. This sets the captions, and initial query mode for the page.

The Object properties for the Holding RecordSource are set so that only one row will be shown in the UpdateableGrid, and the Save model will be set to “Immediate”. This means the new Holding row will be sent as soon as the user enters it, as shown in Figure 6-24.

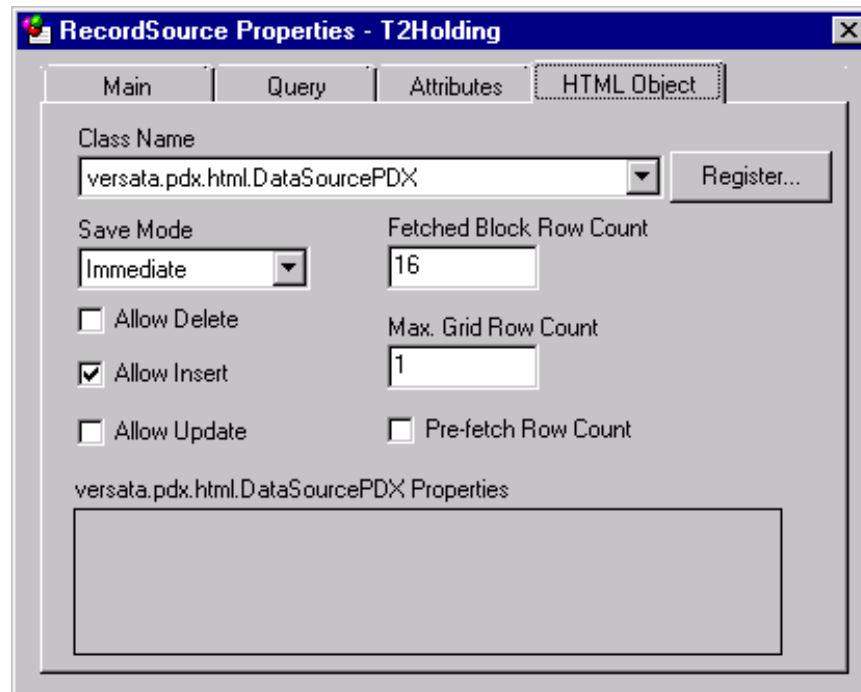


Figure 6-24 Allowing one Holding row to be inserted

The final step for the QuoteBuy page is to create a transition from the Home Page to the QuoteBuy Page. Like the transition from the Startup Page to the Home Page, the transition between Home and QuoteBuy uses the query “uid = AppUser()” — that is, display the Account object where the userID is the logged in user.

### 6.3.5 Creating the Portfolio page

Like the QuoteBuy page, the Portfolio page uses the Account business object as the primary RecordSource, and it uses the Holding business object as the dependent RecordSource. Also, like the QuoteBuy page, the transition to the Portfolio queries the Account with the logged in UserID.

The primary difference in the Portfolio page is that Holdings are displayed, ten rows at a time, on a ReadOnly grid. The only action permitted on the grid is to select a holding and delete it, as shown in Figure 6-25.

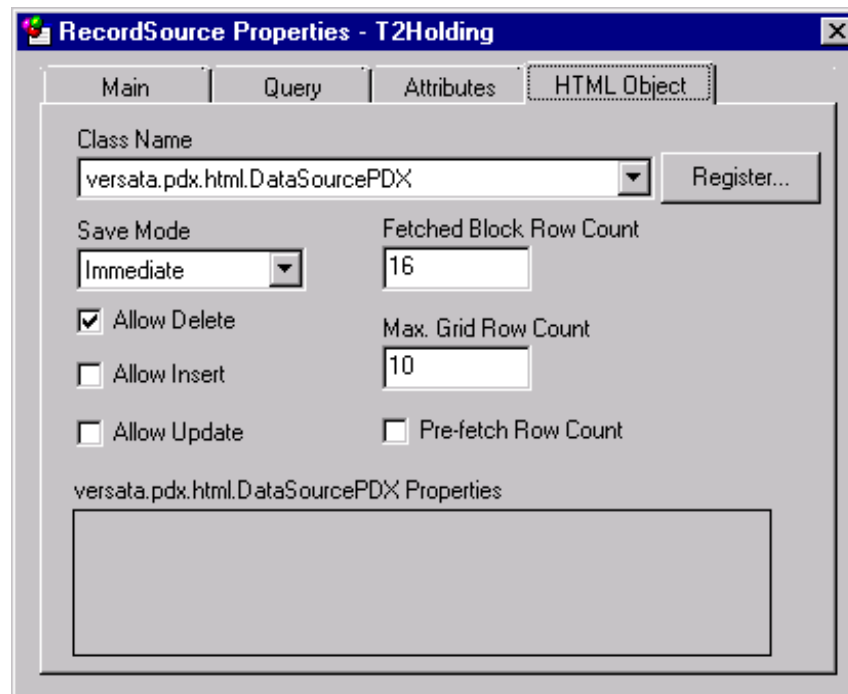


Figure 6-25 Properties for Holding Record Source on the Portfolio page

### 6.3.6 Creating the Profile page

The final page in the application allows the user to update his Profile information. This simple page uses the Profile business object with UpdateableDisplay archetype. With the UpdateableDisplay, the controls on the page to “Save” or “Undo” changes are created automatically.

Like the other pages, the Profile page is completed by editing its properties, including the Captions and Comments.

## 6.4 Completing the application design

With all of the application pages complete, the Presentation Designer pallet looks like this. The application is complete except for adding the transitions between the pages, as shown in Figure 6-26.

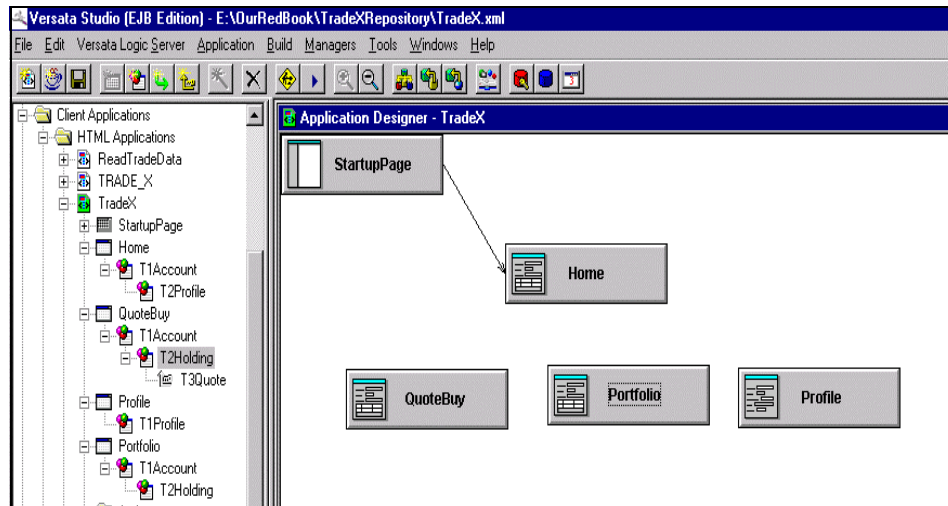


Figure 6-26 Application pages before transitions

Transitions are added from each page to all other pages, using the “Add Transition” Wizard. The main property of each transition is the Query used to enter the page. Like the Home page, each page executed the Query “uid = AppUser()” for the primary RecordSource.

In Chapter 5, “Rule-based development” on page 57 and Chapter 6, “Designing an HTML client application” on page 73, we have completed the design of a complete, end-to-end TradeX application.

Chapter 7, “Deploying the TradeX application” on page 103, demonstrates deployment of components into the WebSphere application server.



## Deploying the TradeX application

Deployment is the process of packaging the components of an application so that the application can be run by users. The Versata Studio provides a graphical Deployment Manager so that all three tiers of the application — persistence tier, business logic tier, and user-interface tier — can be deployed with a few simple steps. Alternatively, Versata provides a command-line interface that can be used to deploy components independently of the Studio. These are helpful if Versata components are part of a larger development effort that uses them to “make” files to build applications.

Behind the scenes, either process sets up the necessary connections, creates the necessary deployment descriptors, and interacts with WebSphere to install components into the WebSphere Application Server.

In this chapter we demonstrate the Versata automated deployment process.

## 7.1 Business object deployment

Business object deployment is the process of adding the Java classes built and compiled for business objects to the repository (.jar) file, and then adding that jar file to the classpath of the Versata Logic Server running in WebSphere. The process also creates and installs the WebSphere deployment descriptors for the business objects created as EJBs. The WebSphere Administrative Console after deployment of the TradeX business components is shown in Figure 7-1.

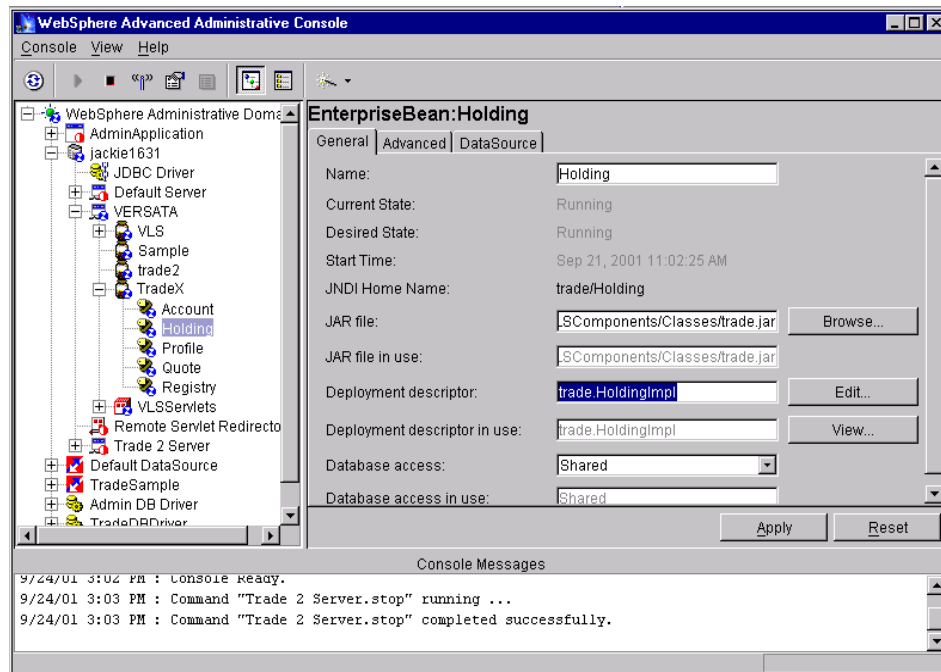


Figure 7-1 WebSphere console after TradeX business logic deployment

The deployment Wizard presents a few simple screens to verify the directory where the business objects \*.jar file will be copied and to ask whether the WebSphere server should be automatically stopped and restarted. (Restarting the WebSphere server, in Version 3.5, allows the new (.jar) file to be recognized.)

These simple steps complete the business logic deployment process as shown in Figure 7-2 and Figure 7-3.

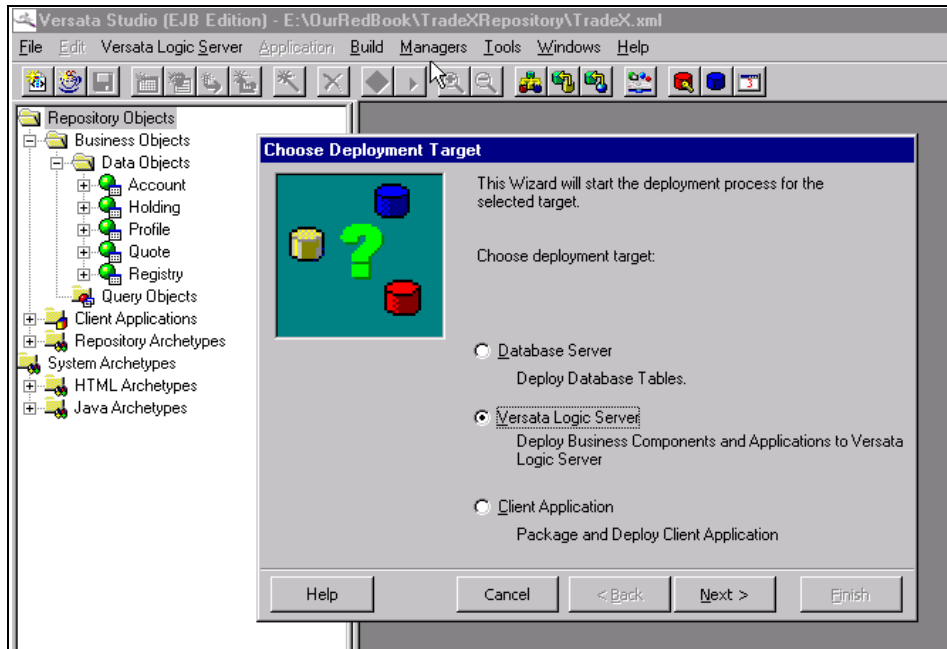


Figure 7-2 Beginning to deploy the business logic tier

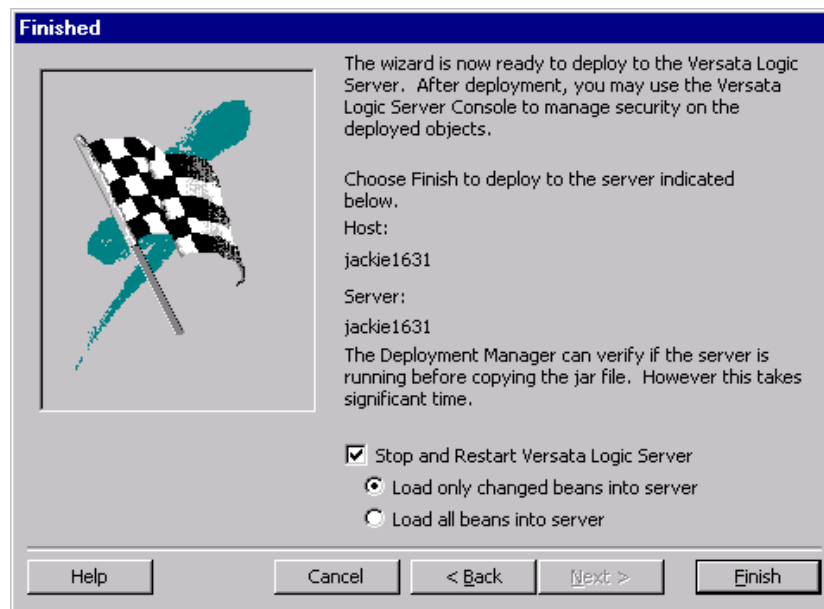


Figure 7-3 Completing the business object deployment process

## 7.2 Database deployment

As we explained in Chapter 4, “Architecture of the Versata Logic Server within WebSphere” on page 39, Versata Data Objects are persistent components that represent data in a data store. Data Objects are implemented as J2EE entity EJBs and, where possible, the Versata Transaction Logic Server persists their data using the Java Transaction Services API (JTA) available from WebSphere. In the case of the TradeX application, the business objects are persisted to a DB2 database.

Database deployment is the process of automatically creating a database schema to mirror the structure of Versata Data Objects so that they can be persisted. This is useful if the data model was imported from an external tool such as Rational Rose or an existing Versata XML repository. It is also useful if the data model changes after re-engineering from an existing database schema. In this case, the Deployment Manager can re-synchronize the entity-EJB model with the original database.

For the TradeX application built in Chapter 5, the data model was not changed from the original DB2 database schema. (An amount attribute was added to the Holding object, but this was a non-persistent attribute.) This means that we can skip the database deployment step when building this initial TradeX application. However, in the next chapters we will substantially enhance the DB2 model. So the following steps will be used to alter the DB2 database shipped with the TradeX application.

### 7.2.1 Setting up an ODBC Data Source Name (DSN)

To re-engineer, create, or resynchronize a schema, Database Deployment Wizard communicates with a DBMS using ODBC. (This is different from the process used to persist data, which uses JDBC through JTA.)

In order to re-engineer or deploy the data model, we first set up an ODBC data source name (DSN) for the TradeX DB2 database. On Windows NT, this is done through the NT Control Panel's ODBC Data Source Administrator.



For the TradeX database, we first add the TradeX System DSN, using the DB2 ODBC Driver and specify the additional DSN settings as shown in Figure 7-4.



Figure 7-4 Add ODBC system data source name

Within the Versata Studio, select the Manager from the top toolbar and specify a DB2 database as shown in Figure 7-5.

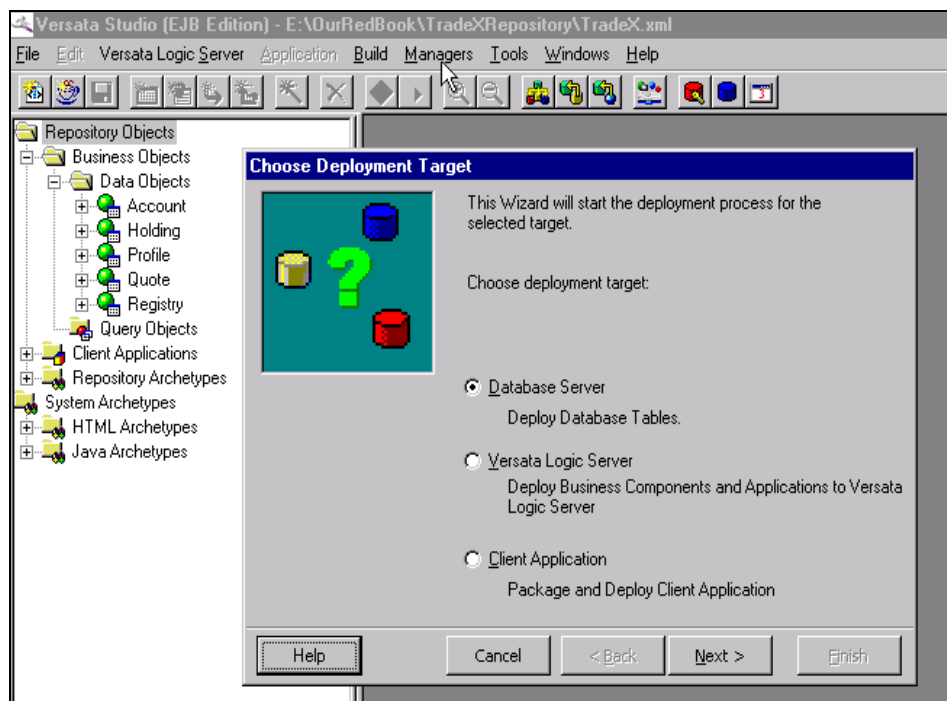


Figure 7-5 Begin deployment to database

Next, select the data objects to create (or re-create) in the DBMS and select whether the system will interact with the Database directly or create a SQL script, which can be used at a later time as shown in Figure 7-6. Finally, if prompted, select the System DSN that describes the TradeX database.

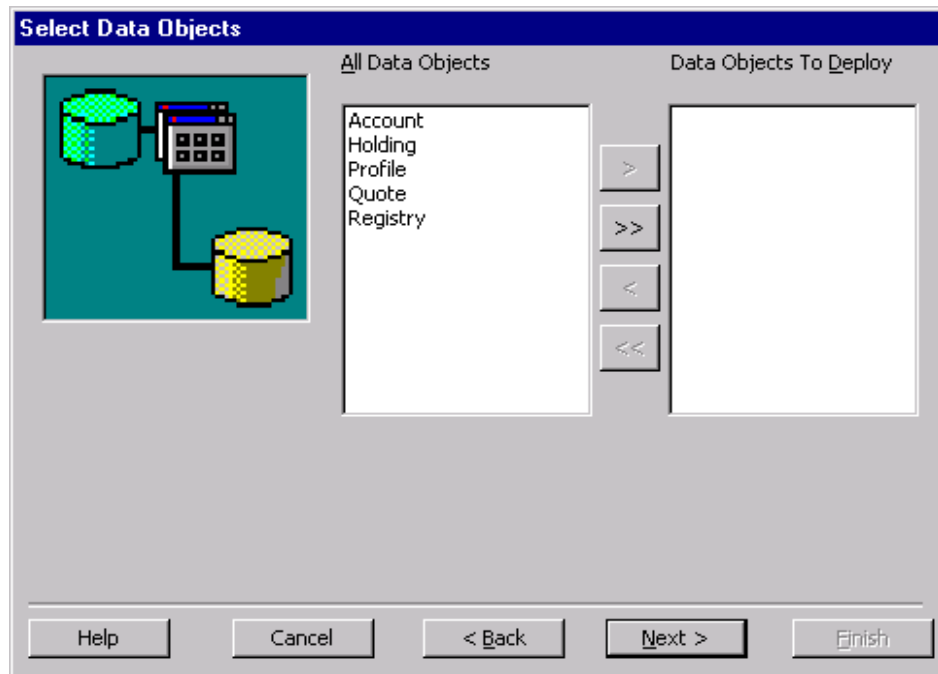


Figure 7-6 Choose tables to be created in database

## 7.3 Reviewing or setting the data server

In Chapter 5, “Rule-based development” on page 57, we touched on how to re-engineer the Versata data model from the TradeX DB2 database. Above, we showed how to re-synchronize the Versata model to the database in case of changes. In addition to these topics, there is one other step to complete the picture of how the Versata Logic Server communicates with data sources. This step sets a DataServer for our business objects.

In Versata, a Data Server is a named connection (pointer) that binds one or more Data Objects to their persistent data store. You will recall from Chapter 4, “Architecture of the Versata Logic Server within WebSphere” on page 39, that the Versata Connector Architecture supports both relational databases (which are typically supported by WebSphere) and non-relational data sources (which may not be.) The Data Server abstraction allows a Versata Data Object to be mapped to its source, regardless of the type of the source.

Data Servers are mapped in the Versata Logic Server Console. The console is a Java program that can be accessed through the Studio toolbar or through the operating system. On NT this is done through the Versata program group -> **Start -> Versata System -> Versata Logic Server Console.**

After logging on to the Console, with the default id “sa” (no password), the DataServers can be viewed under the “Administration” heading as shown in Figure 7-7.

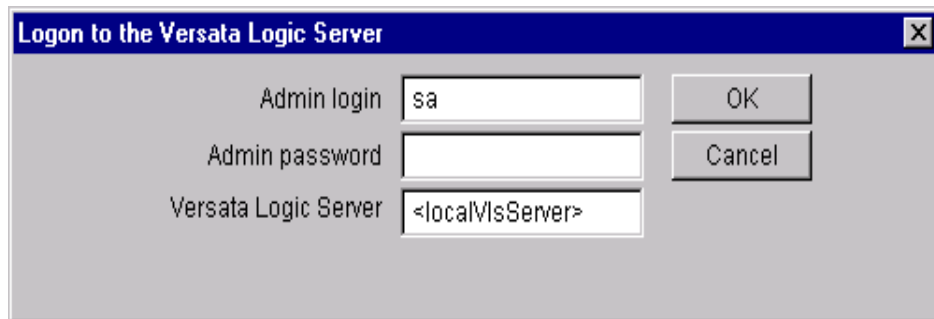


Figure 7-7 Login to the Versata Logic Server console

If business objects have been re-engineered from an existing database, or if the Database Deployment Manager has been used to re-synchronize the database schema, a DataServer may have been automatically created. Scroll through the list of servers to find the database type and name you will be using for the TradeX application.

If the DataServer exists, the default properties such as connection pooling, timeouts, and other tunable values can be retained. If the DataServer has not been created, a new server can be specified from the DataServers list in the left panel of the console as shown in Figure 7-8.

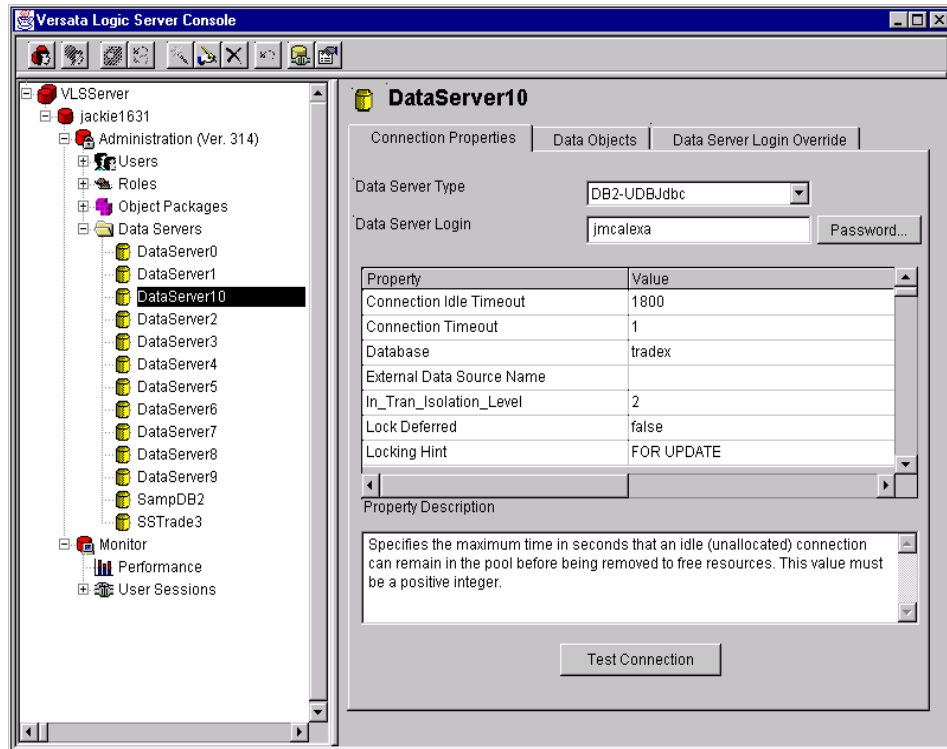


Figure 7-8 Creating or examining data server properties

Deployed data objects also appear in the Console. They are organized by Versata Repository name. Here, the TradeX data objects are assigned to the Data Server defined for the TradeX DB2 database as shown in Figure 7-9. (By default, this was sequentially numbered as DataServer10).

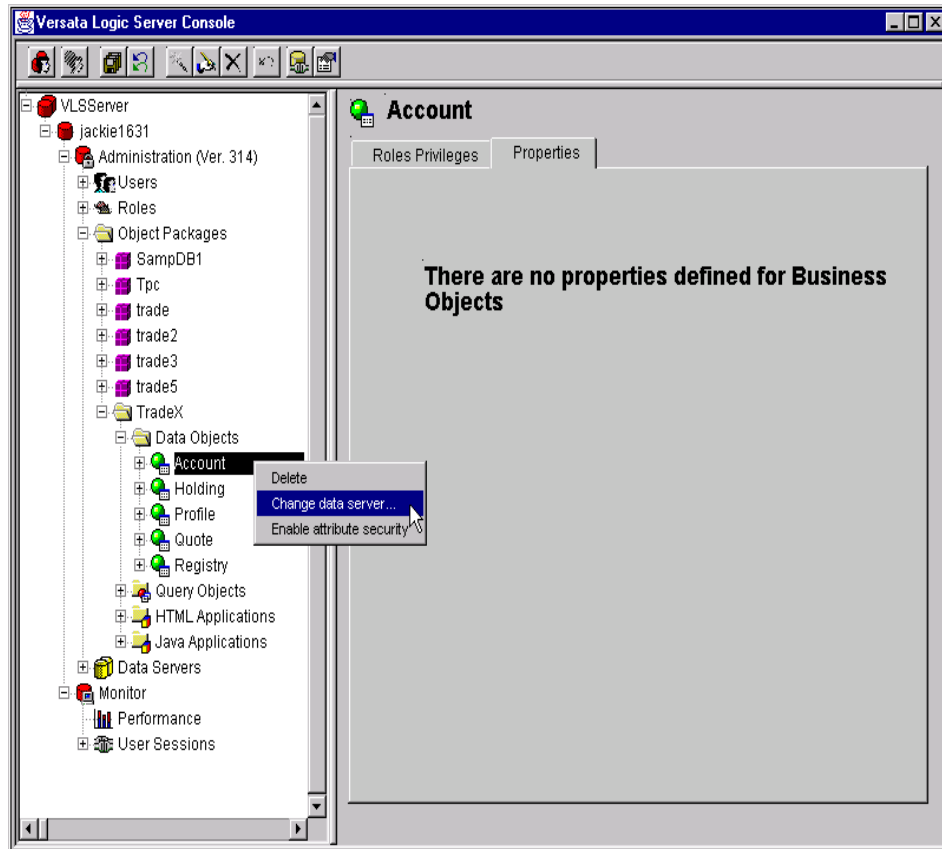


Figure 7-9 Assigning data objects to the TradeX data server

## 7.4 Granting access to TradeX users

The Versata Logic Server controls business object security by assigning users to roles, and granting privileges (read, create, update, and delete) on data or query objects to those roles.

Users can be authenticated and mapped to roles in a variety of ways. For instance, any WebSphere-supported user authentication method can be used to validate users and passwords. Users can be mapped to roles by through WebSphere mechanisms as well. For simplicity, in the TradeX application, we will default to Versata's standard security scheme, which is administered through the Logic Server Console. To test security, we define several users from the information contained in the “Registry” table of the IBM Trade Database as shown in Figure 7-10.

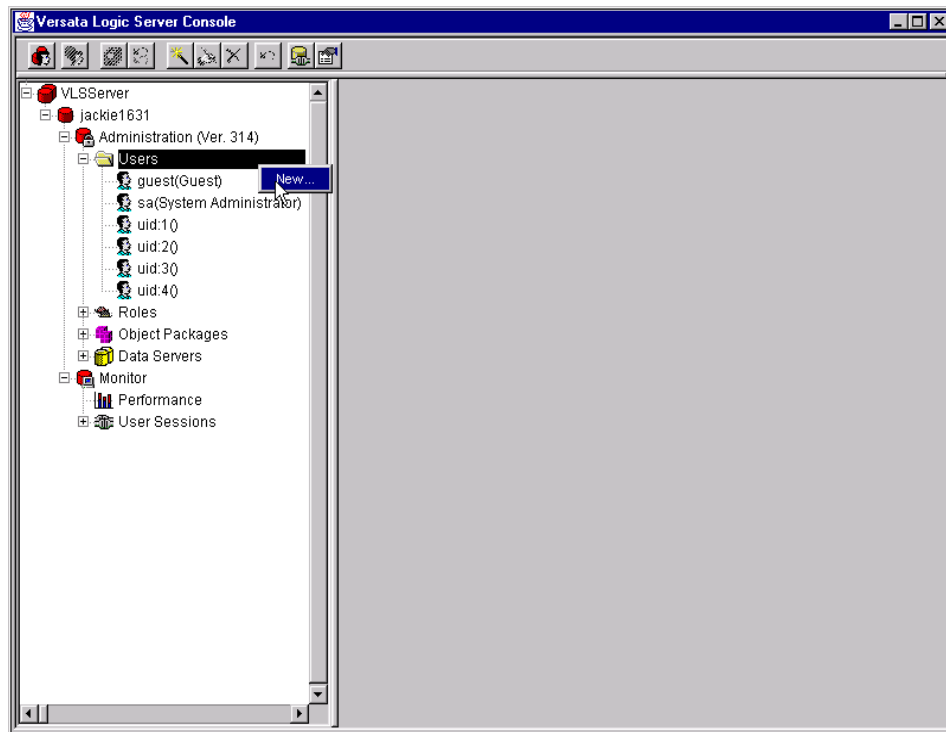


Figure 7-10 Defining TradeX users to the logic server

Then the users are assigned to the general role “Public”. Because the security model of the Trade application is enforced at the application level (by comparing the userID to the logged on user), we grant all object permissions to the Public role as shown in Figure 7-11.

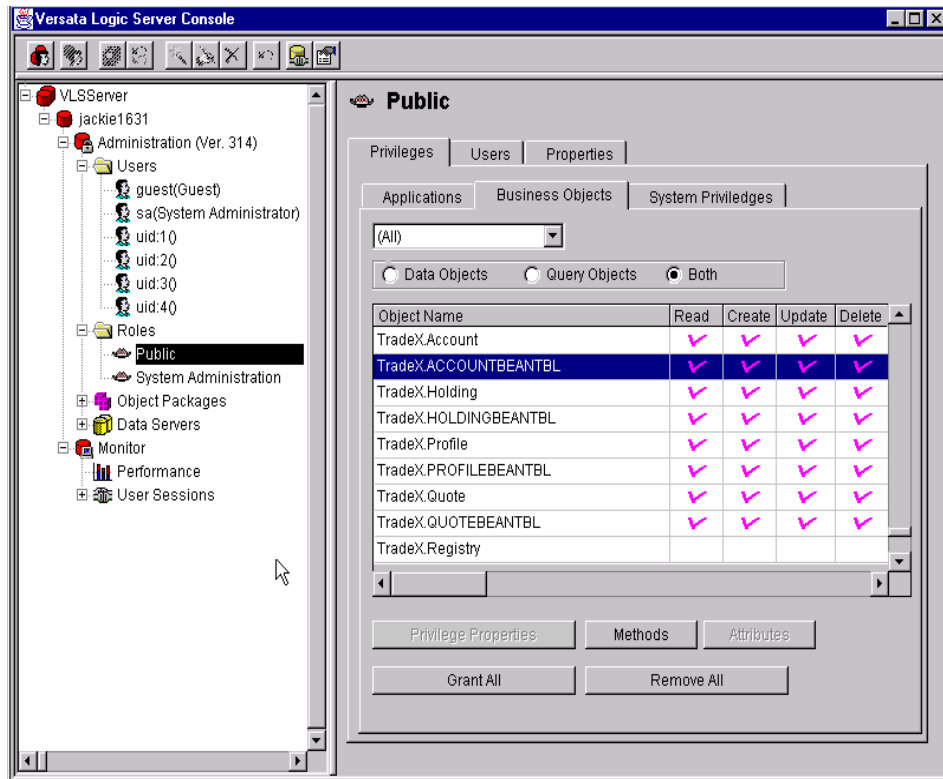


Figure 7-11 Granting permissions to the TradeX role

## 7.5 Client application deployment

Deploying a Versata-created client application installs the application servlet in the WebSphere servlet engine (Web container), copies the application HTML pages to the document directory of the desired HTTP server, and creates and installs the application (.jar) file in the ClassPath of the Presentation Logic Server EJB (PLSContext.) Deployment of these client components is automated through the HTML Application Deployment Wizard as shown in Figure 7-12.

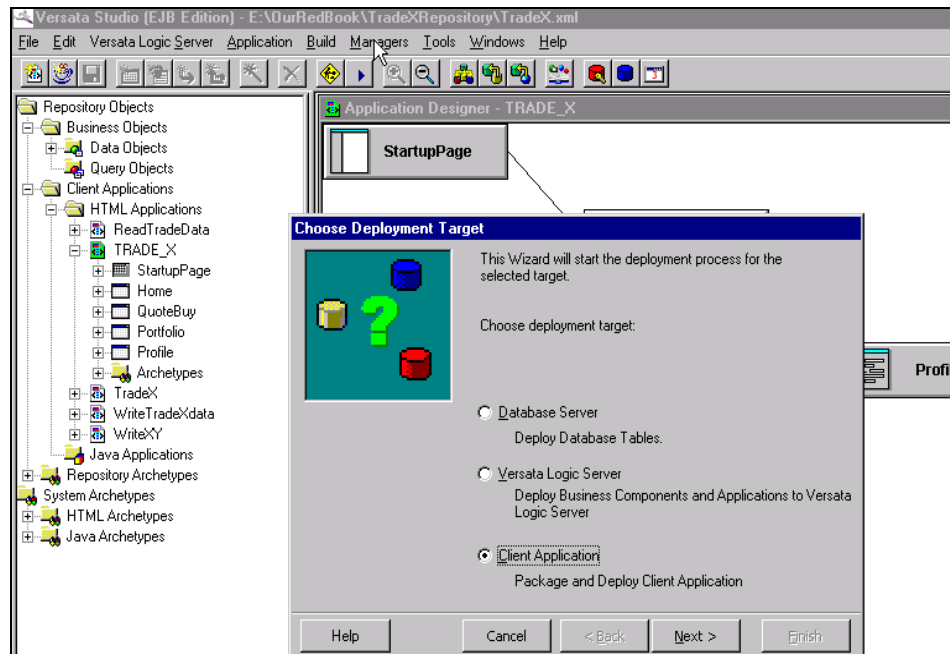


Figure 7-12 Beginning client application deployment



The Deployment Wizard provides a series of dialogs that verify installation directories as shown in Figure 7-13. (The defaults were set during the Logic Server installation.)

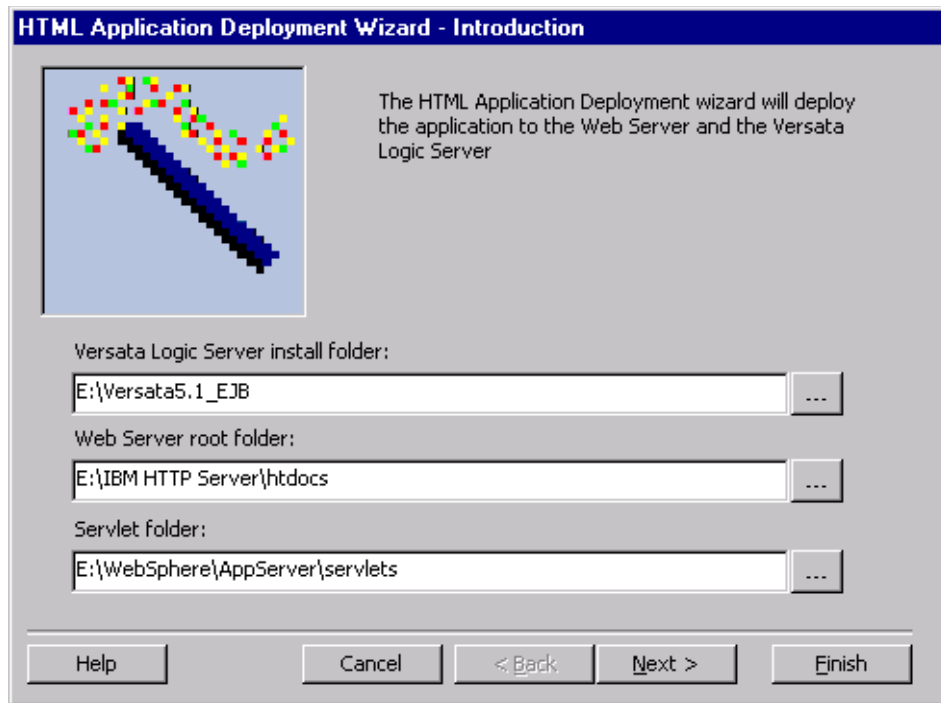


Figure 7-13 Component directories in the Application Deployment Wizard

Finally, the Presentation Server within WebSphere is restarted to recognize the new application as shown in Figure 7-14.

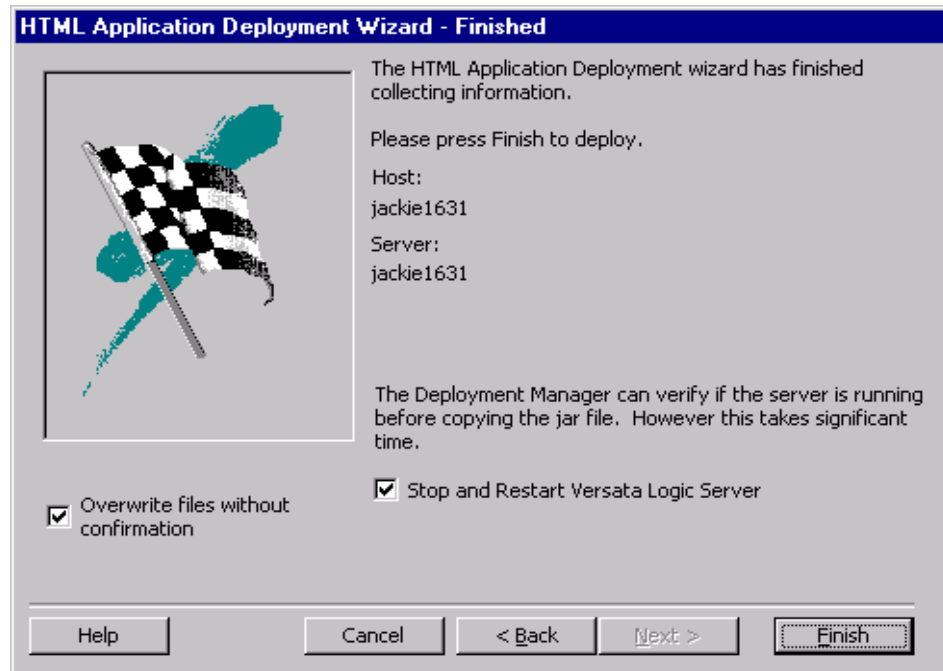


Figure 7-14 Confirming application deployment

## 7.6 Executing deployed applications

After deployment, applications can be started from within the Versata Studio, or they can be accessed through the URL or the servlet that controls the application.

The servlet will have the same name as the application, and, by default, be put in a webapp directory, in a subdirectory reserved for that repository. For instance, in the TradeX application, in the TradeX repository, the servlet URL is `http://localhost/webapp/TradeX/TradeX` as shown in Figure 7-15.

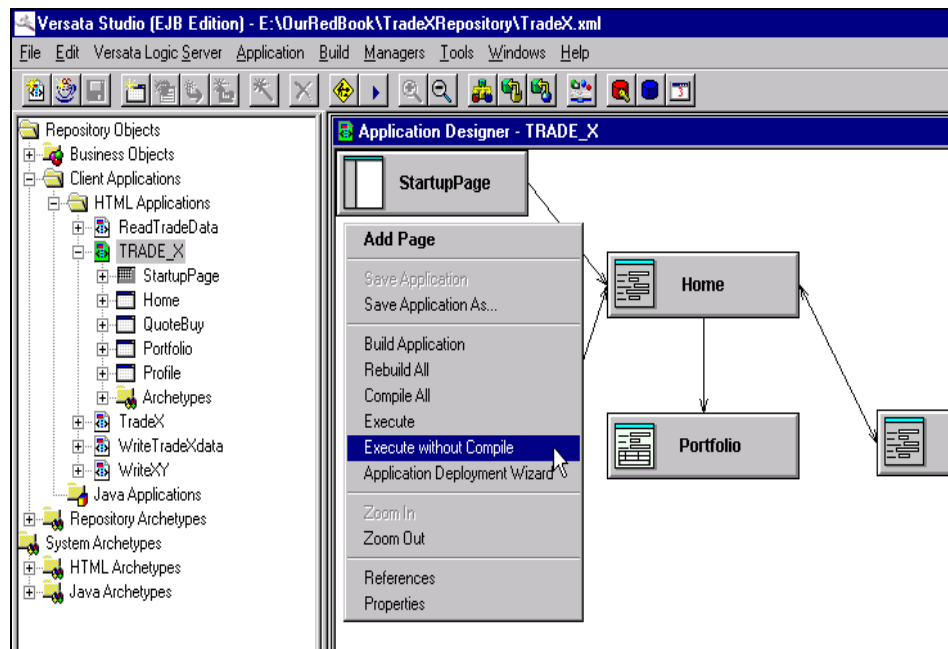


Figure 7-15 Executing an application from the Versata Studio

## 7.7 Generating business and application logic reports

The Versata Studio can generate reports from its repositories to summarize the definition of data models, business rules, and application components. In this way, the Versata Logic Server serves as a dynamic “memory” of an organization's business processes and the high-level logic that implements them.

Shared with business analysts and new members of the development team, the reports become instant, and up-to-date documentation, useful for auditing or regulatory compliance. When used with any third-party source code control system, the repository can be “differenced” to track system changes over time as shown in Figure 7-16.

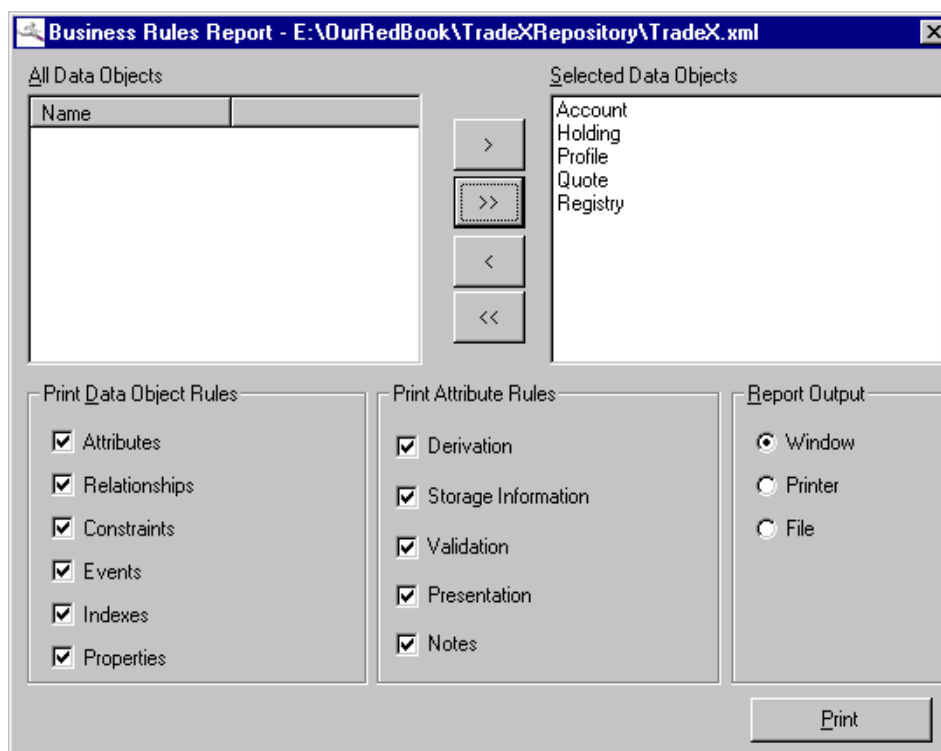


Figure 7-16 Specifying a business logic report

In the following chapter we will demonstrate how to enhance the business logic of TradeX to provide a more real-world application scenario.



## Enhancing TradeX business logic

As we saw in Chapter 5, “Rule-based development” on page 57, the initial business logic of the TradeX application is very basic. Buying stock results in a new Holding, which must be sold in its entirety. A user's account balance is not checked before a purchase and can be overdrawn. In addition, there is no provision for commissions or for “personalization” to distinguish between an active, high balance customer and a potentially less profitable account.

In this chapter we enhance the TradeX application to meet a series of new business requirements. Changes will be made easily and incrementally, through new business rules that adjust the functionality of existing business objects.

## 8.1 New requirements

The new requirements are to accomplish the following:

1. **Allow blocks of shares to be sold against a Holding.** The user's initial purchase of the shares will still cause a new Holding to be added, but each transaction against that Holding, including the “buy” transaction will be recorded. Rules will be added to ensure that a user doesn't sell more shares than he has in his account. (for example, short selling will not be allowed.)
2. **Establish a system of Account types.** Account types will be graded by the cash balance on hand, the size of the portfolio, and the number of trades.
3. **Charge commission on each transaction.** Apply a variable commission rate, depending on the type of account, to calculate the commission amount. As a promotion, newly established accounts (those with less than five trades) will not be charged commission. “Wholesale accounts” (those with large total assets) will be charged a lower rate than “Retail accounts”.
4. **Limit margin selling.** For the purposes of this example, we will assume that the Securities and Exchange Commission (SEC) frequently changes the rule controlling margin selling where a user “borrows” money in his account to cover part of the purchase price. To accommodate the changes, the margin selling rules must be “parameter driven”, that is their behavior must change based on outside data that is, in this case, read from a table. In this way, managers can maintain a table of current SEC regulations and the Versata rule need not be re-written or re-deployed to enforce the new regulation. (Parameters to rules can come from a number of sources, including method calls to external systems, data passed from the client, and so on. In this case, we illustrate a simple, table driven example.)

## 8.2 The TradeXv2 repository

A real-life Versata environment typically includes a development and a production server, so that changes to business logic can be made and tested without affecting a running application. As development proceeds, the Versata repository, along with the source code for customized components are versioned in a source code control system to track the changes.

In our single-user TradeX environment, however, we are working in with a single server. To make our changes in this Chapter, while preserving the functionality of the original TradeX application, we will simply “clone” the TradeX business objects and rule definitions from Chapter 5, “Rule-based development” on page 57. This can be done by importing the TradeX XML files into our new TradeXv2 repository.

As with the basic TradeX logic, to enhance the application we state the business requirement, and demonstrate the steps needed to implement the requirement in the Versata Studio.

## 8.2.1 Requirement 1: Sell partial holdings

The first requirement, to be able to sell partial Holdings, requires a significant change to our object model. A new Transaction business object must be added to capture the original buy operation for a holding, as well as the multiple sell operations against it.

The original Holding object retains attributes such as the index (ID), stock symbol, price and quantity of the original purchase of the Holding. In addition, we add new attributes including those to track the number of transactions against a Holding, the shares remaining in the Holding, and the derived current value of the Holding. These values can be reported to users viewing their portfolio and used in the derivation of other object attributes, such as the Account.AccountType.

A new Transaction object will store the date, quantity, price, and transaction type for each operation against the Holding. Each Transaction will be related to a Holding by its HoldingIdx, which is a foreign key to the Holding indx attribute. A Transaction will be uniquely identified by a combination of its HoldingIndex and a SequenceNumber, which is incremented with each transaction against the Holding.

The new Data Object, Transaction, is created in the Business Logic Designer by specifying its name and its attributes as shown in Figure 8-1.

The initial attributes and data types for Transaction are:

- ▶ HoldingIdx — long integer (corresponds to Holding.indx autonumber data type)
- ▶ SequenceNumber — long integer
- ▶ Price — double (retained from the original Trade model. Versata has a “currency” data type, but since this model was taken from the original Trade model, currency is not used in this example)
- ▶ Quantity — double (retained from original Trade model)
- ▶ Date — Date/Time
- ▶ Amount — Double (retained from the original model)
- ▶ TransType — Integer.
- ▶ Commission — Double

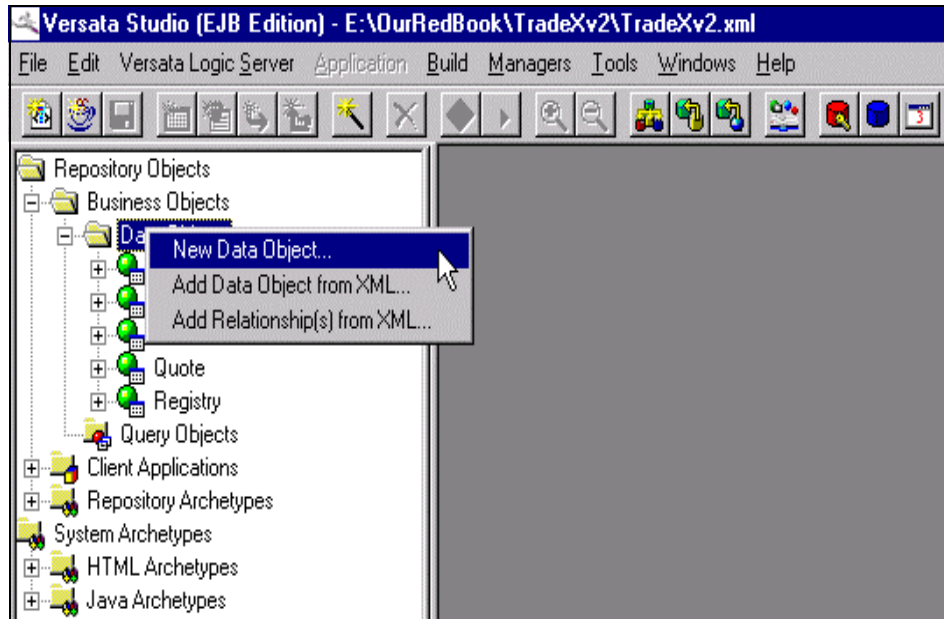


Figure 8-1 Adding a new data object

At the time the attribute is added basic rules for data validation can be specified. Data validation defines limitations for an attribute value that will be enforced at runtime as shown in Figure 8-2. Versata builds in validations based on:

- ▶ Condition — this includes anything that can be specified in the Rule builder such as, for Transaction.Quantity.

Quantity BETWEEN 10 and 10000 /\* Don't allow very small or very large trades:

- ▶ Value required — prevent nulls for this attribute
- ▶ User updates — allow users enter values for this attribute
- ▶ Against a list of values (called a Coded Value List). Coded Value Lists are demonstrated later in this section as we fill out the TransType attribute rules.



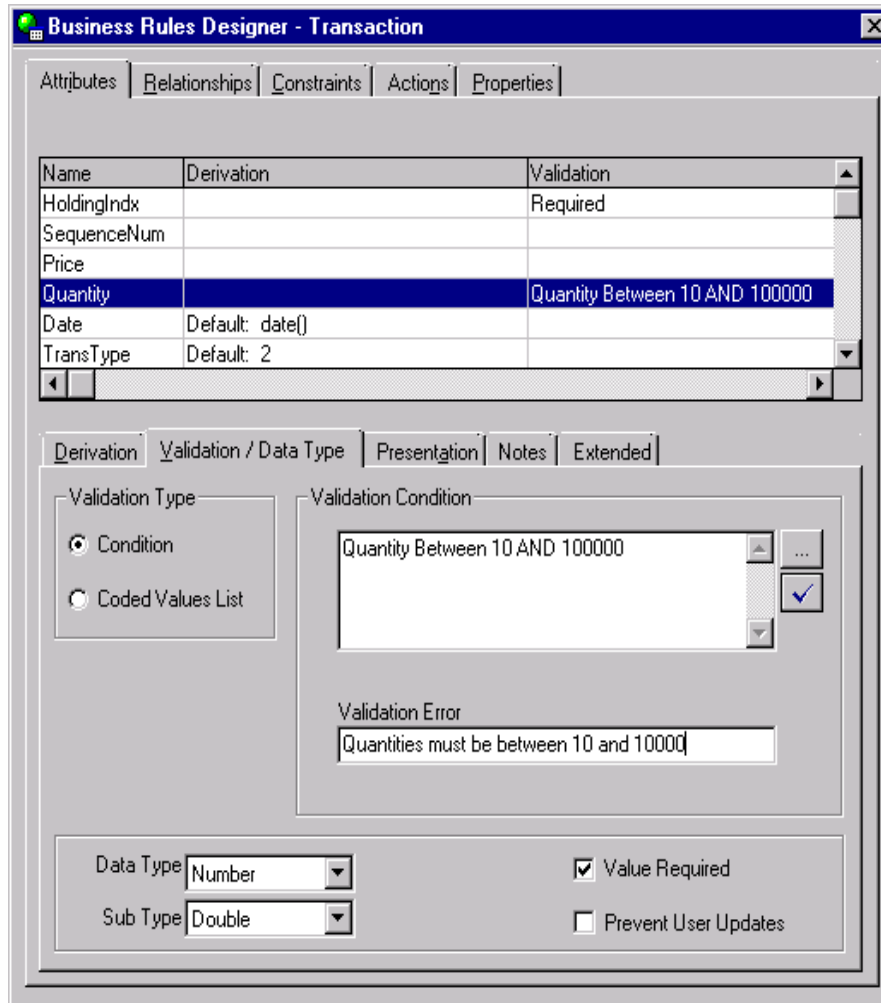


Figure 8-2 Adding transaction attributes and validations

With the basic Transaction object defined, next we define the primary and foreign keys which underlie relationship definitions as shown in Figure 8-3.

The primary key for Transaction (a combination of the HoldingIndx and SequenceNumber) uniquely identifies the Transaction. The foreign key (HoldingIndx) refers to the primary key of the Holding object (indx) to link the Transaction with the Holding.

The Business Logic Designer has an easy way to specify keys, which may or may not be propagated to the database (that is, keys can exist only in the “mid-tier” (EJBs), although performance is improved if they are deployed to the underlying database as well. This is done during database deployment from the Versata Studio.)

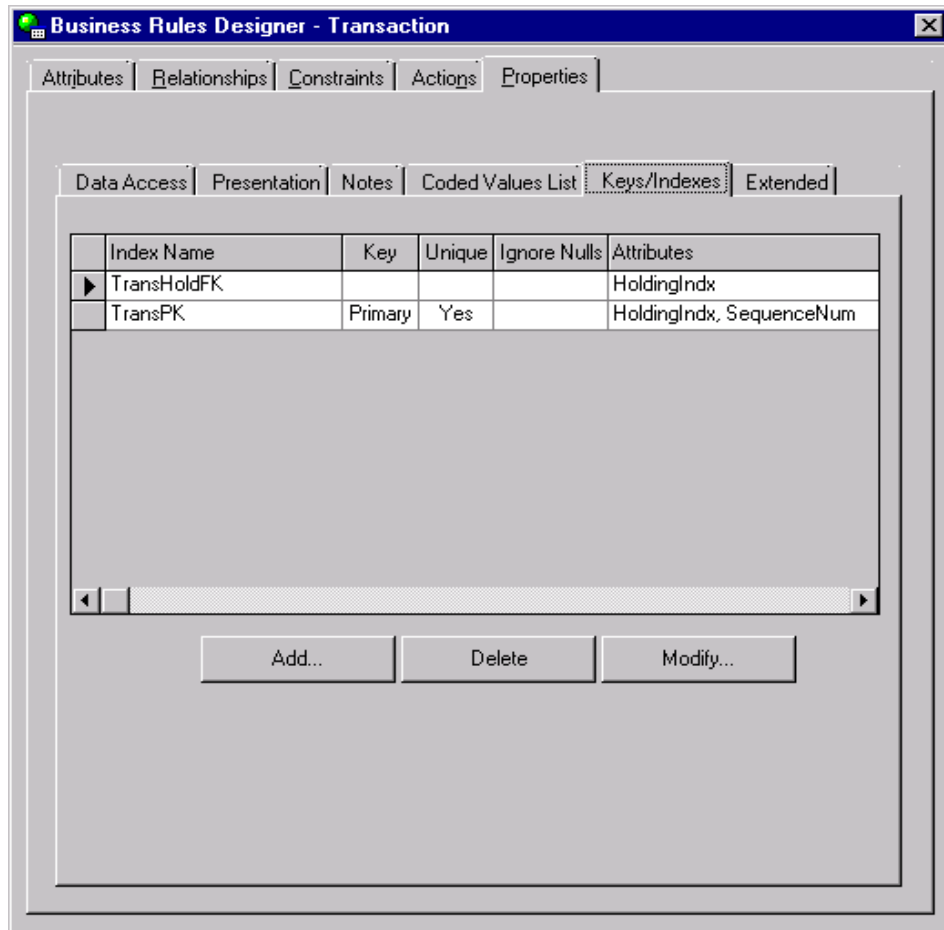


Figure 8-3 Defining transaction primary and foreign keys

With a foreign key defined for Transaction, we now create the relationship between Transaction (child) and Holding (parent) as shown in Figure 8-4.

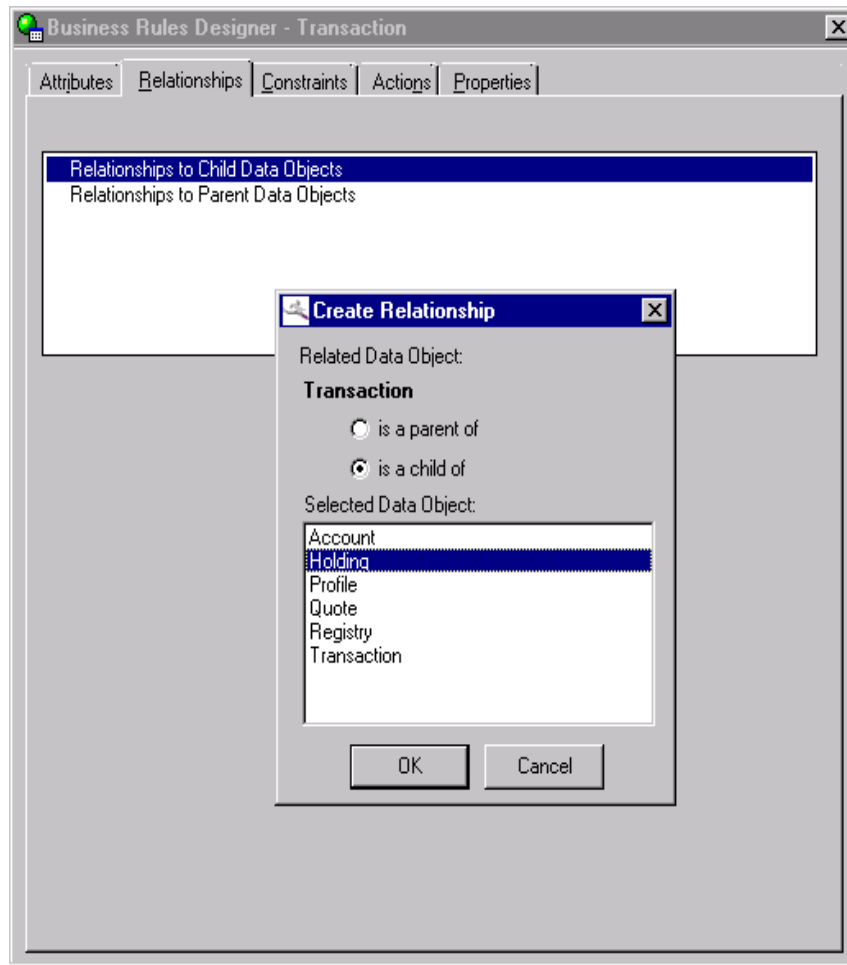


Figure 8-4 Defining the transaction and holding relationship

As we saw in Chapter 5, “Rule-based development” on page 57, part of the relationship definition is the relationship name (used to create a method that gets a related object), and referential integrity options as shown in Figure 8-5.

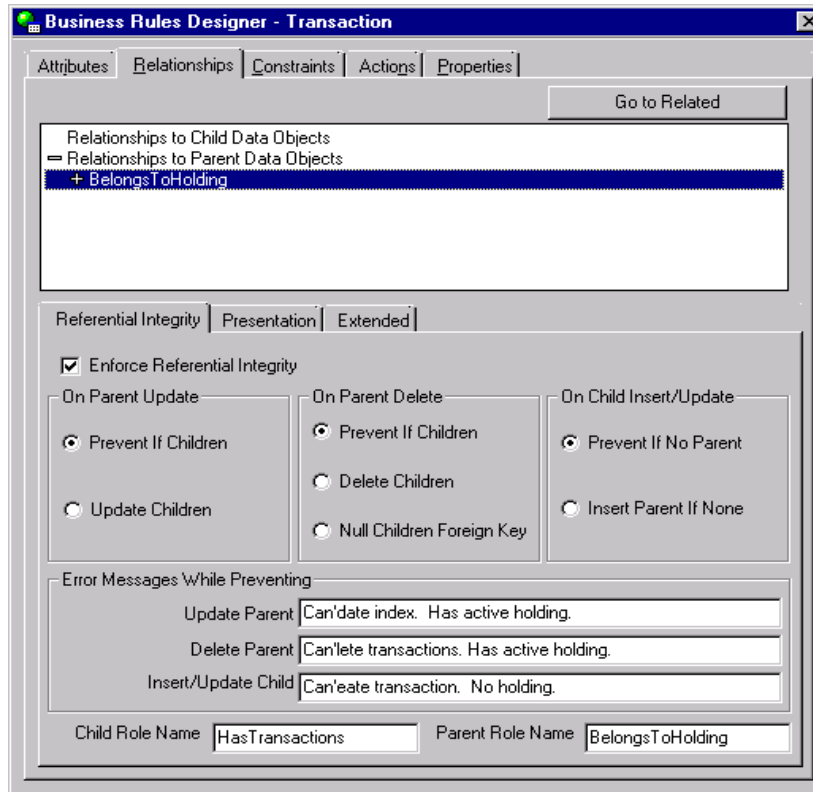


Figure 8-5 Transaction and holding relationship details

With the relationship defined, we can continue to define the rules required to implement the rest of our new transactions. These rules effect the Holding and Transaction object, and, to a lesser extend, the Account object.

### 1a: Requirement

Distinguish between different types of transactions. Define TransType 1 as “buy”, TransType 2 as “sell”.

### 1a: Implementation

Defining what the TransType values represent could simply be an informal understanding among the development team. Business logic developers could code accordingly, and client logic developers could translate the “1” and “2” values to something more meaningful to the user.

Versata does this automatically with a Versata Coded Value list, which is a table of values used to validate entries for an attribute and translate them to client applications. Validating TransType against a Coded Value List will help to document our logic, prevent errors, and allow us to easily add other transaction types in the future.

A simple way to implement this is to create another object (ValidTransType) with two attributes as shown in Figure 8-6:

- ▶ storedValue — the actual integer value saved in a Transaction
- ▶ displayValue — text (“buy” or “sell”) that will be used to describe it on HTML pages or other client applications.

The definition of a business object has a property to flag whether it will be used as a Coded Value List, as seen below.

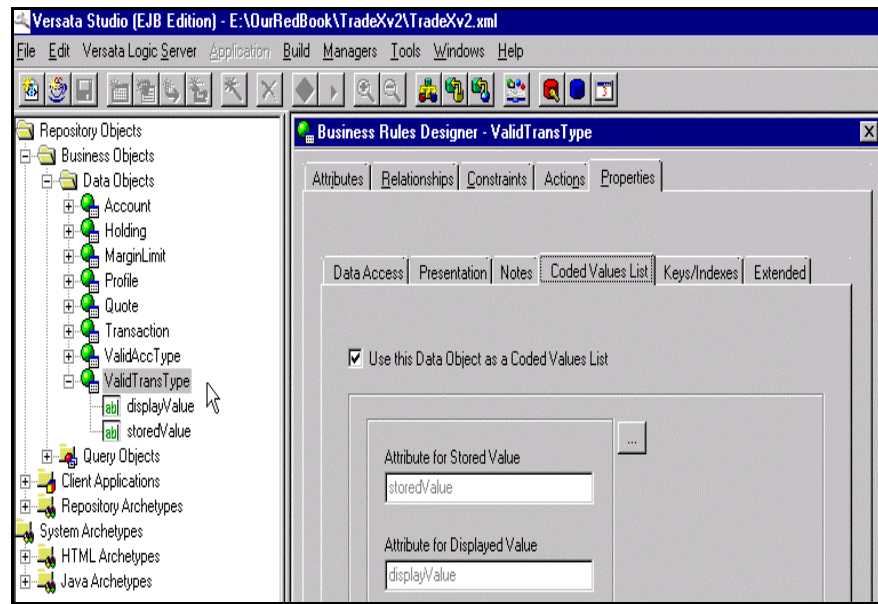


Figure 8-6 Defining the ValidTransTypes as a coded value list

With the new business object defined, we simply indicate that it is to be used for validating the Transaction, TransType attribute, as shown in Figure 8-7. To populate the corresponding database table with the valid values, we can enter them here. They will be pushed to the database the next time we deploy our object descriptions there (using the Deployment Manager -> Deploy to Database option.)

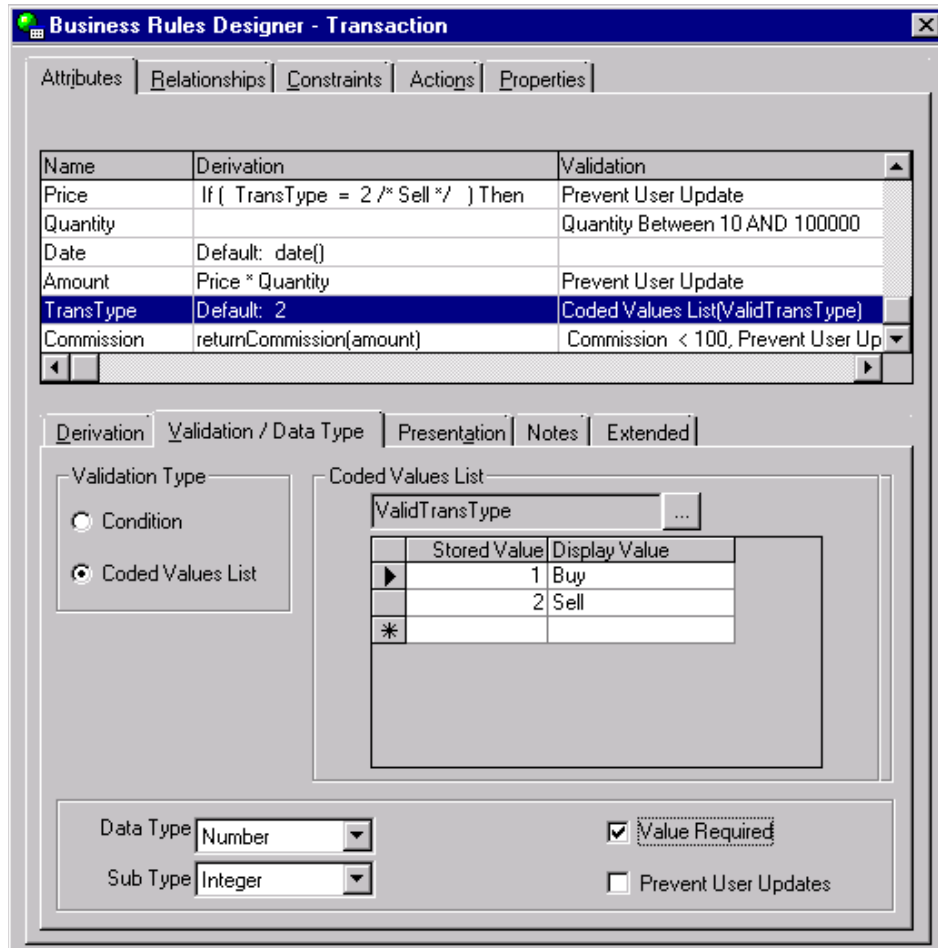


Figure 8-7 Validating transaction types from the coded value list

### 1b: Requirement

When a new Holding is added, create an initial “buy” transaction.

### 1b: Implementation

In the initial TradeX implementation, we had an event rule when inserting a Holding. The rule called a method to debit or credit the Account.balance with the amount of the Holding.

In our new implementation, we will replace this with an event that calls an `InsertBuyTrans()` method as shown in Example 8-1. This method is added to the Holding object's implementation (`HoldingImpl.java`). The method creates a new Transaction object and sets the `HoldingIndx`, `Price` and `Quantity` from values in the Holding object that is being inserted. It also sets the `Transaction Type` to 1 and the `Sequence Number` to 1.

*Example 8-1 New method*

---

```
public void insertBuyTrans() {
    Session s= getSession();
    try{
        TransactionImpl trans =
(TransactionImpl)TransactionImpl.getNewObject(s, true);
        trans.setHoldingIndx(this.getindx());
        trans.setPrice(this.getprice());
        trans.setQuantity(this.getquantity());
        trans.setTransType(1);
        //trans.setSequenceNum(1);
        trans.save();
    }
    catch(Exception ex){
        raiseException(ex.getMessage());
    }
}
```

---

The transaction system in the Versata Logic Server ensures that this logic is included in included in the transaction that inserts the Holding, so that Holdings and Transactions are committed or rolled back together.

### **1c: Requirement**

Inserting a transaction causes the `Account.balance` to be debited or credited with the transaction amount.

### **1c: Implementation**

To do this, we can just move the event rule to debit or credit the `Account.balance` from the Holding object to the Transaction object as shown in Figure 8-8.

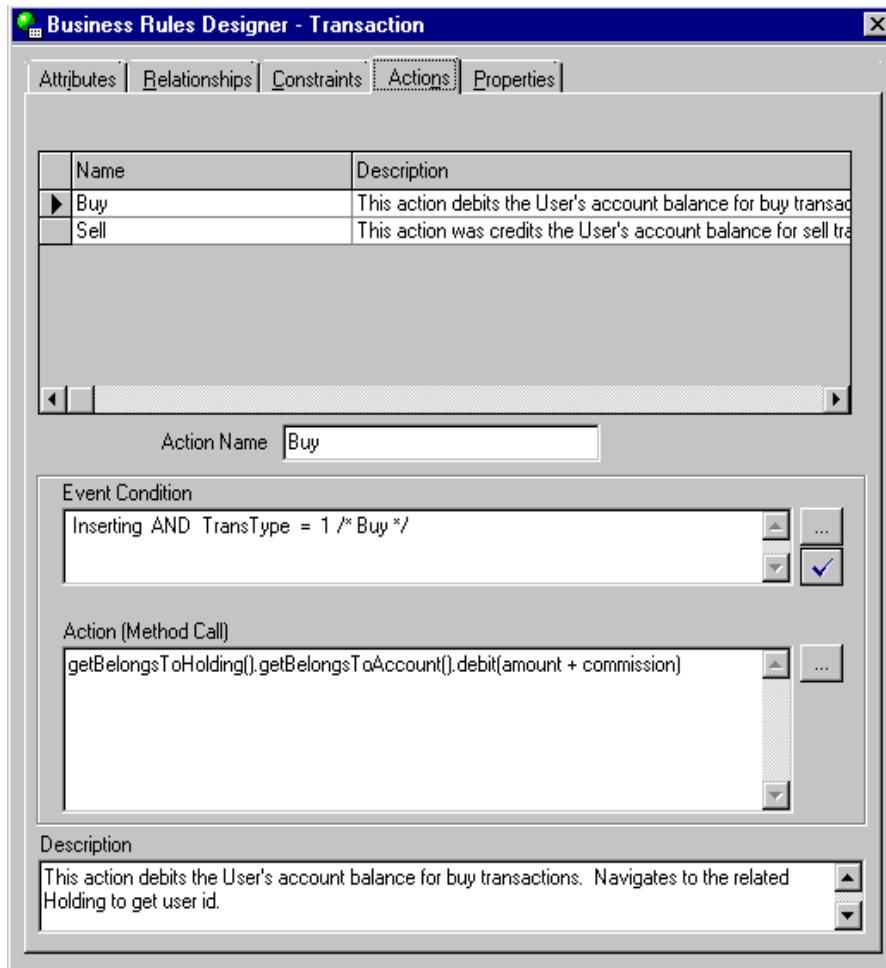


Figure 8-8 Debiting the account balance in a transaction

### 1d: Requirement

Each new Transaction for a Holding is assigned a SequenceNumber incremented by one. (This is similar to the Versata autonumber datatype, but the numbers are not incremented across the whole set of Transactions, only the Transaction for that Holding, for example, HoldingIndx 100, SequenceNum 1, HoldingIndx 100, SequenceNum 2, and so forth.



## 1d: Implementation

There are a number of ways to implement this pattern, but, for variety, here we will combine a rule on the Holding object (to count the number of related transactions) with a predefined “beforeInsert” event in the Transaction object.

First we add a new Holding attribute TransCount and set it to the Count of Transactions as shown in Figure 8-9.

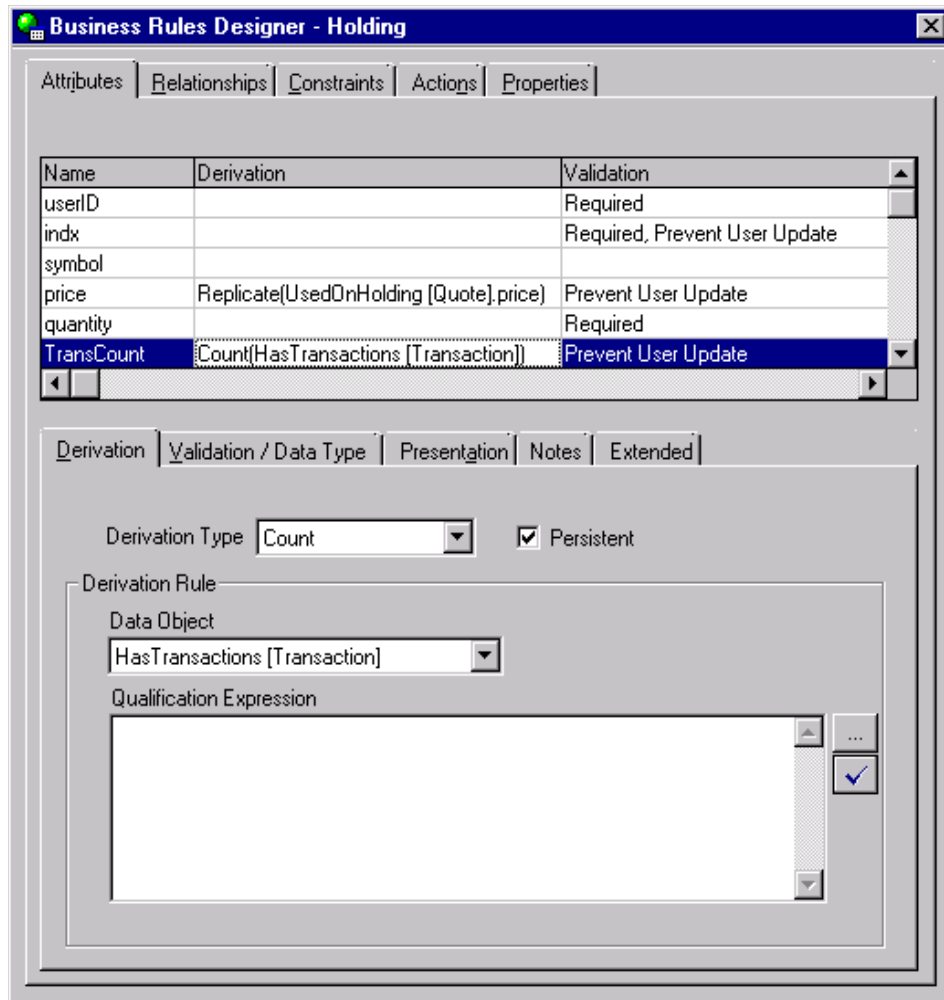


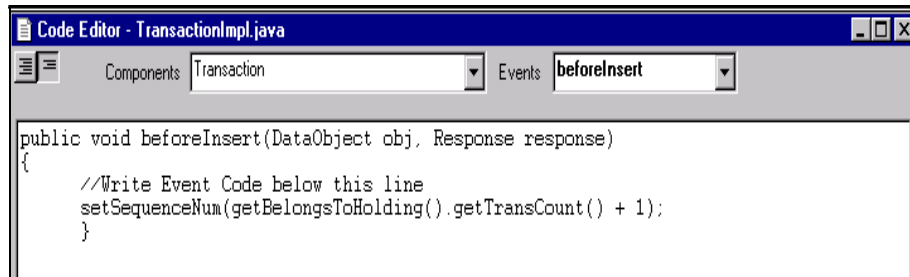
Figure 8-9 Rule that counts number of transactions for a holding

Next we add a single line of Java code to the TransactionImpl.java file. This line is inserted in the pre-built BeforeInsert event as shown in Figure 8-10.

The event model of Versata business objects is designed to simplify business logic customization and is an alternative to calling new methods from rules. For instance, our Holding event rule that calls insertBuyTrans() for each new Holding, could have used these pre-built events instead.

Pre-defined events and listeners are inherited when a business object is created. Exposed events for data objects include: afterCommit, afterDelete, afterInsert, afterQuery, afterRollback, afterUpdate, beforeCommit, beforeDelete, beforeInsert, beforeQuery, beforeResultSetFill, beforeRollback, and beforeUpdate.

Many developers, especially those coming from a VisualBasic or PowerBuilder projects, are instantly familiar with this model and prefer it to creating customer Java methods in business objects.



The screenshot shows a code editor window titled "Code Editor - TransactionImpl.java". The "Components" dropdown is set to "Transaction" and the "Events" dropdown is set to "beforeInsert". The code in the editor is as follows:

```
public void beforeInsert(DataObject obj, Response response)
{
    //Write Event Code below this line
    setSequenceNum(getBelongsToHolding().getTransCount() + 1);
}
```

Figure 8-10 Setting the sequence number inside an event

## Remaining transaction rules

The remaining rules to fully derive and validate the Transaction object attributes are:

- ▶ TransType — is defaulted to “2” (Sell).
- ▶ Date — is defaulted to the system method date().
- ▶ Amount — is calculated as Price \* Quantity
- ▶ Price — When selling price is retrieved from the Quote object by navigating the Transaction->Holding relationship and the Holding -> Quote relationship. (When buying, you will recall, it is set by the insertBuyTrans() method.) The rule for Price build in the Rule Designer is:

```
If ( TransType = 2 /* Sell */ ) Then
    $value = getBelongsToHolding().getUsedOnHolding().getprice()
End If
```

With these steps, we have nearly completed Requirement 1 — to be able to sell partial holdings by implementing a new transaction model. We just add one addition rule to ensure that users can't sell more shares than they have remaining in a holding as shown in Figure 8-11.

To do this, we add an attribute to Holding called QtyOnHand. We place a constraint rule on the attribute such that:

QtyOnHand < 0

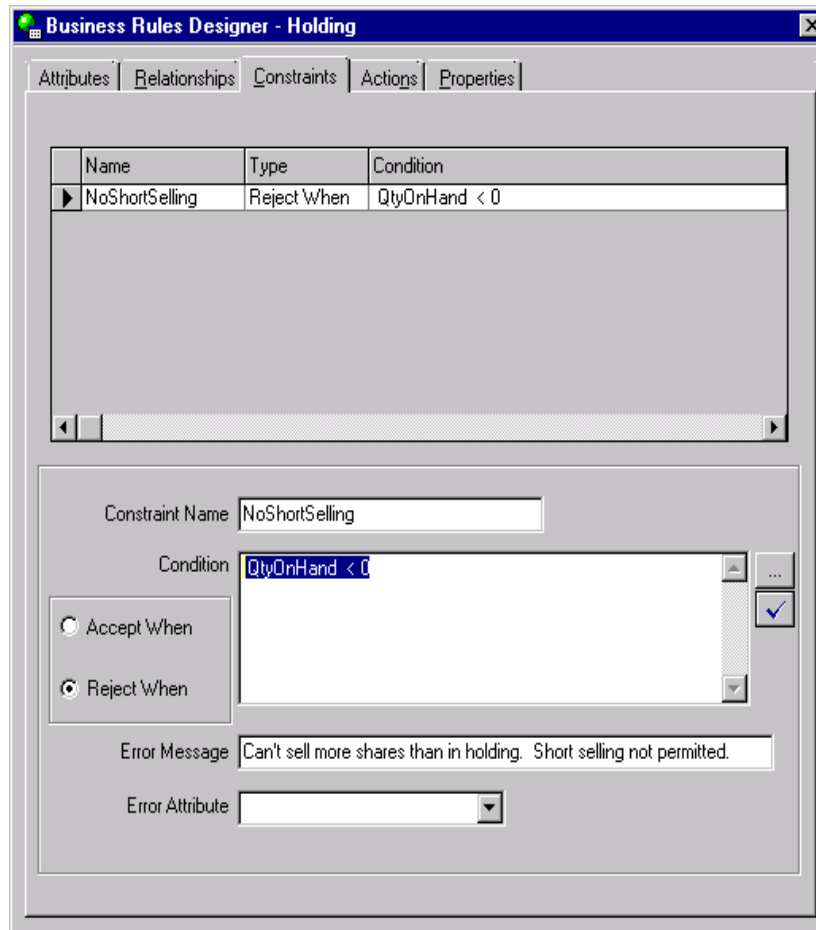


Figure 8-11 Rule that restricts short selling

To derive the Holding.QtyOnHand we create an attribute, as shown in Figure 8-12, QtySold (defined as the Sum of Transactions where TransType = Sold) and define the QtyOnHand as:

quantity - QtyS

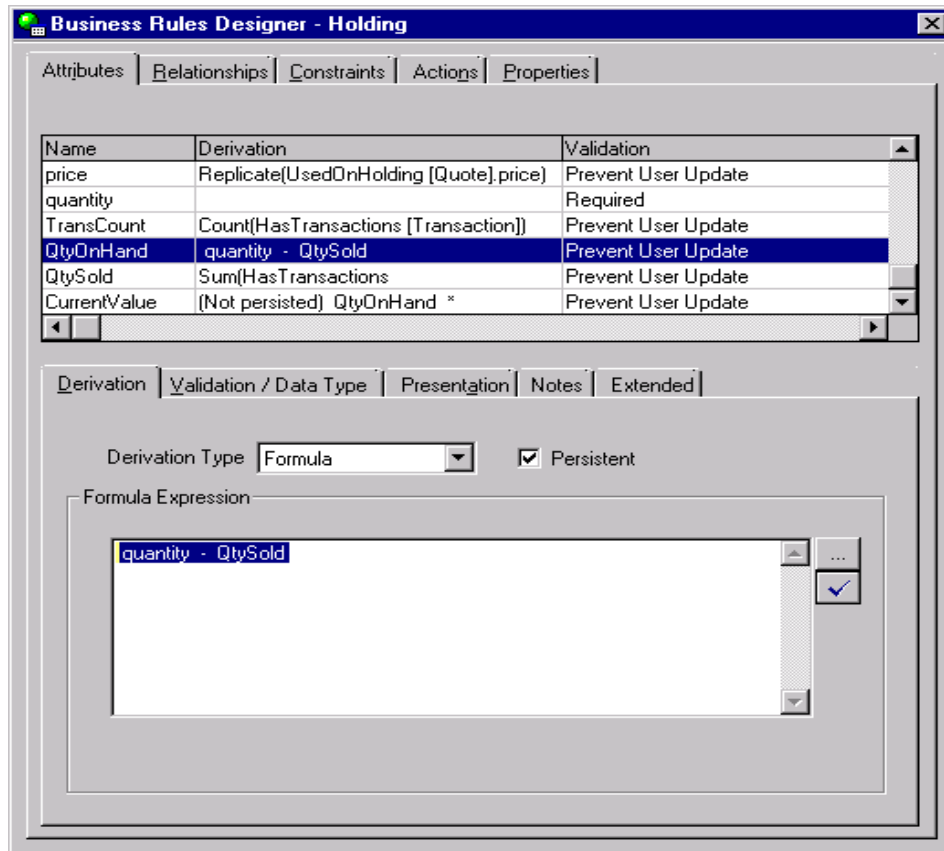


Figure 8-12 Deriving QtyOnHand of a holding

With that, we have completed Requirement #1, the most substantial change we will be making in the TradeX Version 2 application.

## 8.2.2 Requirement 2: Customize rules based on account type

Requirement 2, to establish a system of Account types, is much simpler to implement, since it doesn't require us to fundamentally alter our data and processing model.

As with Transaction types, we use a Coded Values List object - ValidAccType with the following stored and displayed entries:

- ▶ 1, New Account
- ▶ 2, Retail Account
- ▶ 3, Wholesale Account

The business logic about who is assigned what account type is:

- ▶ Anyone with less than five transactions is a New Account.
- ▶ Anyone with five or more transactions who has less than one hundred thousand dollars total assets (combined value of their current holding and their on-hand cash balance) is a Retail Account.
- ▶ Everyone else is a Wholesale Account.

The rule in the Business Rule Designer looks as shown in Figure 8-13.

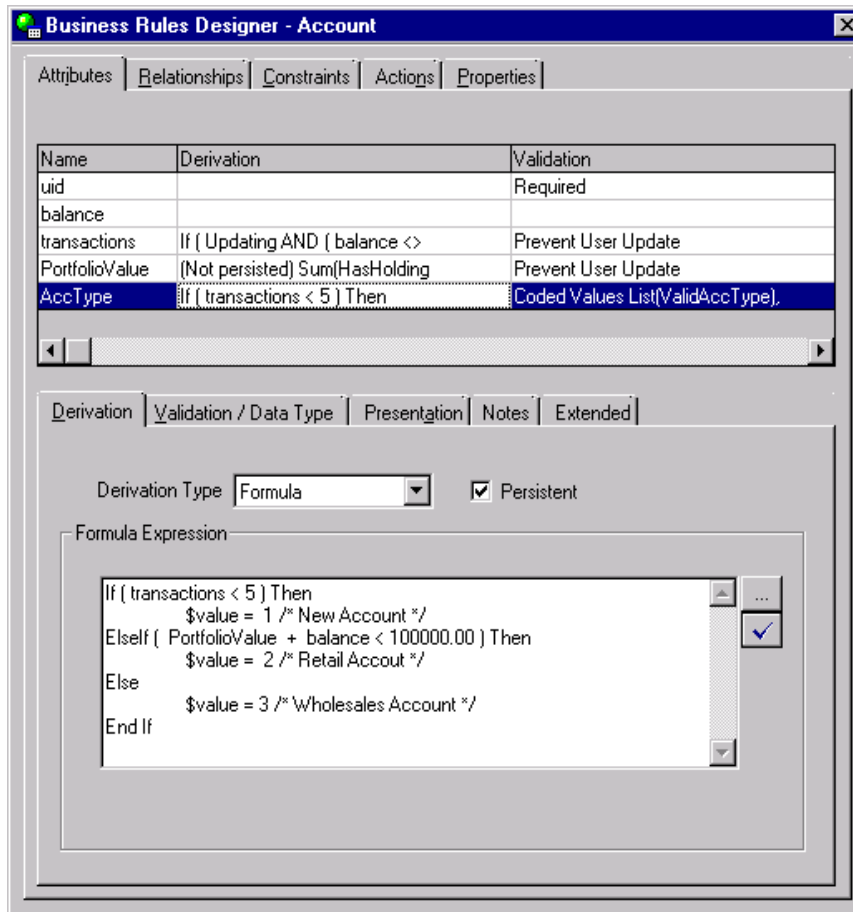


Figure 8-13 Deriving AccType from total assets and number of transactions

### 8.2.3 Requirement 3: Calculate commissions based on account type

With the `Account.AccType` in place, it is simple to derive a personalized commission based on the amount of the transaction and a formula defined in a simple Java method.

The method, added to the `Transaction` object is:

```
public double returnCommission(double amount)
{
    double commission;
    int accType =
    getBelongsToHolding().getBelongsToAccount().getAccType();
    if (accType == 1)
        commission = 0;
    else if (accType == 3)
        commission = .005 * amount;
    else
        commission = .01 * amount;
    return commission;
}
```

The formula for Commission simply calls returnCommission(), passing the amount as shown in Figure 8-14.

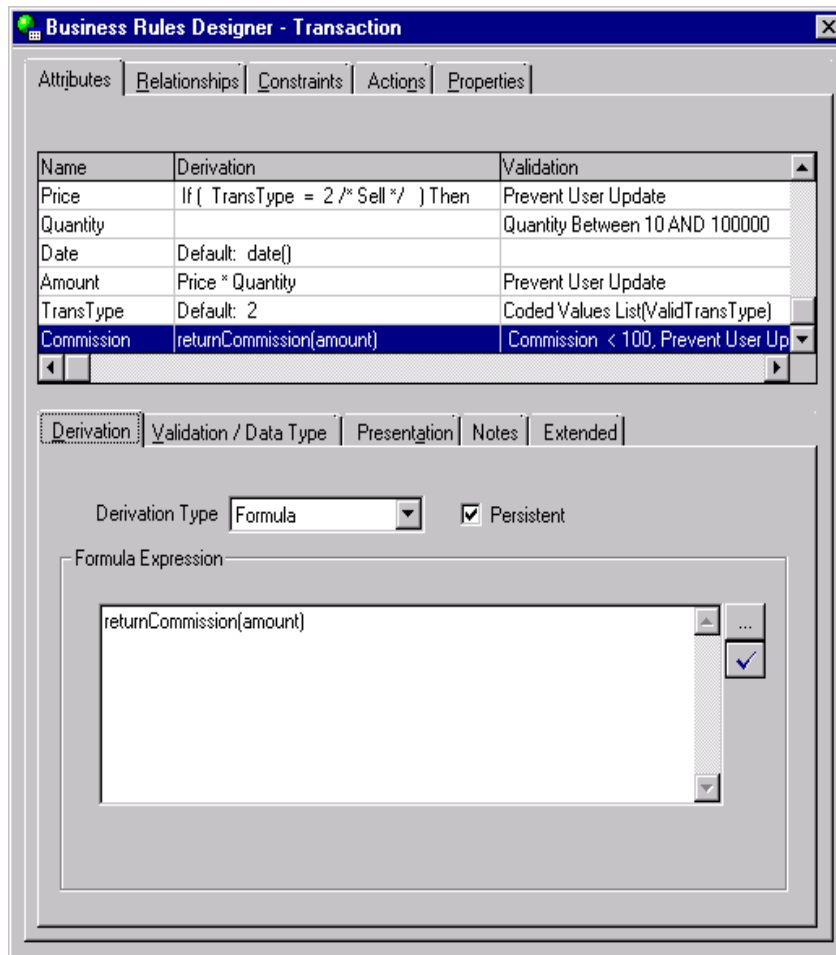


Figure 8-14 Calling a method to return the commission



## 8.2.4 Requirement 4: Limit margin selling

Managers can set limits on a running system (no recompilation of rules needed.)

Margin selling refers to whether the user can incur a negative balance in his account (effectively borrowing funds from the brokerage to make a purchase.)

Margin selling can be easily restricted by a constraint rule on the Account object that says:

Balance < 0 /\* Balance can never go below zero

However, our business requirement is a little more sophisticated than that. The requirement is:

Margin selling is restricted by a borrowing limit. An account balance can go below zero, but it can't go below the borrowing limit (MarginLimit). Margin Limits change frequently so that managers must keep a table of MarginLimits with the EffectiveDate, the OldLimit, and the NewLimit.

In other words, the MarginLimit rule is:

- ▶ Time-drive. Transactions will determine which MarginLimit is in effect at the time the transaction occurs.
- ▶ The rule has no fixed values, instead it looks up the current Margin Trading cut-offs from a table that can be updated in real-time.

Here is one implementation.

MarginLimits are defined in a new object and a corresponding database table. The definition is:

- ▶ MarginLimitType — Primary key. Allows for “tiers” of limits and ties to the MarginLimitType in the Account (the relationship between an Account and its MarginLimit is named UsesMarginLimit)
- ▶ EffectiveDate — Date this rule becomes effective
- ▶ OldLimit — Maximum overdraft allowed before the effective date
- ▶ NewLimit — Maximum overdraft allowed after the effective date

The constraint rule on Account will read:

**Reject when balance < getUsesMarginLimit().getEffectMinimum()**

The EffectMinimum being retrieved from the MarginLimit object is a non-persistent attribute, derived at runtime with a little Java method that compares the EffectiveDate against the transaction date and returns either the OldLimit or the NewLimit.

The complete sequence of illustrations shows:

1. The Constraint on Account, shown in Figure 8-15.

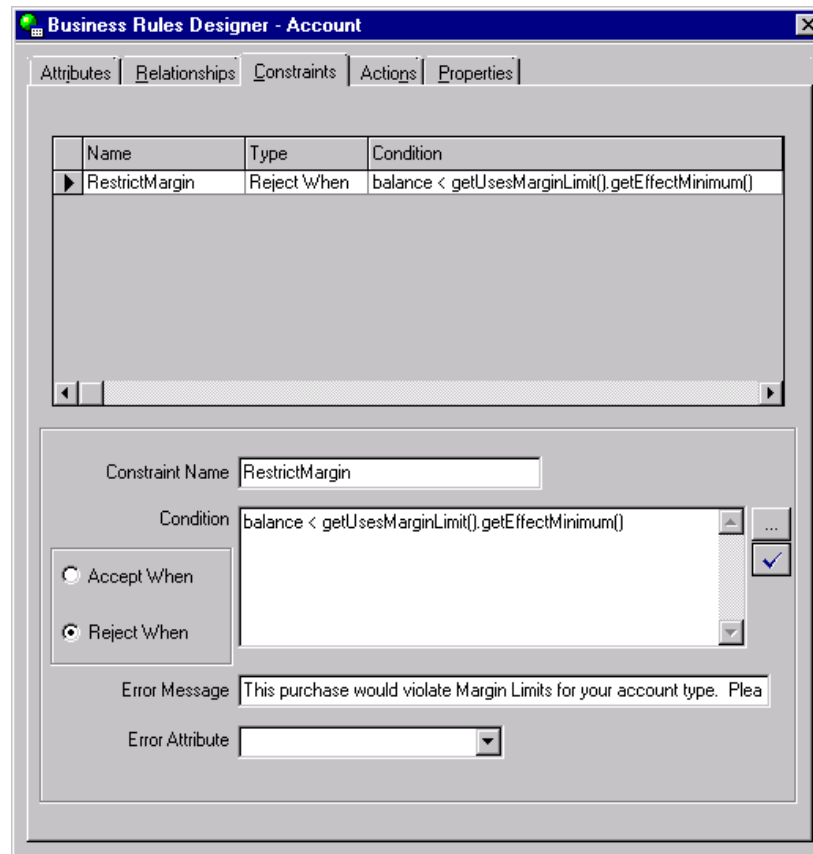


Figure 8-15 A constraint to limit margin selling

The non-persistent attribute in the MarginLimit object is derived from a Java method, to which we pass the EffectiveDate and the old and new limits as shown in Figure 8-16.

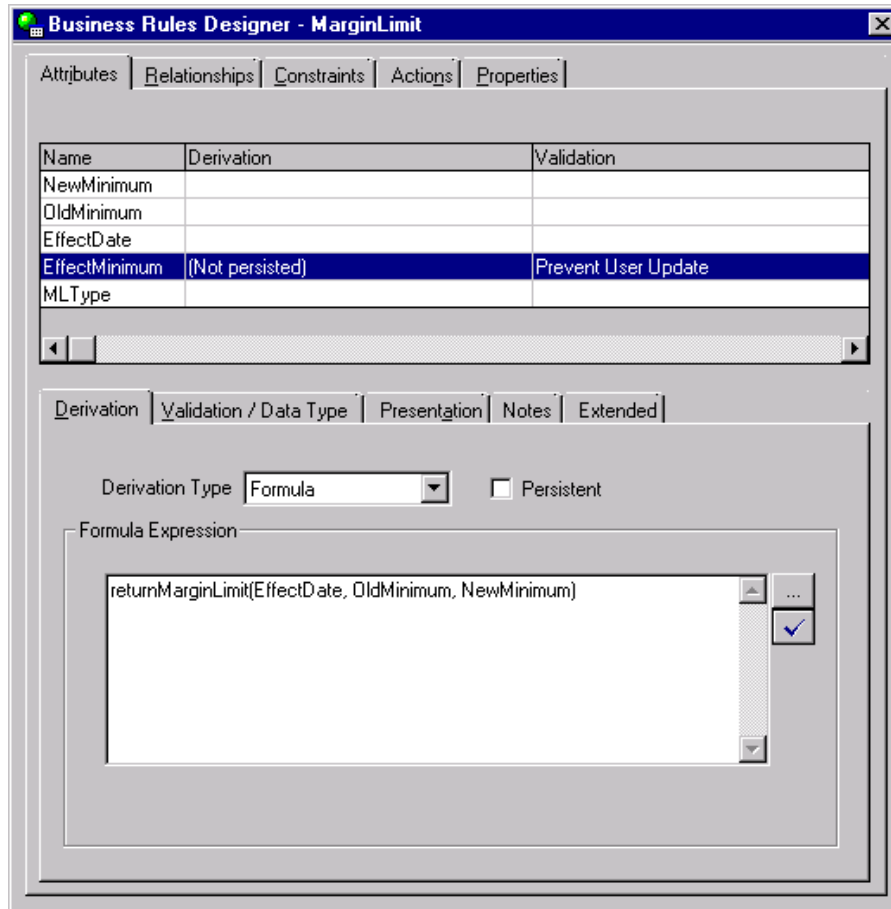


Figure 8-16 Passing attributes to a date comparison method

Finally, the Java method in the MarginLimitImpl.java file is as follows:

```
public double returnMarginLimit(VSDate adate, double oldmin, double
newmin){
    if (adate.after(adate))
        return newmin;
    else
        return oldmin;
}
```

**Note:** There are many possible implementations for the time-driven, table-driven rule pattern. This example breaks down the steps into the smallest increments possible for clarity.

## 8.3 Modified client application using new business logic

To demonstrate the new business logic, we now design a client application using the Versata Presentation Designer demonstrated in Chapter 6, “Designing an HTML client application” on page 73.

### 8.3.1 Capability 1: Creating QueryObjects

In the TradeXv2 application, we want to show the user a history of his “sell” transactions. Because the stock symbol isn't part of the Transaction business object, we must join a Transaction to its Holding to obtain this information. For this we define a QueryObject.

When talking about the types of Versata Business Objects in Chapter 4, “Architecture of the Versata Logic Server within WebSphere” on page 39, we mention that QueryObjects are similar to the J2EE compound or aggregate entity bean.

QueryObjects objects are derived from data objects by filtering data from a single entity-type object or by joining and filtering or deriving data from multiple objects. They can be thought of as the EJB version of a database “view”.

As with database views, in most cases, QueryObjects can read from, inserted or updated or deleted. Changes to a QueryObject are actually performed on the underlying DataObjects it represents.

The first step is to create a new QueryObject and specify the base objects from which it is derived as shown in Figure 8-17. The join can be based on any existing relationship.

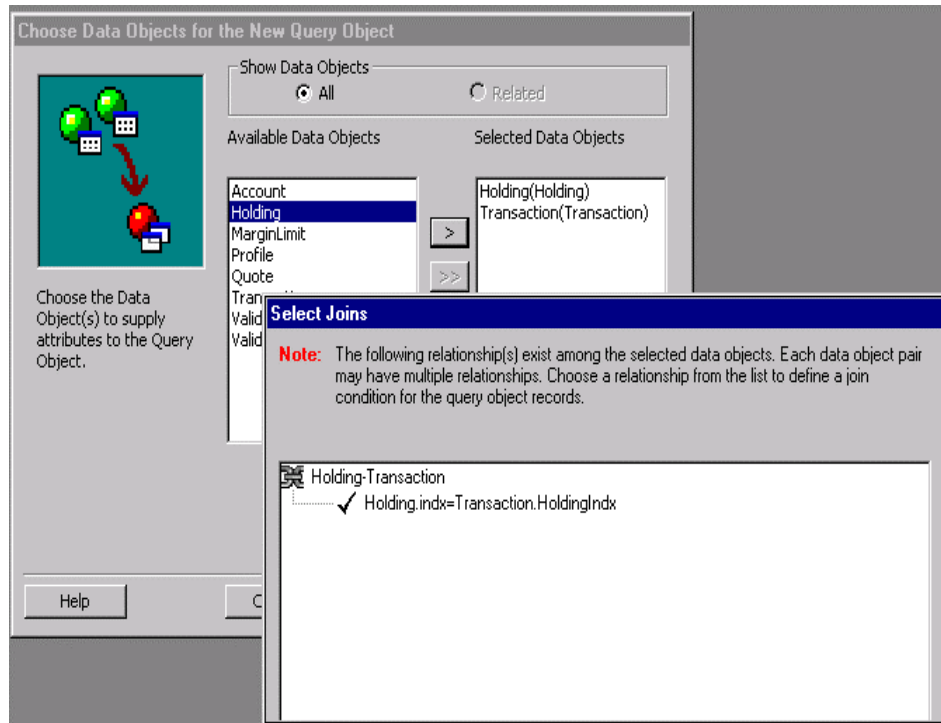


Figure 8-17 Defining the data objects to be joined in a QueryObject

Next we can limit the QueryObject to only the “Sell” transactions as shown in Figure 8-18.

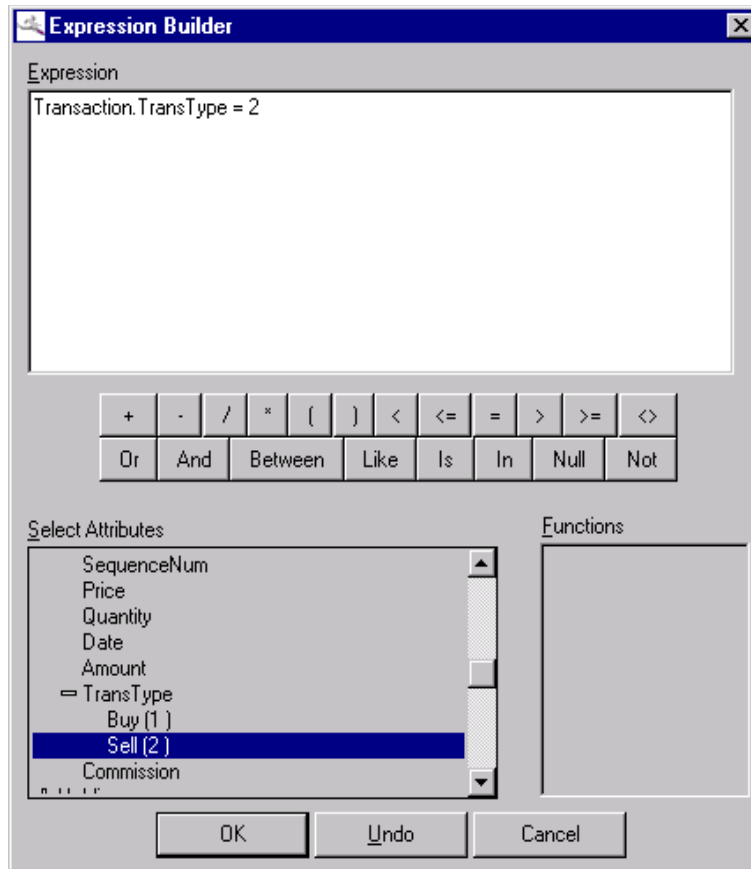


Figure 8-18 Restricting a QueryObject with an expression

A very useful feature of QueryObjects is the ability to compute new attributes at runtime from any combination of values from the underlying DataObjects. For instance, when displaying a user's transactions, is it useful to show whether the transaction was “profitable” — that is whether the amount of the sale of those shares is more than the amount of their purchase as shown in Figure 8-19.

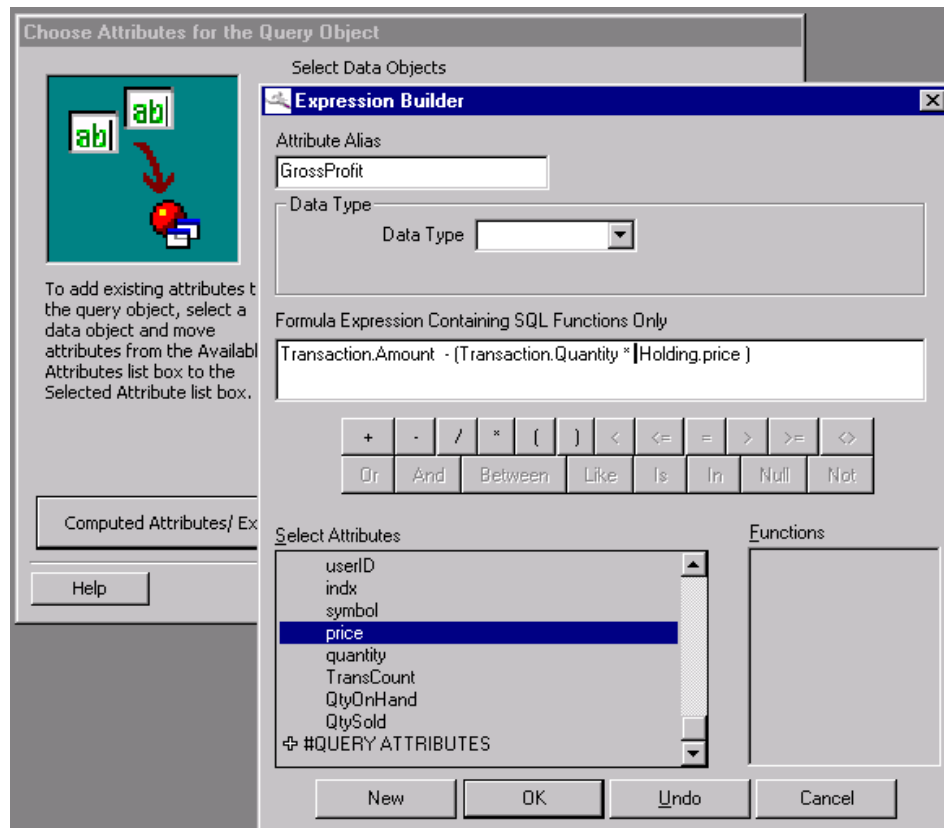


Figure 8-19 Computing a new GrossProfit attribute in the query

Finally, the sort order of the returned ResultSet can be defined as shown in Figure 8-20.

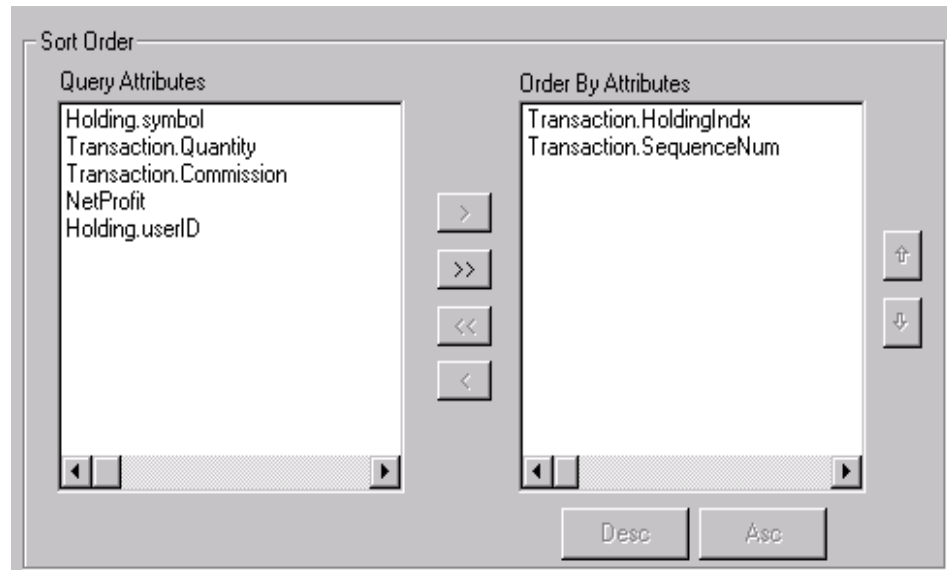


Figure 8-20 Defining the QueryObject sort order

### Other uses for Query objects

QueryObjects are a generally useful abstraction to separate the underlying business object model from its client representation. Many Versata designers base all of their HTML application design on QueryObjects rather than the underlying DataObjects. (In fact, if we had defined QueryObjects in the Chapter 5 exercise, and based our Client design in Chapter 6 on it, we could have run the TradeX application with few modifications against our new TradeXv2 design.)

We conclude this chapter by demonstrating the TradeXv2 application, with additional capabilities of the Versata Presentation Designer.



## 8.4 The TradeXv2 application

The TradeXv2 application, in Figure 8-21, is a new version of the Trade Home Page. The page shows another options for applications built with the Versata Presentation Designer. In this version of the Home Page, the user's Account information is displayed on the top of the page, and several “tabs” appear in the center of the page. The tabs allow the user to toggle through functions such as updating their profile, viewing their portfolio and viewing the new “Checks Profits” report, based on the new QueryObject.

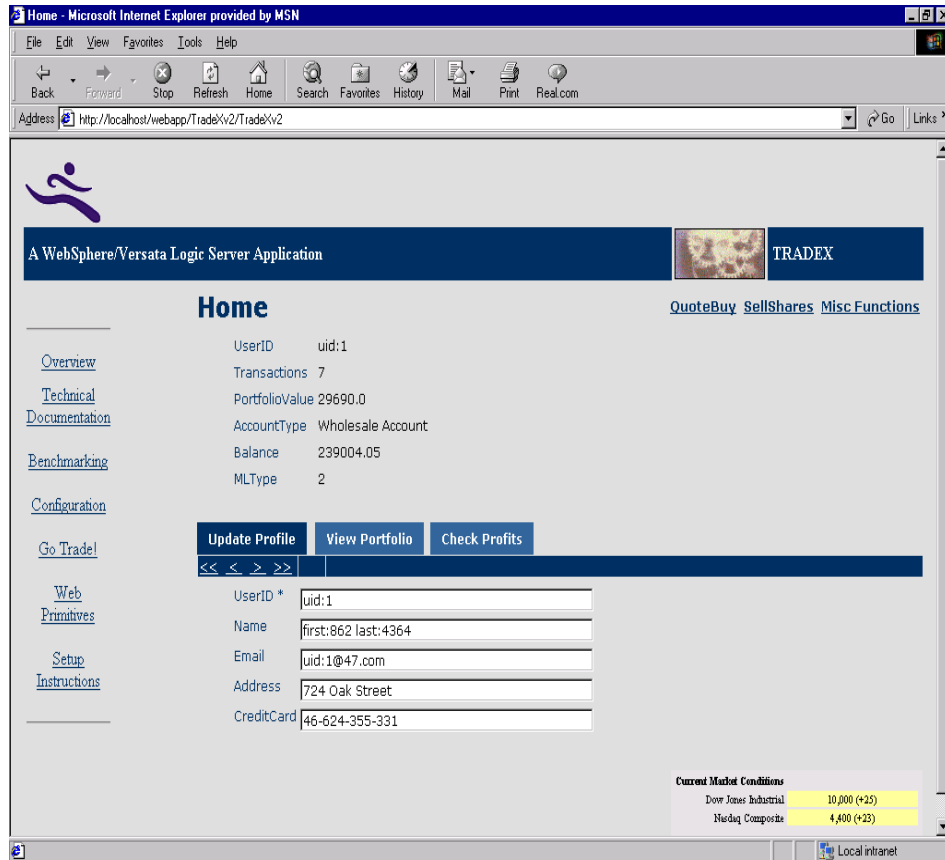


Figure 8-21 TradeXv2 home page

Here we see the report, as shown in Figure 8-22, based on the QueryObject that joins the Holding object with each Transactions and derives a NetProfit from the purchase price (Holding object), sales price (Transaction object) and the commission (Transaction object).

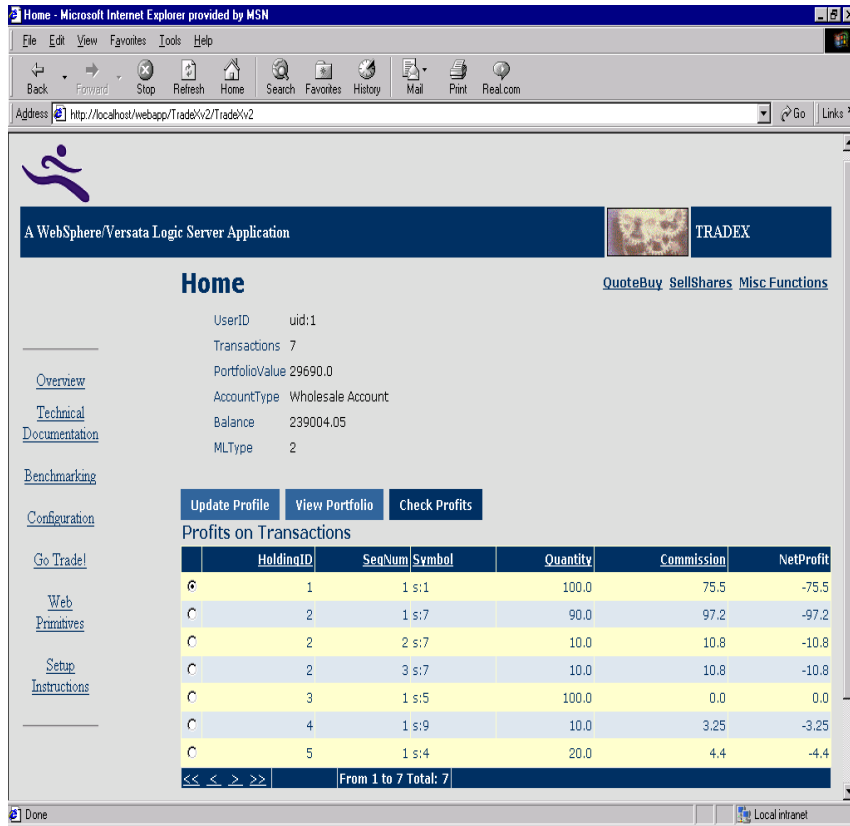


Figure 8-22 Home page tab is based on the TransProfit QueryObject

In addition, on this tab we see the result of many of the new business logic created with rules:

- ▶ The AccountType automatically changes when the transaction count exceeds five and the balance is greater than \$100,000.
- ▶ The commission rate for each trade is calculated based on the AccountType at that point in time, with a special provision for new account (with fewer than five trades).

When buying or selling shares we can see two other rules. The first is the data validation rule on Transaction.Quantity that prevents very small or very large sale as shown in Figure 8-23.

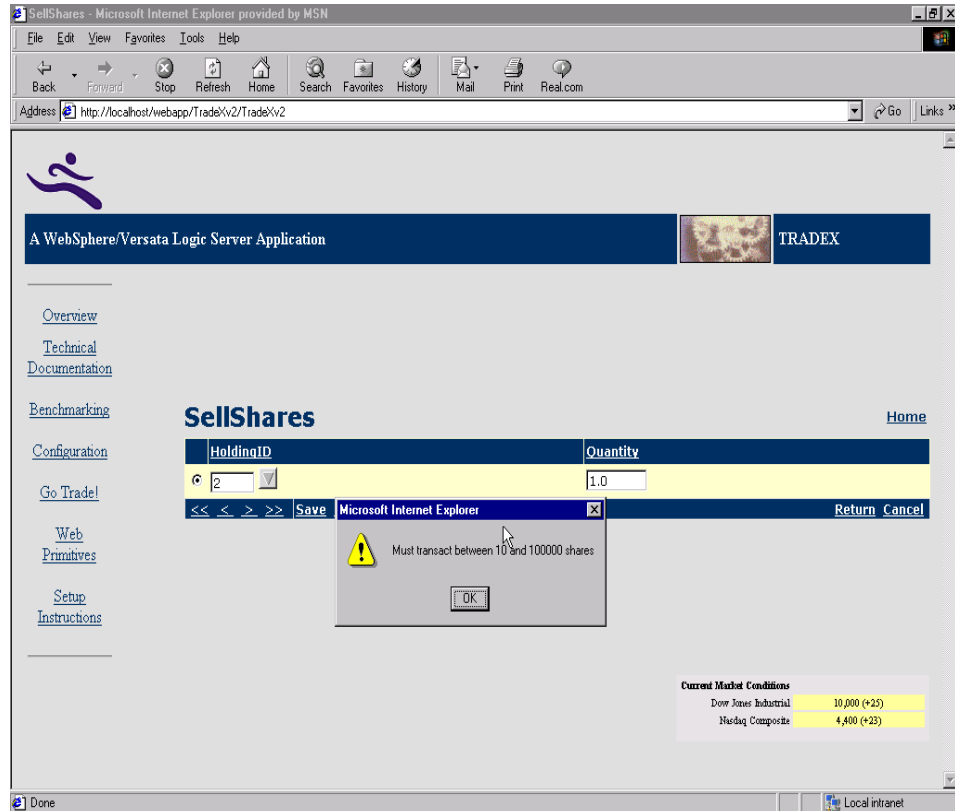


Figure 8-23 Violation of attribution validation rule on Transaction.Quantity

Finally, we see the result when the constraint on Account is violated: the Account.balance can't drop below the minimum balance, set in real-time by a manager, for that "tier" of customers.

## 8.5 Concluding the TradeX extended business logic

In this chapter we demonstrated how the original Trade2 business logic, which has very simple functionality, can be substantially enhanced with a simple succession of rule changes.

Next we will show how the business logic layer can be easily integrated with the existing Trade2 client components shipped from IBM.





## Integrating the IBM Trade2 client

In this chapter we demonstrate two options for interfacing the IBM-provided Trade2 client to the Versata Logic Server objects created in Chapter 5, “Rule-based development” on page 57.

The first option uses the Versata client libraries to access business objects directly from the Logic Server. The second option uses the remote interfaces for business objects that have been deployed by Versata as EJBs.

Although we outline both access options, we place special emphasis on the first method - accessing business objects through the Versata client libraries. This is because the Versata client libraries provide the basis for other client-side access techniques, such as those provided by the Versata JSP toolkit. Understanding the client libraries will give developers many more options for accessing Versata-automated business objects, beyond the functionality provided by their standard EJB-interfaces.

## 9.1 Method 1: Using the Versata client libraries

An overview of method one follows.

### 9.1.1 The TradeAltAccess class from IBM

As we reviewed in Chapter 2, “Trade application overview” on page 9, users access Trade2 through the TradeAppServlet. This servlet calls Java beans and supporting classes to initiate business functions such as buying and selling stocks.

Results from a business function is returned to an appropriate Java Server Page (JSP). In turn, each JSP presents a standard menu of Trade2 action choices. When the user makes a choice on the page, the action is forwarded back to the TradeAppServlet.

In the Trade client, the servlet and its supporting Java classes always mediate between the choice made by the user on a page, and the exact implementation of the business function. This indirection allows for the multiple modes of operation.

The primary mode of operation utilizes a VisualAge for Java access bean (the TradeAccessBean) to provide client-side access to the main Trade session EJB.

Two alternate modes of operation include calling the Trade EJBs without using access beans and interacting with the Trade database directly through JDBC. These are implemented in the TradeAltAccess class. TradeAltAccess also provides a convenient way to implement the Versata alternative access. We begin there to examine the interaction between the Trade client and the Versata Logic Server as shown in Figure 9-1.

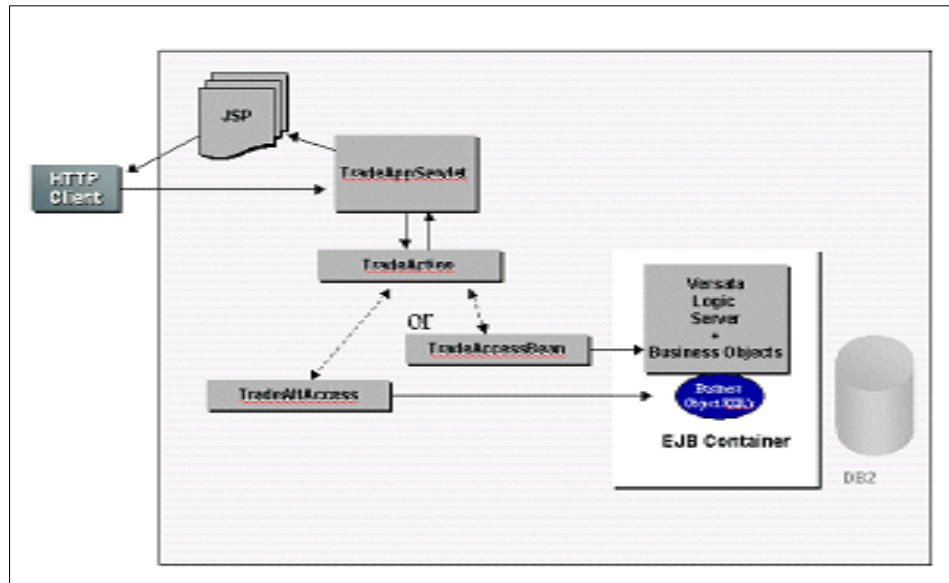


Figure 9-1 Processing Trade client requests to Versata business objects

## 9.1.2 Changes to TradeAltAccess to accommodate Versata

As provided by IBM, the Trade client implements a complete set of Trade business functions in the TradeJDBC class, which is called from TradeAltAccess. As the name implies, TradeJDBC implements its business logic by connecting directly to the Trade database using JDBC.

To add Versata connectivity to TradeAltAccess, we create a new class, TradeVFC. TradeVFC access its business logic by connecting to the Versata business objects through the Versata client libraries.

To accommodate the new class in the TradeAltAccess.java, we make the following changes:

```

{
trade =
    new TradeJDBC(TradeConfig.DS_NAME, TradeConfig.JDBC_UID,
    TradeConfig.JDBC_PWD);
}
  
```

becomes:

```

{
trade = new TradeVFC()
}
  
```

## 9.1.3 TradeVFC.java

TradeVFC implements the entire TradeInterface shown in Figure 9-2.

Method Summary	
double	<b>buy</b> (java.lang.String userID, java.lang.String symbol, double quantity) Purchase a stock and create a new holding for the given user.
QuoteObject	<b>createQuote</b> (java.lang.String symbol, double price, java.lang.String details) Given a market symbol, price, and details, create and return a new <a href="#">QuoteObject</a>
double	<b>getBalance</b> (java.lang.String userID) Return the user account balance for the specified customer
HoldingObject[]	<b>getPortfolio</b> (java.lang.String userID) Return the portfolio of stock holdings for the specified customer
java.util.Map	<b>getProfile</b> (java.lang.String userID) Return the user profile information for the specified customer
QuoteObject	<b>getQuote</b> (java.lang.String symbol) Return a <a href="#">QuoteObject</a> describing a current quote for the given stock symbol
int	<b>login</b> (java.lang.String userID, java.lang.String password) Attempt to authenticate and login a user with the given password
boolean	<b>logout</b> (java.lang.String userID) Logout the given user
java.lang.String	<b>register</b> (java.lang.String userID, java.lang.String password, java.lang.String fullname, java.lang.String address, java.lang.String email, java.lang.String creditcard, double initialBalance, boolean login) Register a new Trade customer.
void	<b>removeNewUsers</b> () Remove all newly registered users by TradeScenarioServlet (i.e.
void	<b>resetRegistry</b> (java.lang.String userID) Reset the Trade login registry for the specified user
double	<b>sell</b> (java.lang.String userID, java.lang.String symbol, int indx) Sell a stock and removed the holding for the given user.
void	<b>setProfile</b> (java.lang.String userID, java.lang.String fullName, java.lang.String Address, java.lang.String email, java.lang.String creditCard) Update a customers profile information with the given profile information

Figure 9-2 Trade interface

Each of the methods is implemented in a similar way:

1. A VSSession with the Versata Logic Server is obtained from a Logic Server session pool.
2. A new VSQuery is created for the session, specifying the business object to be accessed in the method.



3. A new `VResultSet` is initialized for the `VSQuery` from metadata about the object.
4. For methods that query objects, such as `getPortfolio()`, a `SearchRequest` is constructed from passed parameters. Then the query is executed, returning the object or sets of objects. Sets of objects are interacted over as `VRows`.
5. For operations that update or insert objects, such as `buy()`, a `VRow` in the `VResultSet` is filled with the passed parameters and the row is updated or inserted.

The following section shows the code for the `getPortfolio()`, and the `buy()` methods. In order to understand the code, here is some background on the Versata classes and interfaces used.

### **VSSession**

A `VSSession` object represents a logic session for each login with the Logic Server. The object constructor accepts parameters such as logic server instance (name) user login name, and logic password to establish the connection.

`TradeVFC` implements a private class, `SessionPool`, to connect a pool of `VSSessions` to the Logic Server. As dictated by the existing Trade client, this class uses a generic logon and password for the entire pool. Later, the `Trade login()` method will be called across this generic connection.

### **VSQuery**

`VSQuery` is the fundamental Versata client-side data access class. It opens a channel to retrieve data records from a VLS server and stores the data in a local instance of a `VResultSet`.

A `VSQuery` is typically constructed on an open session and is passed the business object name (also called the `MetaQuery` name) that will be accessed with the query. The object name and its associated metadata determines the shape of the result set. For example metadata describes: 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 and so on. Metadata can also describe 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.

Typically a `VSQuery` will be passed a `SearchRequest` which is used by the server to process the query. The Search Request will include the names and values of the query parameters being passed and any additional “where” clause needed to construct the query.

## **VResultSet**

A VResultSet provides client-side access to data records returned from a VSQuery. ResultSets offer a number of performance optimizations.

For instance, objects are fetched from databases in tunable blocks. This allows the developer to balance the database I/O benefit achieved when fetching fewer and larger blocks of objects with the increased network traffic that results from a large block size.

If more rows are retrieved than the ResultSet can display, they are cached in tunable buffers. Within the buffers, rows can be iterated over using methods such as `findFirst()`, `getRowAt()`, `next()`, and `previous()`. Also, within the ResultSet, row column values can be accessed in any order.

ResultSets are updateable. The column values for any number of rows in the set can be updated and held until the entire ResultSet is saved to the Logic Server. Unsaved changes support an `undo()` method, which re-sets the original values in the ResultSet.

ResultSets can be retrieved from Versata QueryObjects as well as DataObjects. QueryObjects allow multiple objects to be combined and retrieved as a single structure.

## **VRow**

A VRow represents a single data record on a VResultSet. Columns in the row can be accessed through a name or index. For performance, row data is returned by value, however, rows can be re-cast to Versata business objects for fine-grained manipulation. VRows support `insert()` and `delete()` operations within the ResultSet. Their metadata (column name, counts and types) can be queried.

### **9.1.4 The TradeVFC buy() method**

As an example of how the Versata client libraries can be used to insert new business objects, we walk through the `buy()` method in `TradeVFC.java`.

Part 1 - In the first section, the `buy()` method's signature is defined as shown in Figure 9-3. `buy()` accepts parameters for the `userID`, `symbol` and `quantity`. These are initially collected by the JSP page or application servlet. Next, a message is created if the application is set to verbose mode.

```
public double buy(String userID, String symbol, double quantity)
throws TradeException {
    if (verbose)
    {
        TradeLogging.logMessage(
            "TradeVFC.buy - Attempting buy("
            + userID
            + ","
            + symbol
            + ","
            + quantity
            + ")...");
    }
}
```

Figure 9-3 Declaring the buy() method

Part 2 - This section declares the VSSession and VSQuery variables, gets a session from the pool and creates a VSQuery for the Holding object on the current session as shown in Figure 9-4. It also creates empty ResultSet for the VSQuery.

```
VSSession session = null;
VSQuery query = null;

try
{
    session = sessionPool.getSession();
    query = new VSQuery(session, "Holding", "", "");
    VSResultSet rs = query.getNewResultSet();
}
```

Figure 9-4 Initializing the VSSession, VSQuery and VSResultSet

Part 3 - This section inserts an empty row into the VSResultSet and sets its value to the passed parameters as shown in Figure 9-5. GetData(), gets the column object by its name before setting it with the passed value.

```
VSRow row = rs.insert();
    row.getData("UserID").setString(userID);
    row.getData("Symbol").setString(symbol);
    row.getData("Quantity").setDouble(quantity);
```

Figure 9-5 Setting the values for the new row

Part 4 - This section sends the changed ResultSet to the Logic Server for rules processing. Then the query is closed. Within the Logic Server, after all rules are processed, changes will be committed to the database. as shown in Figure 9-6.

```
rs.updateDataSource();
    query.close();
```

Figure 9-6 Sending updated ResultSet to logic server

Part 5 - This section returns the new Account balance from the logic server. A new SearchRequest is declared for the Account object and a parameter is passed with the userID name/value pair. The query is executed and the first row is returned as shown in Figure 9-7

```
SearchRequest sr = new SearchRequest();
    sr.addParameter("Account","UID",userID);
    query = new VSQuery(session,"Account",sr,new SearchRequest());

    rs = query.execute();
    row = rs.first();
    if (row != null)
        return row.getData("Balance").getDouble();
}
```

Figure 9-7 Returning the new account balance

Part 5 - Finally, errors are caught and handled and the session and the query, if still open, is closed as shown in Figure 9-8.

```
catch (Exception e) {
    //create a tradeexception so that other classes know that this has already
    //be dealt with (error messages etc) and throw it for handling at the appropriate level
    TradeExceptionWrapper e2 =
    new TradeExceptionWrapper(
        e,
        "trade.TradeVFC.buy(...), error while creating holding");
    e2.addMessage("symbol " + symbol);
    e2.addMessage("user " + userID);
    e2.setSeverity(9);
    e2.userError = false;
    throw e2;
    } finally {
        if (session != null)
            sessionPool.releaseSession(session);
        if (query != null)
            query.close();
    }
    return 0;
}
```

Figure 9-8 Handling potential exceptions

## 9.1.5 The TradeVFC getPortfolio() method

The other methods in TradeVFC can be implemented using the same general approach. Here is the method to get the user's portfolio (a set of all of his Holdings.)

Part 1 - getPortfolio() is passed the userID and returns an array of Holding objects. In addition to initializing a new VSSession, VSQuery and SearchRequest, a vector is created to hold the rows of data elements as they are returned as shown in Figure 9-9. (A vector is used because we don't yet know the number of rows to be returned.)

```
public HoldingObject[] getPortfolio(String userID)
    throws RemoteException, TradeException {

    Vector v = new Vector();

    SearchRequest sr = new SearchRequest();
    sr.addParameter("Holding", "UserID", userID);
    VSSession session = null;
    VSQuery query = null;
```

Figure 9-9 Beginning the getPortfolio() method

Part 2 - A session is assigned from the pool, the query is executed and the first row is returned as shown in Figure 9-10.

```
try {
    session = sessionPool.getSession();
    query = new VSQuery(session, "Holding", sr, new SearchRequest());
    VSResultSet rs = query.execute();
    VSRow row = rs.first();
```

Figure 9-10 Retrieving the first row of holdings

Part 3 - Each row in the ResultSet is processed, getting the value of each element and adding it to the vector as shown in Example 9-1.

Example 9-1 Iterating over the ResultSet

```
while (row != null) {

    v.addElement(new HoldingObject(row.getData("UserId").getString(),
        row.getData("indx").getInt(),
    row.getData("Symbol").getString(),
    row.getData("Price").getDouble(),
    row.getData("Quantity").getDouble()));
    row = rs.next();
}
```

Part 4 - Finally, an array of HoldingObjects of the correct size is allocated, populated and returned as shown in Example 9-2.

*Example 9-2 Return the array of HoldingObjects*

---

```
    HoldingObject[] retVal = new HoldingObject[v.size()];  
    v.copyInto(retVal);  
    return retVal;
```

---

## 9.2 Method 2: Utilizing EJB interfaces

The alternative to accessing business objects through the Versata client libraries is to access them through their EJB interfaces.

As we saw in Chapter 5, each business object has a property that instructs the Versata Studio to deploy an EJB for that object. When deployed as an EJB, the system constructs the object's Home and Remote Interface.

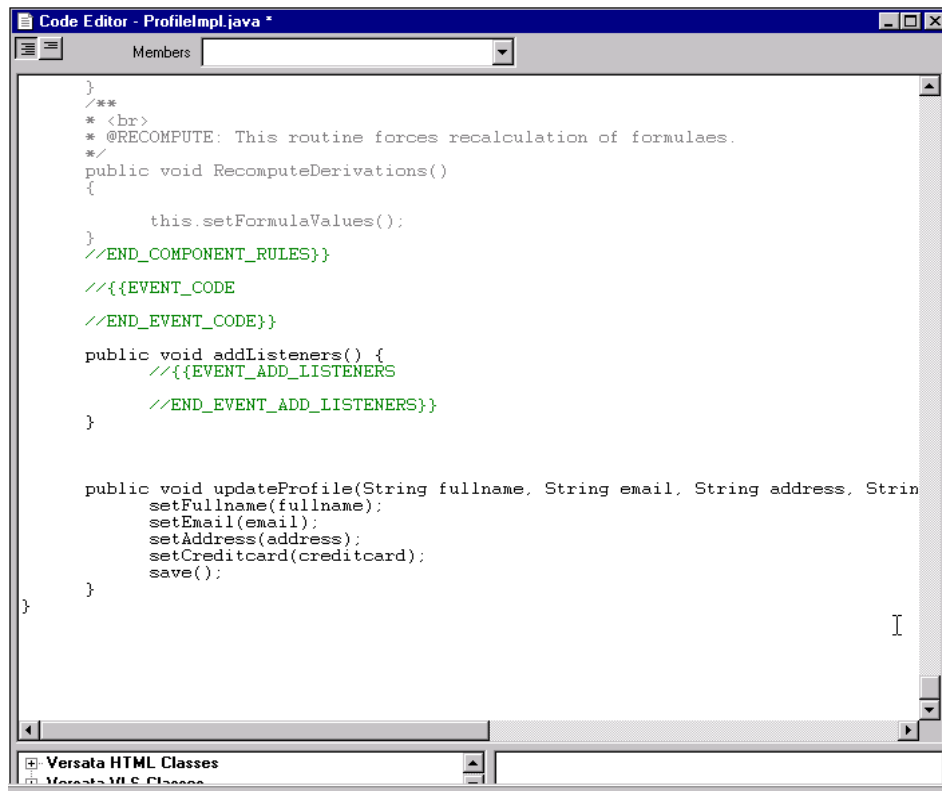
The Home interface will automatically include methods to create the object and find the object by its primary key or by a SearchRequest.

The Remote Interface will initially be empty. The developer should add those methods that will be used remotely to the Remote Interface file, usually by copying methods from the business objects generated Java class.

In the Trade application provided by IBM, most business functions are controlled by a single session EJB - the TradeBean. TradeBean implements each of the business methods in the Trade EJB remote interface and several EJB lifecycle methods.

Since the existing application defines the method for each business function, the Versata business object functions were slightly adapted to match these functions. For example, TradeBean implements the setProfile() method by calling an remote interface on the Profile object called updateProfile(). To provide the same functionality, an updateProfile method was added to the Versata business object. This method "wraps" other methods automatically provided in the business object, which set the object's attributes.

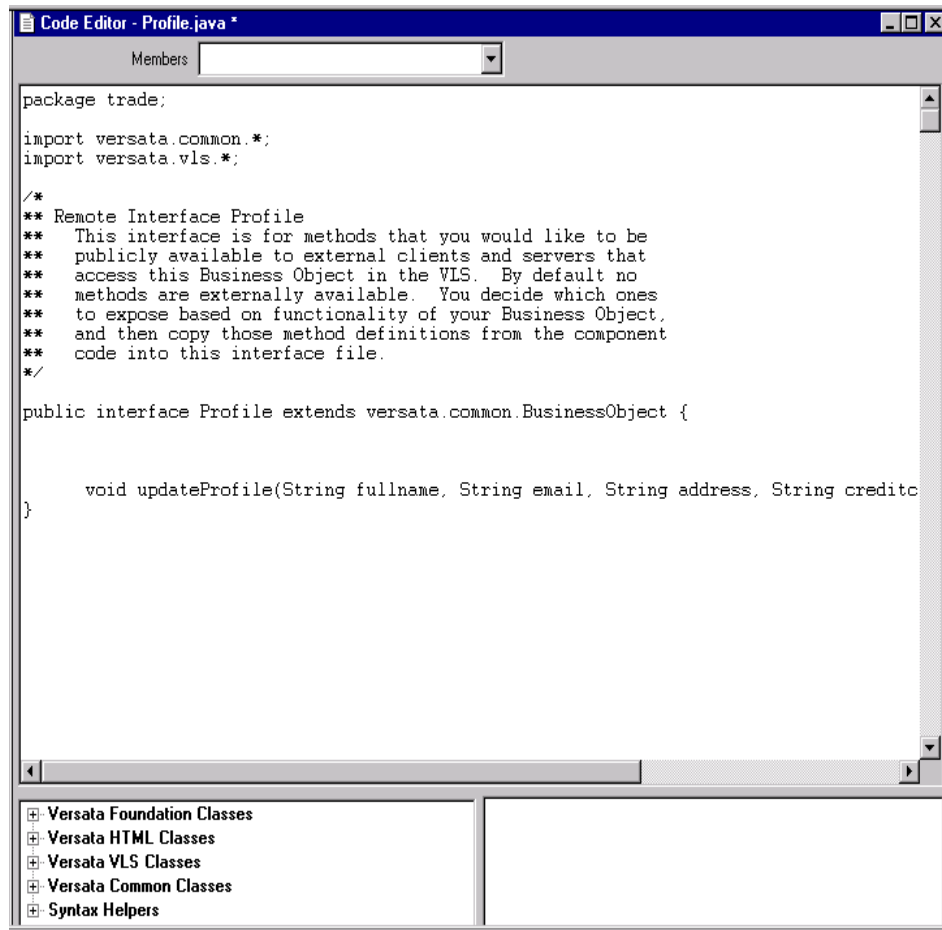
Figure 9-11 shows the business object method as it is implemented in the Versata Studio.



```
Code Editor - ProfileImpl.java *
Members
}
/**
 * <br>
 * @RECOMPUTE: This routine forces recalculation of formulaes.
 */
public void RecomputeDerivations()
{
    this.setFormulaValues();
}
//END_COMPONENT_RULES}}
//{{EVENT_CODE
//END_EVENT_CODE}}
public void addListeners() {
    //{{EVENT_ADD_LISTENERS
    //END_EVENT_ADD_LISTENERS}}
}
public void updateProfile(String fullname, String email, String address, Strin
    setFullname(fullname);
    setEmail(email);
    setAddress(address);
    setCreditcard(creditcard);
    save();
}
}
I
Versata HTML Classes
Versata MS Classes
```

Figure 9-11 The updateProfile method added

After defining the method in the business objects implementation file, it can then be included in the Remote Interface created by Versata as shown in Figure 9-12.



```
Code Editor - Profile.java *
Members
package trade;

import versata.common.*;
import versata.vls.*;

/**
** Remote Interface Profile
** This interface is for methods that you would like to be
** publicly available to external clients and servers that
** access this Business Object in the VLS. By default no
** methods are externally available. You decide which ones
** to expose based on functionality of your Business Object,
** and then copy those method definitions from the component
** code into this interface file.
**/

public interface Profile extends versata.common.BusinessObject {

    void updateProfile(String fullname, String email, String address, String creditc
}

Versata Foundation Classes
Versata HTML Classes
Versata VLS Classes
Versata Common Classes
Syntax Helpers
```

Figure 9-12 Adding the business object method to the remote interface

After defining the business object methods required by the TradeBean, and after copying them to the object's remote interface file, the Trade application can be run without modification in EJB mode.



## 9.3 Alternative for JSP access: Versata JSP Toolkit

Because the Trade client uses servlets and Java beans to access business logic, the Versata JSP Toolkit was not used in these redbook examples. It does, however, provide an easy-to-use alternative for developers who need to integrate Java Server Pages with business objects managed by the Versata Logic Server. This section provides a description of the Toolkit for developers looking for a simpler client model than the one provided with the Trade application.

The Versata JSP toolkit encapsulates many Versata client library functions in a custom JSP Tag Library and supporting Java classes. Downloadable from the Versata Developer's Web site, the Toolkit is used by customers who want to access Versata business objects from JSP-based e-commerce applications, from user-developed JSP pages, or from other content-delivery systems.

The JSP 1.1 specification supported by the Versata JSP Toolkit provides a number of approaches to retrieving and controlling the dynamic content of pages. The toolkit supports each of these approaches including:

- ▶ Inline Java coding. With the Versata JSP Toolkit, you can access the VLS with inline Java code embedded within the HTML in a JSP page.
- ▶ JavaBeans. With the Versata JSP Toolkit, you can access the VLS from JavaBeans referenced within JSP pages. JavaBeans offer sophisticated data control and binding to GUI elements. A default JavaBean, called VLSBean, comes bundled with the Versata JSP Toolkit, and issued for interacting with the VLS.
- ▶ Custom tags (also known as tag extensions). Custom tag libraries are a feature of JSP 1.1. Resembling XML tags embedded in the JSP page, they provide an alternative to writing code in JSP scriptlets in order to invoke Java-based functionality. The Versata JSP Toolkit provides several custom tags which:
  1. Establish and authenticate VLS connections
  2. Provide access to business objects controlled by the VLS
  3. Support display and updates of VLS ResultSets

### 9.3.1 Supported functionality

For connecting from JSPs to the Logic Server, the Toolkit supports:

- ▶ Session persistence across pages
- ▶ Connection recovery
- ▶ Result set generation from Where clause, Order By clause, and metaquery definition (in-line scripting)
- ▶ Remote method calls (through VLSBean or in-line scripting)
- ▶ Caching of result sets across pages
- ▶ Security session creation
- ▶ Session-level failover through application server clustering
- ▶ Load balancing across multiple VLSs

In addition, it supports runtime binding between VLS data sources and the GUI controls on HTML pages through the:

- ▶ Declarative access to result sets, without a need to access the VSResultSet class (implemented through Versata custom tags)
- ▶ Ability to override data queries in order to modify result sets or set up alternate result sets (implemented through subclassing)
- ▶ Client caching of result sets to allow user navigation
- ▶ Navigation through result sets by block and by individual record (implemented through Versata custom tags)
- ▶ Display and editing of the following data types: Text, Memo, Integer, Numeric, Currency, Boolean, Image, and Date/Time
- ▶ Display and editing of attributes with coded values list rules
- ▶ Display and editing of grids (grid code must be written within the JSP page)
- ▶ Inserts
- ▶ Updates
- ▶ Deletes
- ▶ Buffering of inserts, updates, and deletes

**Note:** This feature is implemented differently than in other Versata System clients, where by default deletes are not buffered.

- ▶ Parent-child transitions

### 9.3.2 Tag library overview

The tag library included with the Versata JSP Toolkit includes these tags:

- ▶ DataConnection tag to communicate with an underlying pool of VLS connections and pass authentication data across the connection.
- ▶ DataSet tag to provide access to business objects residing in the VLS. The tag iterates over the contents of a query object or data object and allows other HTML constructs to be contained within it, in order to provide access to individual fields.
- ▶ DataField tag that works in conjunction with the DataSet tag to support display and updates within the rows of a dataset.

## 9.4 Conclusion

The Versata Logic Suite offers a number of ways to integrate Versata business objects with any client technology which permits Java calls. Two variations used in this Redbook are calls to the Versata client library from the TradeAltAccess Java class and direct access from the Trade session EJB to the Versata-created EJBs using their remote interfaces.

In addition, Versata offers a JSP Toolkit to which supports access to business objects directly from Java Server Pages.

These connectivity options are available for applications which are not automated by the Versata Presentation Server.





## **Integrating Versata Logic Suite with WebSphere Studio Application Developer**

If you wish to enhance or debug the Java code generated by Versata for your application, it is helpful to use an Integrated Development Environment (IDE) that will facilitate this process. Perhaps the best IDE to use for this environment is IBM's WebSphere Studio Application Developer (WSAD).

In this chapter we show you how to import, run, and debug your Versata application code from WebSphere Studio Application Developer.

## 10.1 Introduction

WebSphere Studio Application Developer is IBM's latest and most advanced development environment for J2EE applications, and is the follow-on technology for WebSphere Studio and VisualAge for Java. The new tools provided in WebSphere Studio Application Developer support end-to-end development, testing, and deployment of e-business applications. They enable developers to create and test Servlets, JSPs, and Enterprise Java Beans.

In this chapter we discuss how the Versata Logic Suite can be integrated within WebSphere Studio Application Developer in order to take advantage of its code development and testing capabilities. This integration is also particularly useful when the Versata developed application is extended using WebSphere Studio Application Developer.

### 10.1.1 WebSphere Studio Application Developer

WebSphere Studio Application Developer is designed from the ground up to meet the requirements for all new types of applications. These requirements include open standards, Java, XML, Web services, testing, varying levels of integration with other components and ISV products, placability, expandability, role-based development, increased usability for all users, enhanced team support, as well as increased speed to market. WebSphere Studio provides integrated development tools for all e-business development roles, from Web developers to Java developers to business analysts to architects to enterprise programmers.

### 10.1.2 Integrated testing with WebSphere Application Server

WebSphere Studio Application Developer does not just contain a simulated test environment, but contains an actual single server version of WebSphere Application Server within the product. This enables the developer to build and test J2EE applications quickly using an environment that is very similar to the production WebSphere Application Server. This means that the developer can test Servlets, JSPs, and Enterprise Java Beans right within the development environment without having to deploy them to an outside server.

Then, using WebSphere Studio Application Developer's advanced debugging features, the developers can find and fix problems more quickly and accurately than they otherwise could. They can also unit test modifications to code without lengthy recompile and redeployment steps.

## 10.2 Versata Logic Server within WSAD

WebSphere Studio Application Developer version 4.0.2 and Versata Logic Suite version 5.5.1 were used for this exercise. Additional steps are needed for earlier versions of Versata. Since the process is greatly simplified with Versata 5.5.1, we recommend that this and later versions be used.

### 10.2.1 Preparing Versata application for import into WSAD

Before you are ready to import your application into WebSphere Studio Application Developer, you must properly prepare the application.

#### Preparing the EAR file with source

In order to run a Versata Application within WebSphere Studio Application Developer, you must create an EAR file with source code. For a J2EE application, the EAR file contains all items necessary for the deployment of the application, including EJB applications (JAR files), Web applications (WAR files) and deployment descriptors. When a Versata application is deployed normally in the Versata Logic Suite, the resulting EAR file does not contain source code. However, in order to use the debugger, within WebSphere Studio Application Developer, source code is required.

#### Copying the required files

To enable you to create an EAR file with source code, Versata has provided a process in the form of an executable batch file. This file, along with some other necessary files, is included in the resource materials for this document. Before performing this step, copy these files:

- ▶ wsEARCreate.bat
- ▶ wsEARTDS.lst
- ▶ wsEARImport.bat

You get these files from the directory where you installed the resource files to the directory: **<Versata Install Folder>/VLS/bin**

## Compiling the Versata application in debug mode

Since one of your reasons for importing your application into WebSphere Studio Application Developer may be to debug the code, it is important that the code be compiled in debug mode. To accomplish this, you must change one of the environment variables and then recompile the code within Versata Logic Suite. To do this, perform the following steps:

1. Open the file <Versata Install Folder>\setVersataEnv.bat in a text editor.
2. Find the entry:  

```
set JAVAC_OPTION=
```

and change it to:  

```
set JAVAC_OPTION=-g
```
3. Save and close the file
4. Open Versata Logic Studio and open the repository containing the application you wish to deploy to WebSphere Studio Application Developer.
5. Select Versata Logic Server->Deploy Transaction Logic from the menu.
6. Follow the prompts to deploy the transaction logic.
7. Highlight and then double-click to open the desired client application, and select managers->deployment manager from the menu to deploy the application.
8. Follow the prompts to deploy the client application.

## Creating the source EAR file

To create the EAR file containing source code, perform the following steps:

1. Open a Command prompt.
2. Navigate to the Versata Logic Server executable directory by keying:

```
cd <Versata Install folder>\vls\bin
```

where <Versata Install folder> is the name of the folder where you installed the Versata Logic Studio.

3. Key in the following:

```
wsEARCreate -repository <repository name> -copysource -repository_dir  
<full path to repository>
```

For example:

```
wsEARCreate ?repository SampDB1 ?copysource ?repository_dir  
d:\Versata\VLS?5.5?WebSphere\Samples\SampDB1
```

This step will create a file called <repository name>\_Deployed.ear (for example, SampDB1\_Deployed.ear) in <Versata Install Folder>\temp.



## 10.2.2 Importing applications into WSAD

Once you have prepared the EAR file that contains the source for your application, you are then ready to begin the process of importing the application into WebSphere Studio Application Developer.

### **Creating a classpath variable for the Versata install directory**

Since you will be making several references to the Versata Logic Suite install directory, it is a good idea to set up a classpath variable that refers to this directory. This will save time in defining class paths, and will make your application more portable. To set up a Classpath Variable for the Versata Logic Suite installation directory, do the following, as shown in Figure 10-1:

1. From the menu, select **Window->Preferences**.
2. In the hierarchy tree on the left, expand Java and select **Classpath Variables**.
3. Click the **New** button.
4. Enter VERSATA\_ROOT as the name and click the **Folder** button.
5. Navigate to the folder where you installed Versata Logic Suite (Example: D:\Versata\VLS-5.5-WebSphere), and click **OK** from the New Variable Entry panel.
6. You should now see VERSATA\_ROOT as one of the variables listed. Click **OK** from the Preferences panel.

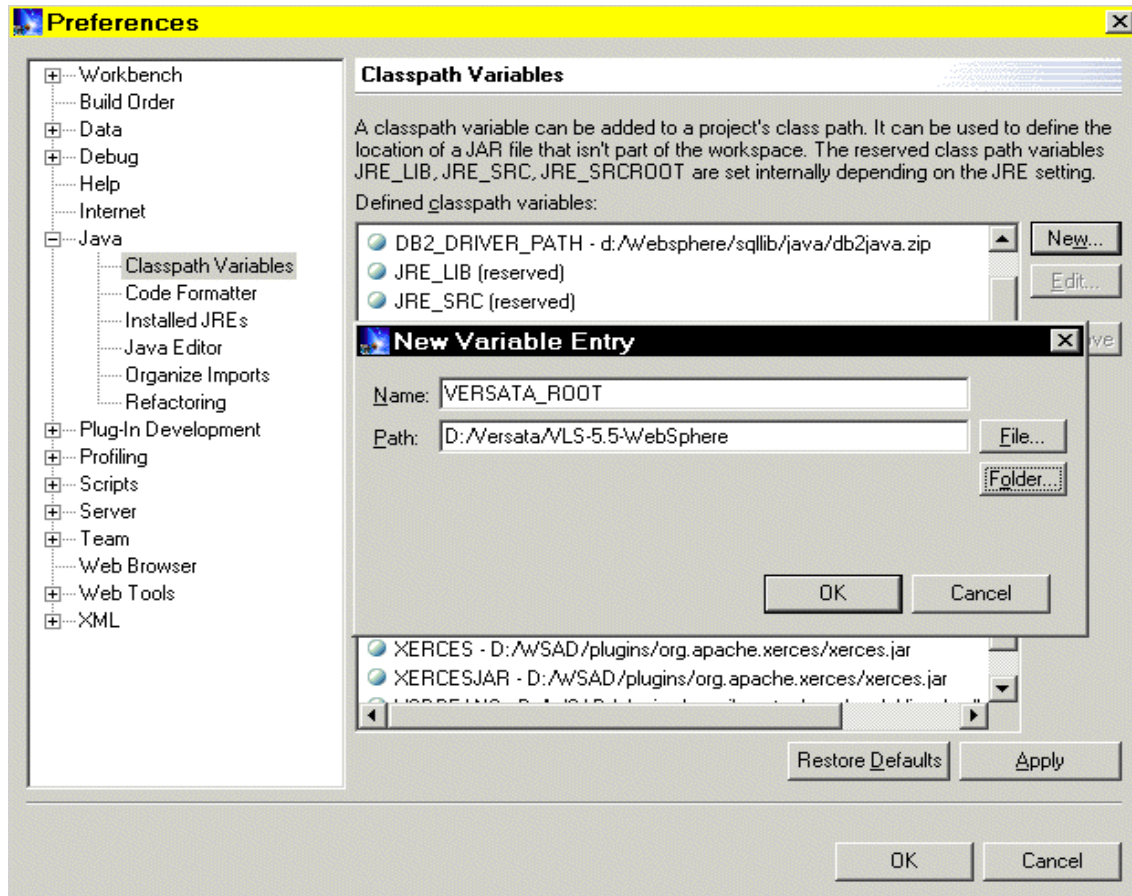


Figure 10-1 Classpath variable setup

## Importing the Versata Logic Server code

In order to run your application, the Versata Logic Server must be running. Since the Versata Logic Server will be running within WebSphere Studio Application Developer, you must import the files needed by the server. This code is found in <Versata Install Folder>\VLS\lib\vlsBeans55.ear.

1. Open WebSphere Studio Application Developer from the **Start** button.
2. From the menu, select **File->Import**.
3. Select EJB JAR File from the list of options and click the **Next** button as shown in Figure 10-2.

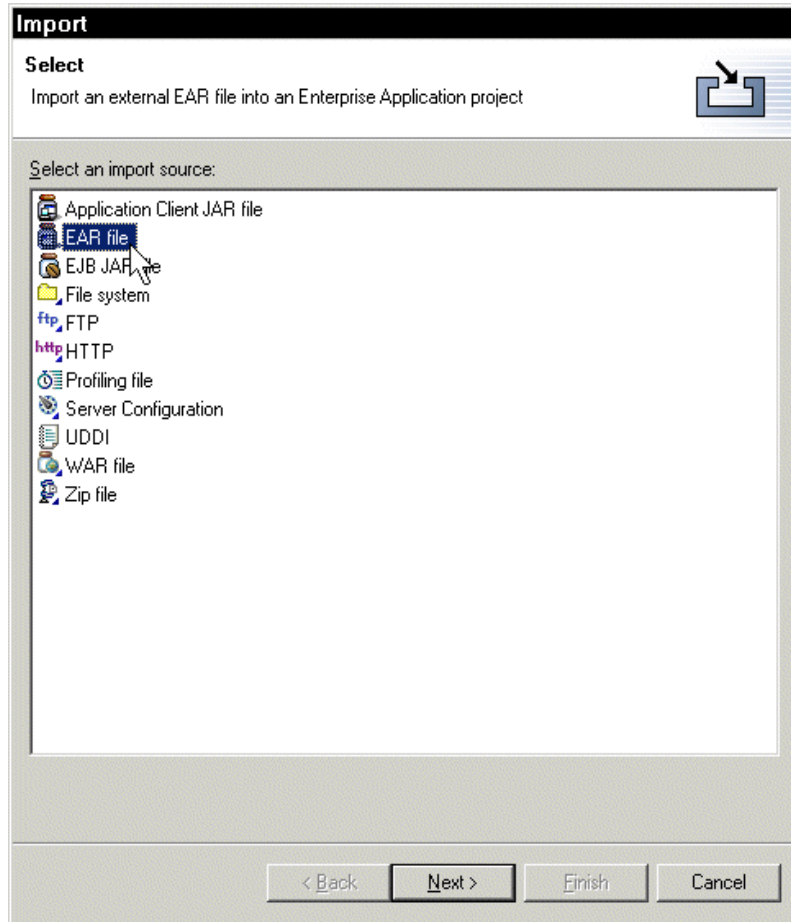


Figure 10-2 Importing EAR jar file

4. Click the **Browse** button by the EAR File text box.
5. Navigate to <Versata Install Folder>\VLS\lib\vlsBeans55.ear and double-click it, or select open. This will return you to the previous panel and place the full path to the vlsBeans55.ear file in the EAR File text box.
6. In the EJB Project box, enter vlsBeans55\_EAR (The name does not matter. Just select a meaningful one) as shown in Figure 10-3.

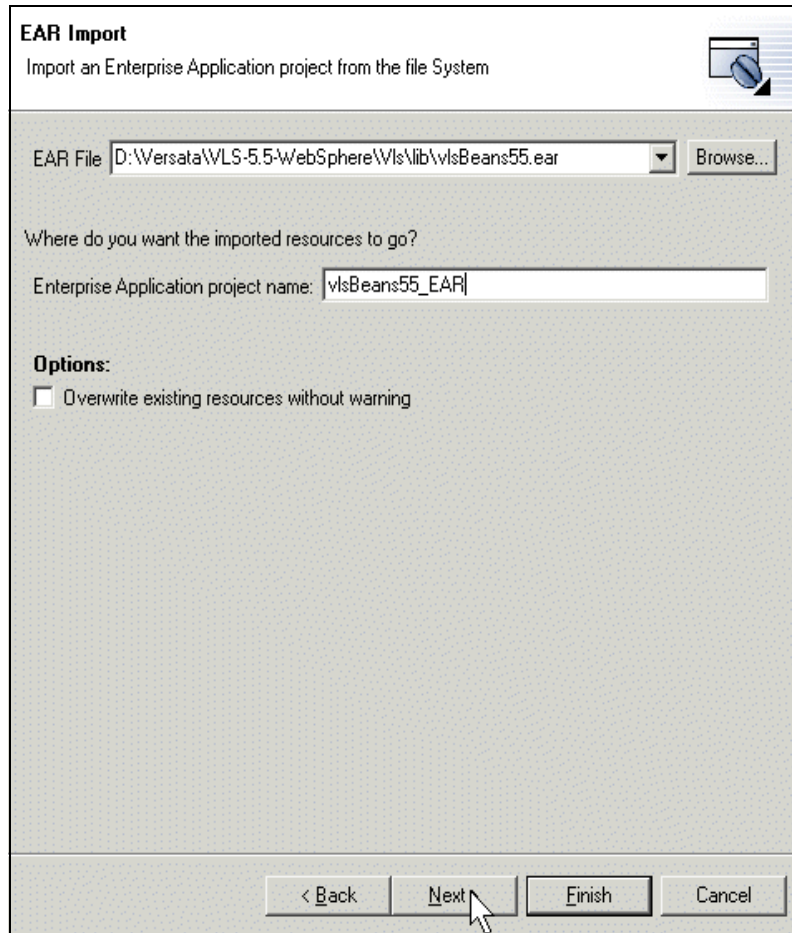


Figure 10-3 Project name input

7. Click the **Next** button from the EAR Import panel.
8. Click the **Next** button from the Manifest Class-Path panel.
9. On the EAR Modules panel, select the vlBeans55.jar module. Enter vlBeans55\_EJB as the New Project Name as shown in Figure 10-4.

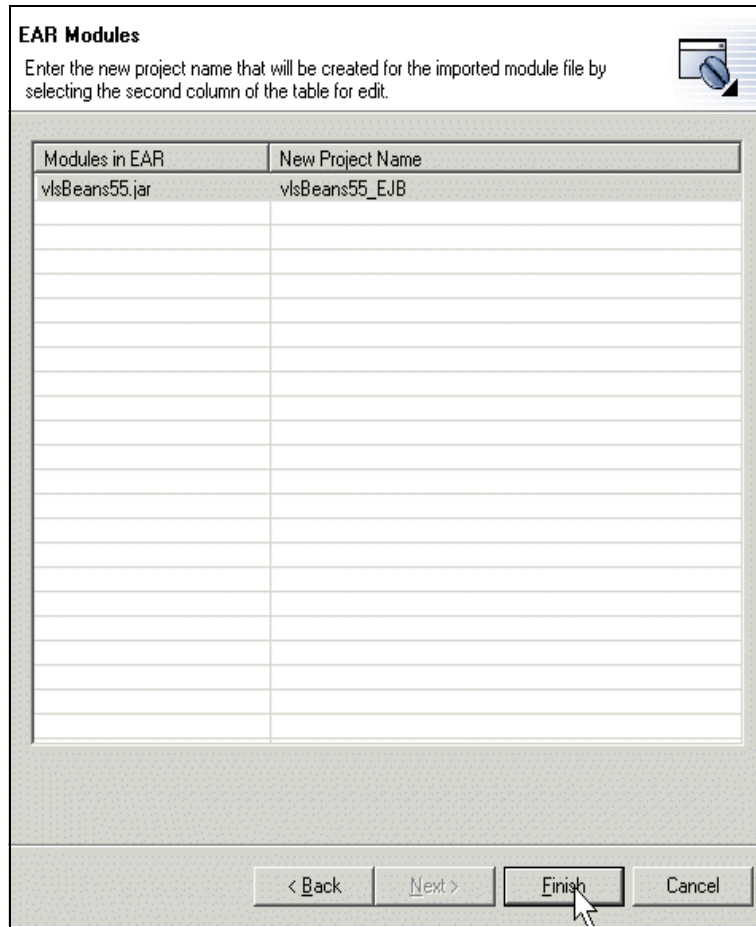


Figure 10-4 New project name setting

10. Click the **Finish** button. This step will create the EJB and EAR projects, import the code, and attempt to compile it. You will see several errors in the Tasks view, but you should ignore them for the moment.

### Setting the build properties for the vlsBeans55\_EJB project

In order to eliminate the compile errors generated by the previous step, it is necessary to update the java build path of the vlsBeans55\_EJB project. To do so, perform the following steps, as shown in Figure 10-5:

1. Make sure you are in the J2EE perspective of WebSphere Studio Application Developer. To open this, select perspective->open->J2EE from the menu.

2. Make sure you are in the navigator view by selecting the navigator tab on the upper left window of the J2EE perspective.
3. Right-click the vlsBeans55\_EJB project and select properties from the context menu.
4. From the properties panel, select java build path.
5. Select the libraries tab and click the **Add Variable** button.
6. Enter VERSATA\_ROOT as the Variable Name, and Vls/lib/vlsEJB55\_g.jar as the Path Extension. If you like, you can use the Browse buttons to retrieve these values.
7. Click **OK** from the Classpath Variable Selection panel.
8. Repeat steps 5 through 7 above, giving VERSATA\_ROOT as the Variable Name, and Vls/lib/vlsEJB55.jar as the Path Extension.
9. Click **OK** from the Properties panel, thus causing the project to be rebuilt.

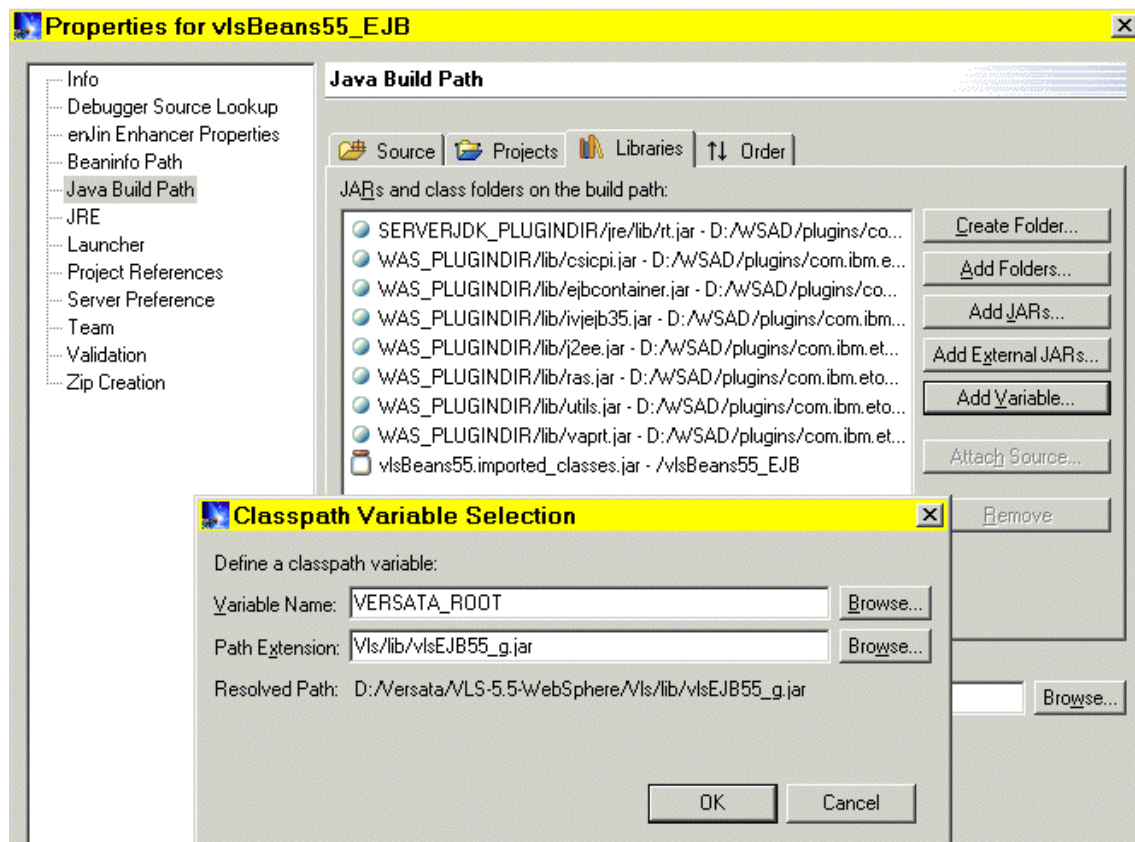


Figure 10-5 Properties for vlsBeans55\_EJB

At this point, you should see no more fatal errors in the tasks view. You will see some warning errors, but you can ignore those.

## Importing your Versata application EAR file into WSAD

You are now ready to import the EAR file for your application that you created in an earlier step. To do this, perform the following steps as shown in Figure 10-6:

1. From the WebSphere Studio Application Developer menu, select File->Import.
2. Select EAR File from the list of options and click the **Next** button.
3. Click the **Browse** button next to the EAR file text box. Then, navigate to <Versata Install Folder>\temp\<repository name>\_Deployed.ear (e.g. SampDB1\_Deployed.ear) and double-click it, or select open. This will return to the previous panel and place the full path to the file in the Ear File text box.
4. In the Enterprise Application project name box, enter <repository name>\_EAR. The name doesn't matter. Just select something meaningful. Then click the **Next** button.

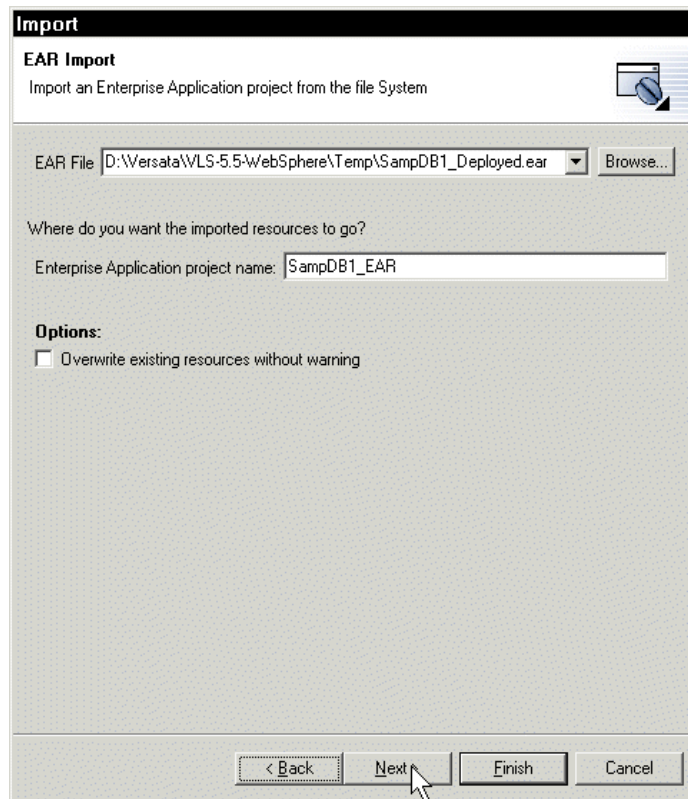


Figure 10-6 EAR import

5. On the manifest class path panel as shown in Figure 10-7, select the <repository name>.war file. Then make sure the check box to the right of <repository name>.jar is checked and click next. This indicates a dependent relationship between the Web project and the EJB project.

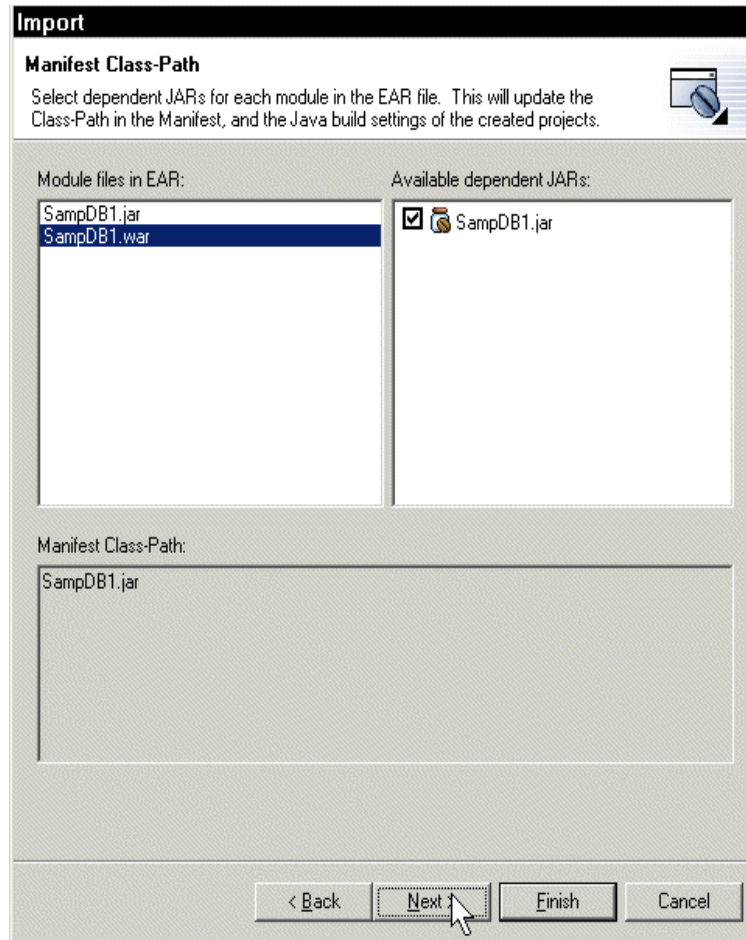


Figure 10-7 Manifest class path

6. On the EAR modules panel in Figure 10-8, select the <repository name>.jar module and enter <repository name>\_EJB as the new project name. Then select the <repository name>.war file and enter <repository name>\_WEB as the new project name.





## Setting the build properties for the application EJB project

Your Versata EJB application refers to classes that are in the some Versata JAR files. In order to make your application compile without errors within WebSphere Studio Application Developer, you must set the class path to point to these external JAR files.

1. Make sure you are in the J2EE perspective of WebSphere Studio Application Developer. To open this perspective, select perspective->open->J2EE from the menu.
2. Make sure you are in the navigator view by selecting the navigator tab on the upper left pane of the J2EE perspective.
3. Right-click the <repository name>\_EJB project and select properties from the context menu.
4. From the properties panel, select java build path.
5. Select the libraries tab and click the **Add Variable** button.
6. Enter VERSATA\_ROOT as the Variable Name, and Client/lib/vfcEJB55\_g.jar as the Path Extension. If you like, you can use the Browse buttons to retrieve these values.
7. Click **OK** from the Classpath Variable Selection panel.
8. Repeat this process three more times, each time giving VERSATA\_ROOT as the variable name, and the following as the Path Extension:
  - a. Client/lib/vfcEJB55.jar
  - b. Vls/lib/vcsEJB55\_g.jar
  - c. Vls/lib/vcsEJB55.jar
9. Click **OK**. This action will cause the project to be rebuilt.

The number of compile errors in the Tasks panel will decrease, but you will still see a number of them. You can continue to ignore them at this point.

## Setting the build properties for the application Web project

As with the EJB project, your Versata Web application refers to classes that are in the some Versata JAR files. In order to make your application compile without errors within WebSphere Studio Application Developer, you must set the class path to point to these external JAR files.

1. Make sure you are in the J2EE perspective of WebSphere Studio Application Developer. To open this perspective, select perspective->open->J2EE from the menu.
2. Make sure you are in the navigator view by selecting the navigator tab on the upper left window of the J2EE perspective.
3. Right-click the <repository name>\_WEB project and select properties from the context menu.
4. From the properties panel, select java build path.
5. Select the libraries tab and click the add Variable button.
6. Enter VERSATA\_ROOT as the Variable Name, and Client/lib/vfcEJB55\_g.jar as the Path Extension. If you like, you can use the Browse buttons to retrieve these values.
7. Click **OK** from the Classpath Variable Selection panel.
8. Repeat this process three more times, each time giving VERSATA\_ROOT as the variable name, and the following as the Path Extension:
  - a. Client/lib/vfcEJB55.jar
  - b. Vls/lib/vcsEJB55\_g.jar
  - c. Vls/lib/vcsEJB55.jar
9. Select the Projects tab, and make sure that both the vlsBeans55\_EJB project and the <repository name>\_EJB project are checked.
10. Click **OK**. This action will cause the project to be rebuilt.

Just to make sure all the projects are rebuilt using the settings you have entered, select Project->Rebuild All from the menu. This will recompile all the projects in the workspace.

At this point, you will see some warning errors and possibly some fatal errors in the tasks view. The only fatal errors you should see will be HTML errors such as missing start tags. You can ignore these as well as the warning errors. If you still see fatal errors other than the HTML related ones, go back and review the previous steps and make sure you have performed them all correctly.

### 10.2.3 Configuring the server to test the application

You are now ready to set up the server that will allow you to test your application. WebSphere Studio Application Developer contains a single server version of WebSphere Application Server Version 4.x that you can configure and run from within Application Developer. The steps outlined below will show you how to configure a server that will run your Versata application.

#### **Creating a server instance and configuration**

The first step in setting up a server is to create an instance and configuration. You do this by performing the following steps as shown in Figure 10-9:

1. Open the Server perspective by selecting Perspectives->open->other->Server from the Application Developer menu.
2. From the menu, select File->new->Other->Server->Server Instance and Configuration and click Next.
3. Enter VLS\_Server as both the server name and folder (again, any name will do). Under Server instance type, expand WebSphere Servers and select WebSphere v4.0 Test Environment. Leave the template box with the default value of none and click the Finish button. When a dialog box asks if you want to create a server project named VLS\_Server, click Yes. This action will create a new server project complete with a server instance and a server configuration.

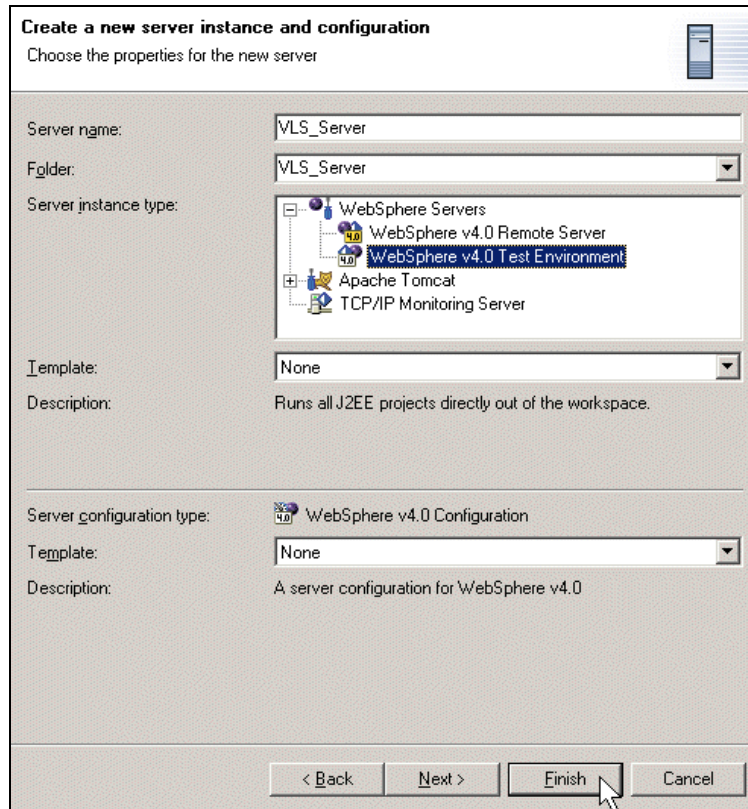


Figure 10-9 New server instance creation

## Setting the module visibility

The module visibility of the server must be set to Server. To do this:

1. In the Server Configuration view, expand Server Configurations if necessary and double-click VLS\_Server as shown in Figure 10-10.
2. Select the General tab.
3. From the drop-down box labeled Module Visibility, select Server.
4. Save the changes and close the editor.

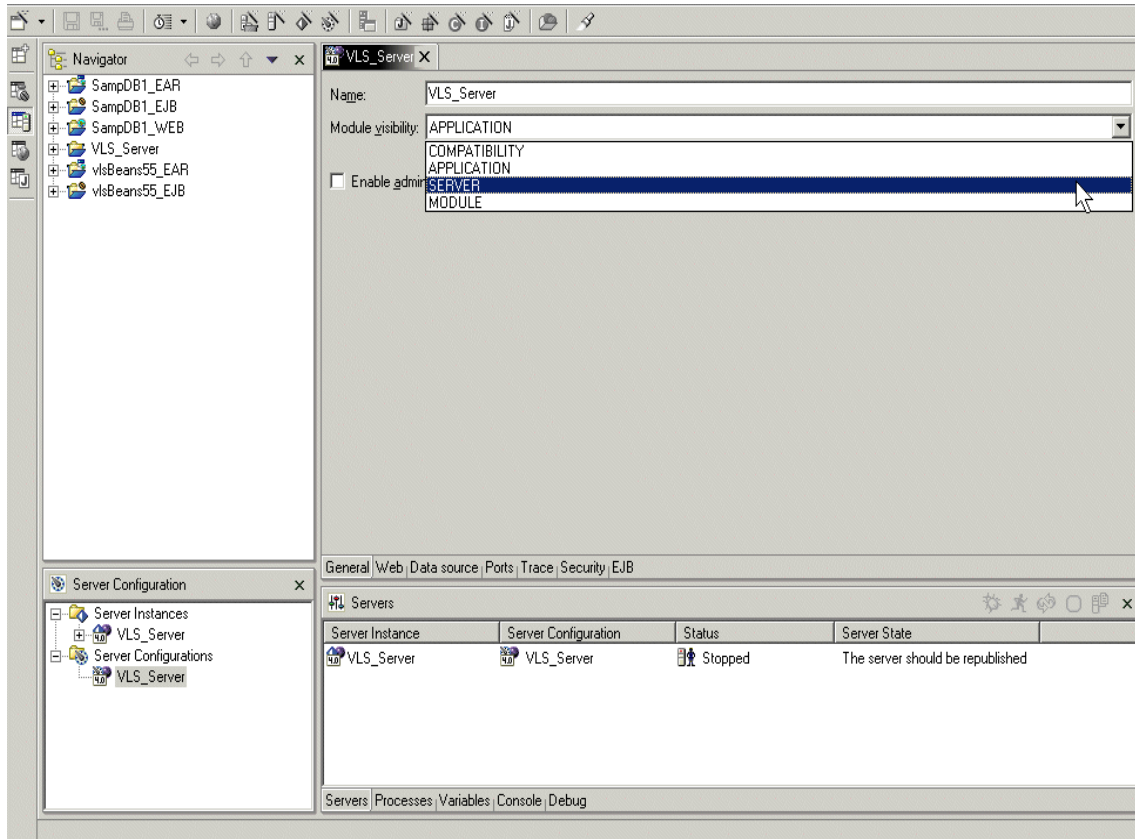


Figure 10-10 Setting module visibility

## Attaching the enterprise application to the server

Now that you have a Server set up, you must tell the server which application it should run. You do this by performing the following steps as shown in Figure 10-11:

1. In the Server Configuration view, expand Server Configurations if necessary.
2. When you right-click on VLS\_Server and select Add Project, you will be presented with a drop-down box listing the two enterprise application projects you created earlier. If you followed the suggested naming conventions, they will be named VLSBeans55\_EAR and <Repository Name>\_EAR. Go ahead and add both of these projects to the server configuration by right-clicking VLS\_Server and selecting Add Project-><Project name> from the context menu.

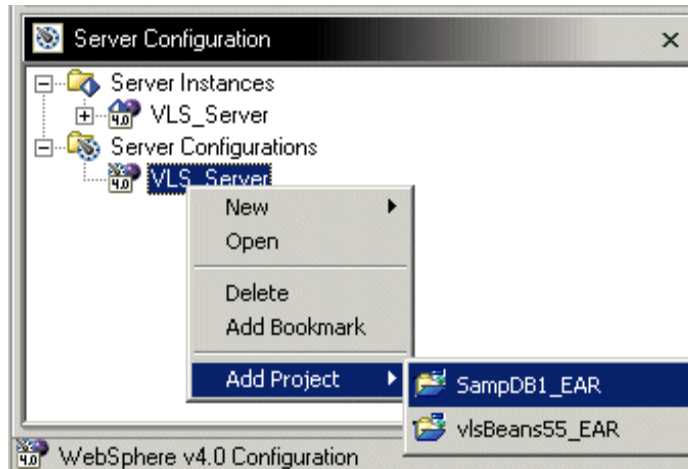


Figure 10-11 Attaching to server configuration

## Configuring the server instance

The server instance you have just created must have access to all the classes necessary to run your Versata application. This means that you must set up the classpath for the server.

### ***Configuring server the server instance classpath by copying the VLS\_Server.wsi file***

When you configure the classpath manually as outlined below, the editor updates an XML file called `VLS_Server.wsi`. A sample of this file is included with the resource materials for this document. As an alternative to keying in all the entries, you could simply import this file into the `VLS_Server` project, replacing the existing one. You could then modify the file as needed. This sample file is for deployment of the `SampDB1` repository. If you are deploying a different repository, you will want to replace all references to `SampDB1` with the name of your project. This file also assumes that you have created a classpath variable named `VERSATA_ROOT` pointing to your Versata Install Directory.

### ***Configuring the server instance classpath manually***

To manually configure the server instance classpath, complete the following steps:

1. From the Server Configuration view, expand Server Instances and double-click `VLS_Server`. This will open an editor for the `VLS_Server` instance.
2. Select the Paths tab. This panel contains two text areas: One labeled WebSphere specific class path and the other labeled Class Path. In the steps

below, we will be working with the second box labeled Class Path and the buttons to the right of it.

3. Using the Add Variable button, add the following entries:
  - a. Variable name: VERSATA\_ROOT, Path Extension: /Client/lib
  - b. Variable name: VERSATA\_ROOT, Path Extension: /Vls/lib
4. Using the Add Folder button, add the following entries:
  - a. <Repository Name>\_EJB\bin
  - b. <Repository Name>\_WAR\webapplication\WEB-INF\classes
5. Using the Add Variable button, add the following entries:
  - a. Variable name: VERSATA\_ROOT, Path Extension: Client/lib/vfcEJB55\_g.jar
  - b. Variable name: VERSATA\_ROOT, Path Extension: Client/lib/vfcEJB55.jar
  - c. Variable name: VERSATA\_ROOT, Path Extension: Vls\lib\vlsEJB55\_g.jar
  - d. Variable name: VERSATA\_ROOT, Path Extension: Vls\lib\vlsEJB55.jar
  - e. Variable name: VERSATA\_ROOT, Path Extension: Vls\lib\vlsBeans55\_g.jar
6. Using the Add Variable button, add the DB2 driver path. To do this:
  - a. Click the Add Variable button.
  - b. From the Variable Name drop-down box, select DB2\_DRIVER\_PATH.
  - c. Leave the Path Extension box empty.
  - d. Click OK
7. The next step is to add the JAR files in the <WASD Install Folder>\Applications Developer\plugins\com.ibm.etools.websphere.runtime\lib directory. To allow for any application that could possibly be run in this environment, you should add all of them, but the list below gives the most essential ones that will be sufficient for most applications. The classpath variable WAS\_PLUGINDIR points to the WAS runtime directory within Application Developer, and is created automatically for you. To add these JAR files, click the Add Variable button, selecting WAS\_PLUGINDIR as the variable name and the following as extensions:
  - a. lib/csicpi.jar
  - b. lib/ejbcontainer.jar
  - c. lib/http.jar
  - d. lib/httpsession.jar



- e. lib/ivjejb35.jar
  - f. lib/iwsorb.jar
  - g. lib/iwstools.jar
  - h. lib/j2ee.jar
  - i. lib/nssrcm.jar
  - j. lib/ras.jar
  - k. lib/websphere.jar
  - l. lib/webcontainer.jar
  - m. lib/xml4j.jar
8. Add the tools.jar file from the plugin directory com.ibm.etools.server.jdklib. To do this, click the **Add Variable** button, selecting SERVERJDK\_PLUGINDIR as the Variable Name and lib/tools.jar as the Path Extension.
  9. Save your work by clicking **Ctrl-s** and close the editor window.

## 10.2.4 Running and debugging the application

Once the application has been installed, and the server configured, you are ready to actually run and debug your application from within WebSphere Studio Application Developer.

### Starting the server

The first step in running your application is to start the server. To do this, perform the following steps:

1. From the Server perspective, open the Servers view. By default, this view is located on the lower left panel. You will have to find its tab to select it.
2. Right-click VLS\_Server and select Start from the context menu. Note: selecting start will allow you to run the application with debug disabled. This is faster, and is recommended for simply running the application, but if you want to set breakpoints and debug the application, you should select debug instead of start.
3. Watch the Console view for the message "... server open for e-business".

The server is now running both the Versata Logic Server and your client application.

## Running your HTML application

Now that the server is running, you are ready to run your client HTML application from within WebSphere Studio Application Developer. To do this, perform the following steps:

1. Open the J2EE perspective by selecting Perspective->Open->J2EE from the menu.
2. Open the J2EE view by clicking the J2EE tab.
3. Expand Web Modules and <Repository Name>.war.
4. For each HTML client application you have deployed, you will see a servlet icon that looks like this:



Figure 10-12 Servlet icon

5. Right-click the icon corresponding to the application you wish to run, and select Run from Server from the context menu as shown in Figure 10-13. This will open up a Web browser window within WebSphere Studio Application Developer that you can use to navigate through the pages of your application.

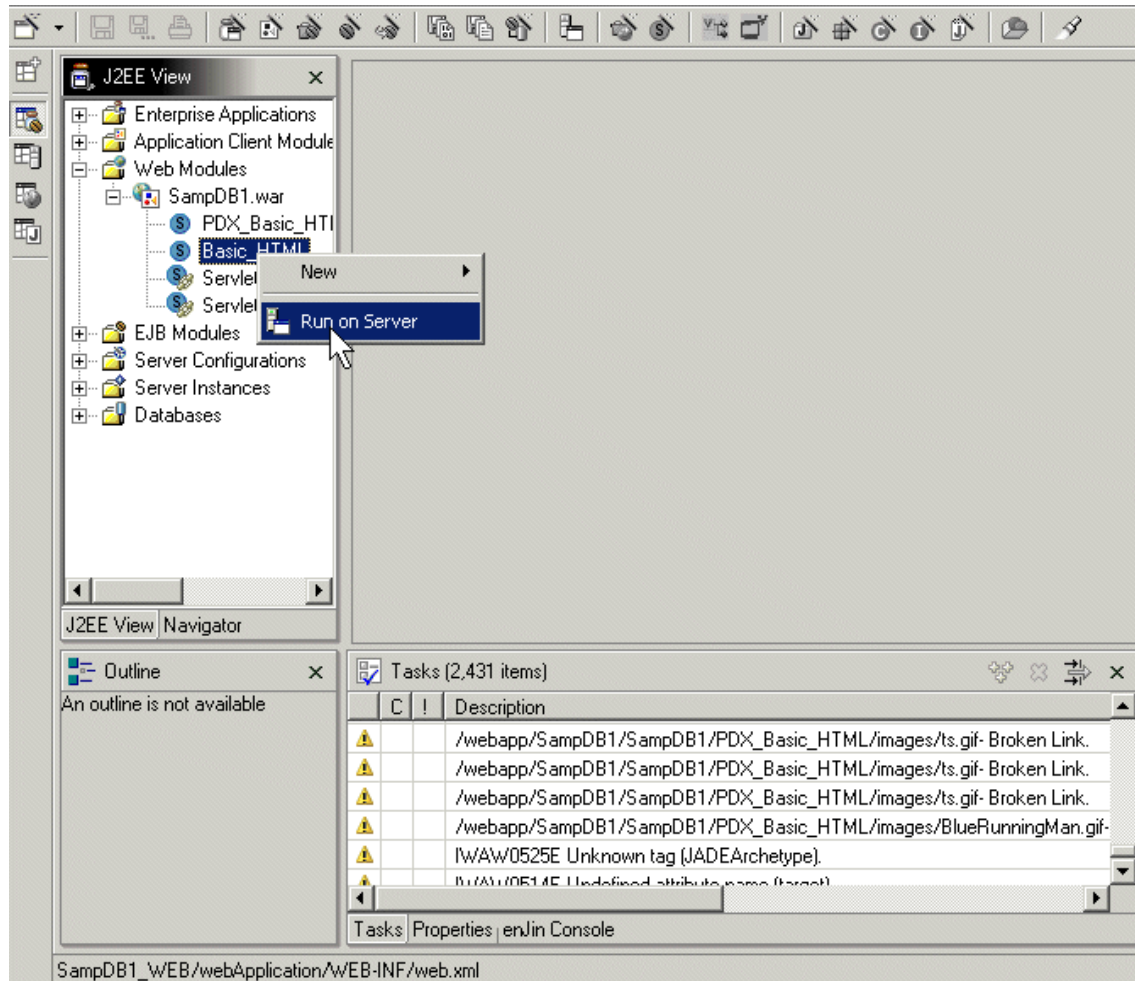


Figure 10-13 Run on Server

**Important:** Because of the way that Versata handles Web applications, you will not be able to see images displayed on your HTML pages when running them in this environment. You may also frequently see the message Virtual Host or Web Application Not Found. However, these problems are only cosmetic in nature and should not prevent you from following the links on the page and exercising the application.

## Debugging your application

Probably the main reason for importing a Versata Application into WebSphere Studio Application Developer is to allow you to use its advanced debugging features to examine the code while it is running. To debug an application:

1. Find the source file of the class you want to debug. Source files for EJB projects are found in the `ejbModule` folder in a package named after the repository. Source files for Web projects are found in the `Source` folder in a package named `<Repository Name>.<Application Name>`.
2. Open the source file in the java editor by double-clicking it from the Navigator view.
3. Find the place in the code that you would like to debug, and set a breakpoint at that point. To do this, you simply double-click in the left margin on the line of code where you want the breakpoint. A colored dot will appear in the margin area indicating that a breakpoint has been set.
4. Start the server in debug mode. To do this, follow the instructions on starting the server given above, but specify `debug` instead of `start`.
5. Run the application as you normally would. When the code containing the breakpoint is executed, the Debug View will be displayed in the Server perspective. From this view, you can step through the code. To fully utilize the debugging features, however, you should open up the Debug perspective. For more information on how to use the debugger, consult the WebSphere Studio Application Developer online documentation.

## 10.3 Importing modified application into Versata

If you modify the application code within WebSphere Studio Application Developer, you are obviously going to want to incorporate those changes into your application when it is run from the normal process. This section explains how to bring the modified application back into Versata.

### 10.3.1 Exporting the application from WSAD

To export the application from WebSphere Studio Application Developer:

1. Select File-Export from the menu.
2. Select EAR File as the type of export.
3. Specify the EAR project you wish to export (e.g. SampDb1\_EAR) and the destination path and file name (e.g. d:\SampDB1\_Modified.ear)
4. Make sure the Export Source Files check box is selected.
5. Click **Finish**.

### 10.3.2 Import the application into the repository

The file wsEARImport.bat is provided to allow you to import the file that you exported from WebSphere Studio Application Developer back into the Versata repository. Here are the steps to use it:

1. Open a command window.
2. Navigate to the Versata Logic Server executable directory by keying: `cd <Versata Install folder>\vls\bin` where <Versata Install folder> is the name of the folder where you installed the Versata Logic Studio.
3. Key the following: `wsEARImport -earfile <input ear file> -repository_dir <repository directory>`
  - a. Example: `wsEarImport -earfile d:\SampDB1_Modified.ear -repository_dir d:\Versata\VLS-5.5-WebSphere\Samples\SampDB1`

**Note:** The wsEARImport utility will only import .java files to Versata's repository.

You can deploy the EAR file that you export from WebSphere Studio Application Developer directly to WebSphere Application Server by including the `appdeploy.properties` and the `vlsdeploy.properties` files in the EAR file.





## Developing with UML and rules

The question that usually follows a serious evaluation of Versata is “How can rule-based automation be used by my existing development teams?” This question reflects the fact that many organizations already employ a defined development methodology and any new programming paradigm must fit into that methodology.

Although business rules can be added to any development scenario, Versata technology has most frequently been integrated into the Rational Unified Process (RUP), provided by Rational ® Software.

## 11.1 UML and the Rational Unified Process

The Rational Unified Process (RUP) is a comprehensive software engineering approach that defines activities and roles throughout the development life-cycle. It uses the industry standard Universal Modeling Language (UML) to communicate and document software requirements, architectures and designs.

A variety of tools, from Rational as well as from other vendors, can be used to model, create, manage and test software using the RUP methodology. To support the Rational Rose design environment, Versata provides “adapter” technology which coordinates Rose models with the business logic created in the Versata Studio.

To support the development process, independent of any tool, Versata provides a training class, called “UML to Rules”, that shows how existing UML “artifacts” (designs and other work products) can be extended to cover business rules. The following sections discuss some possible approaches for developing in Versata when using RUP.

## 11.2 RUP phases and iterations

One of the key goals of the Rational Unified Process is iterative software development, with each system iteration providing a more comprehensive solution to the business problem

Within each development cycle, there are four defined phases:

- ▶ Inception Phase — where core requirements, “actors”, and initial use cases are defined
- ▶ Elaboration Phase — where detailed analysis is completed and a working prototype is produced
- ▶ Construction Phase — where remaining components are developed and integrated
- ▶ Transition Phase — where the software is delivered to end users for validation and identification of further requirements

### 11.2.1 Versata and the Inception phase

The purpose of the inception phase is to define the scope of the project. Versata recommends starting from a Business Perspective, beginning with Business Processes and Business Policies.



## 1. Begin with a process model

Versata explains that a logical starting point in project definition is a Process Model. Process modeling can uncover (without detail) key business functions and determine which are candidates for rule-based automation.

Figure 11-1 illustrates processing a customer order with Business Actors who benefit from process execution and Business Workers who carry out the processes. Key business processes may be further detailed with Activity Diagrams, textual Use Cases and State Diagrams.

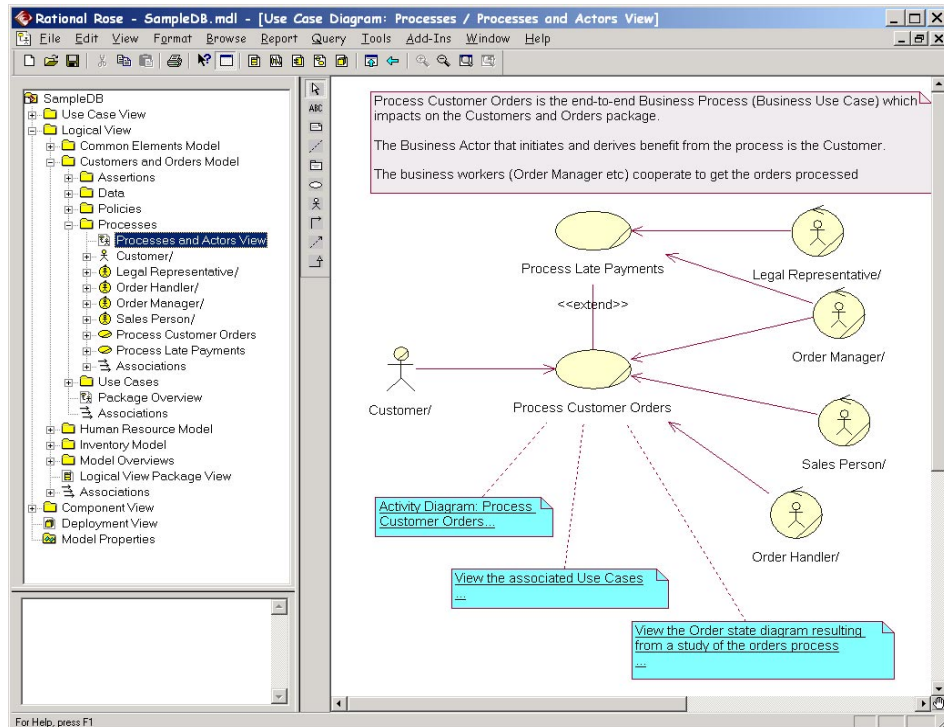


Figure 11-1 Model for processing a customer order

## 2. Add activity diagrams

For key processes, an Activity Diagram will summarize the workflow between Actors. There are two uses of Activity Diagrams. First, they are the basis for identifying Use Cases, the key unit for project estimation, development, and testing. Second, activity information can be utilized with process automation technology such as Versata's Process Logic Designer and Process Logic Engine, which are add-on's to the Versata Logic Server.

### **3. State a high-level policy model**

A Versata-specific step in the Inception Phase is the development of a “Policy Model”.

Policies, or high-level system goals, are added to the design of Versata automated logic because of the declarative nature of rules-based programming. As we see when implementing the Trade rule for margin selling (described as “The account balance can't go below the limit set by the SEC”), it is important to capture these high-level statements in the system design.

When training users on the “UML to Rules” approach, Versata explains how to use UML to state system Policies independent of process models. This is important because, as you will recall, rules are enforced independent of any specific transaction. The margin selling constraint is automatically checked when selling stock, when transferring funds, or when performing any other transaction that touches the balance attribute.

In the Elaboration Phase, Policies will be linked to specific “Business Assertions” (rule-enforceable logic), which will be further elaborated into derivations, constraints and other specific rule declarations. The design process drills down from Policies to Business Assertions to Rule as shown in Figure 11-2.

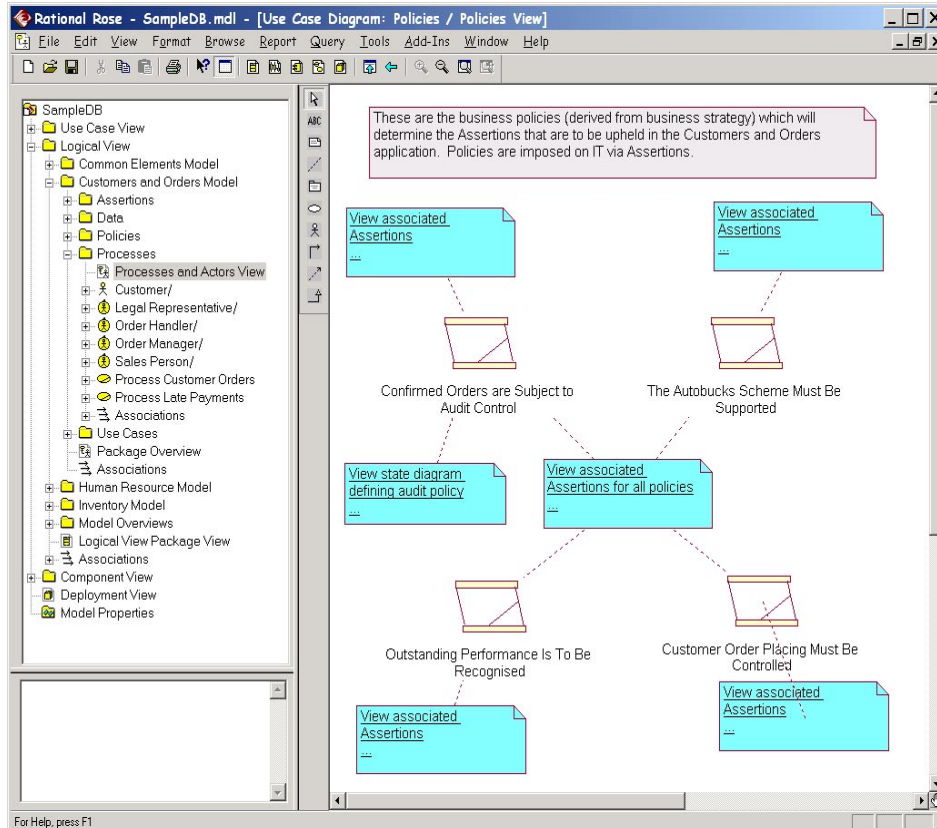


Figure 11-2 System policy model the Versata SampleDB example

#### 4. Develop conceptual class diagrams

With high-level policies stated, initial class diagrams may be started. In the Inception Phase, Class Diagrams represent a simple Conceptual Model of the object model.

Since class diagrams will be completed during the Elaboration Phase, for Versata purposes the Conceptual Model can simply sketch object relationships, eliminating Foreign Keys and other details. Since the conceptual objective is to simply make system elements clear to Business Users, performance-oriented oriented designs such de-normalizations are not needed at this time.

#### 5. Identify use cases and business assertions

One output of the Inception Phase is the initial project scope. This can be determined by identifying Use Cases and key Use Case Assertions.

## Use cases

A use case is a sequence of actions a system performs that yields an observable result of value to a particular actor. The result can be as simple as obtaining the current balance of the actor's account, or it can be very complex, for instance buying a holding of stocks for the actor. Use cases are described in text, and modeled in Use Case Diagrams showing actors, systems, system boundaries, use cases, and the relationships between these elements.

Figure 11-3 shows the “Create New Order” use case for the Versata SampleDB, which is shipped with the Versata Studio.

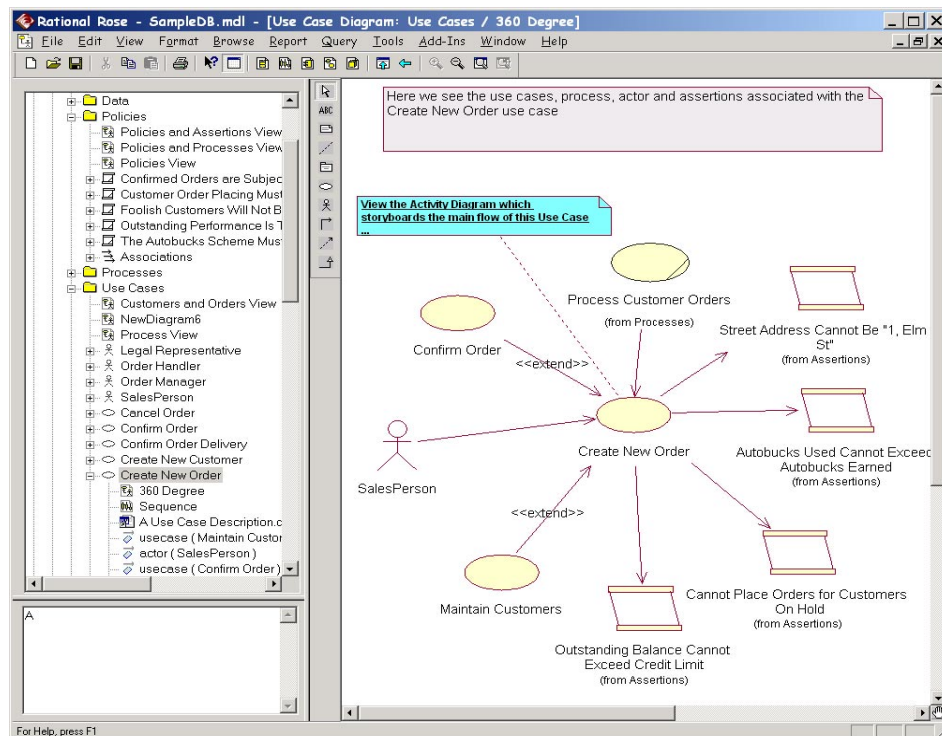


Figure 11-3 Rational Rose use case for a Versata-automated functions

## Business assertions

Versata suggests that Use Cases be identified and briefly summarized (but not elaborated) in the Inception Phase. Versata ties another rule-based UML extension, called a Business Assertion, to use cases.

Business Assertions are the statements of what must occur during the execution of a Use Case. In some cases, an assertion may translate directly into single rule such as a constraint that must be checked or a derivation that must be performed. One such assertion in the extended TRADE application is “Accounts have the following Types: New, Retail, and Wholesale”.

In other cases, an assertion may require more than one rule be created. A TRADE example is the rule that says “If the PortfolioValue + the Balance < 100000.00 AND the number of transactions is more than five, then the account type is Retail”. Since we didn't yet have an attribute for PortfolioValue, this assertion required two rules to implement. One rule derived the PortfolioValue and one rule derived the AccountType.

Business Assertions form the basis for rule design and testing. Designing multiple rules to solve a Business Assertion is the key skill in using Versata rules technology.

Within a UML modeling tool such as Rational Rose, the specific rule types (derivations, constraints, etc.) are stereotyped from UML operations (methods.) This allows a Business Assertion to be detailed into the multiple rules that may implement it.

## **6. Determine scope**

The models described above are used to determine the initial project scope and deliver an initial plan and estimate. Versata provides recommendations on the how this can be expanded into a detailed scope with key Use Cases, Assertions, and Objects identified.

Ranking the various artifacts allows key models to be expanded in the Elaboration Phase.

### **11.2.2 Versata and the Elaboration phase**

Rational describes the Elaboration Phase as the most critical of the four phases. Elaboration de-risks the project by uncovering sufficient details to prove the architecture, stabilize the data model, and produce a complete, working prototype.

Because rules-based automation converts detailed specifications into directly executable components, success in the Elaboration phase is virtually ensures a successful Versata-based project.

## **1. Elaborate key use cases**

Elaboration begins by identifying key Use Cases through a scoring process. For the key Use Cases the first step is to populate and normalize the data model. Because the data model is a fundamental to Versata relationships and other rules, particular attention should be given to data types and sub-types, referential integrity requirements and especially data state changes.

## **2. Analyze interesting state changes**

As we have learned, transactional rules are “data-change oriented”. Therefore Versata recommends modeling key business entities with a State Diagram. State diagrams uncover dependencies between the attribute values in multiple entities.

For instance, the State Analysis, in Figure 11-4, details the Order entity and potential order states from the Versata SampleDB repository. Here the creation of a new changed Order (not “Collect on Delivery”) is dependent on the Customer.OnHold flag being set to false. This suggests a potential constraint on the Customer entity that would prevent any that would create the number of unpaid transactions to increase if the Customer was on hold.

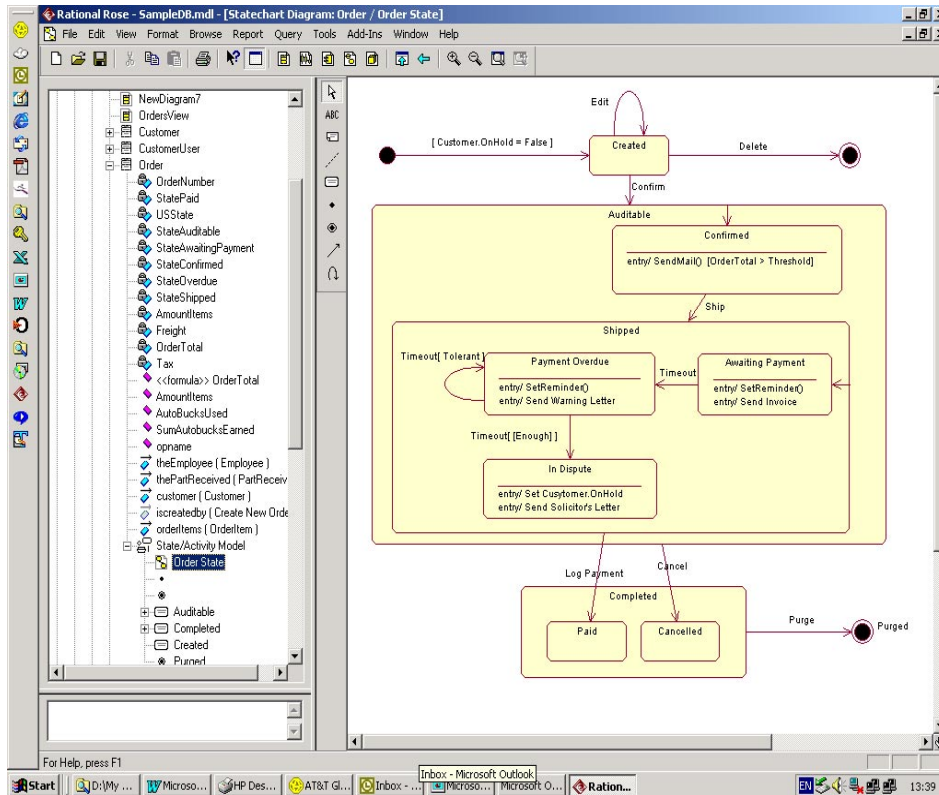


Figure 11-4 State diagram for an order entity

### 3. Utilize sequence diagrams

To complete the design process for this constraint, the modeler would define a Business Assertion “preventOrderOnHold”. This Business Assertion is reflected in step 10 and 11 of the Use Case Sequence Diagram as shown in Figure 11-5. These map the Business Assertion directly to the two Versata rules that enforce it: one to determine the number of unpaid orders, and the second to enforce the constraint.

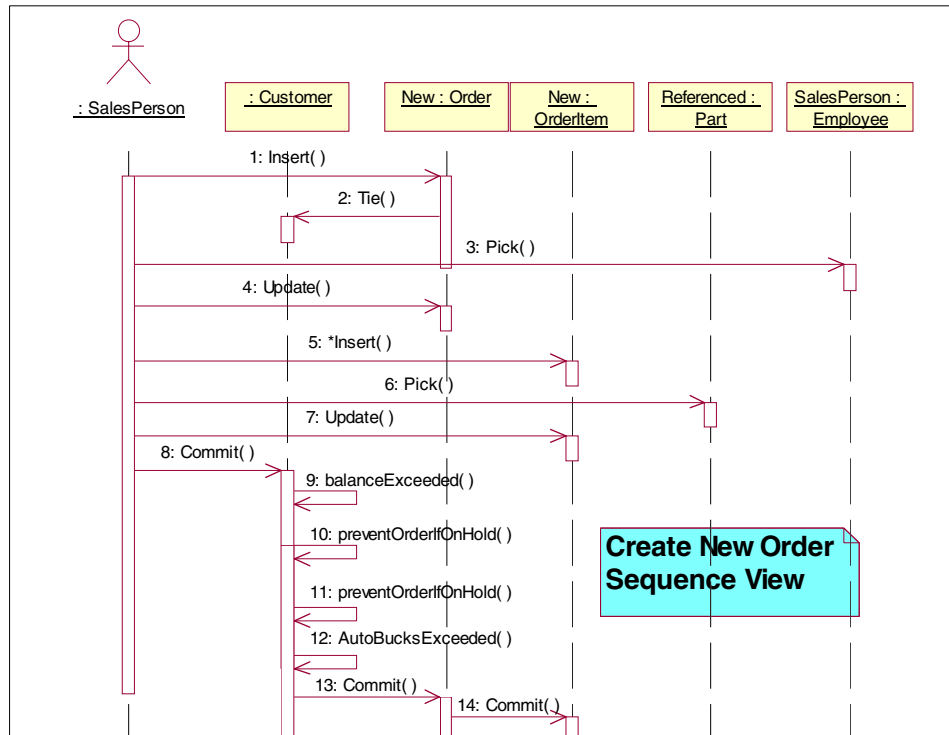


Figure 11-5 Steps 10 and 11 map the business assertion to two rules

With the two rules identified to satisfy the Business Assertions, they are easily defined in the Versata Studio as shown in Figure 11-6 and Figure 11-7.



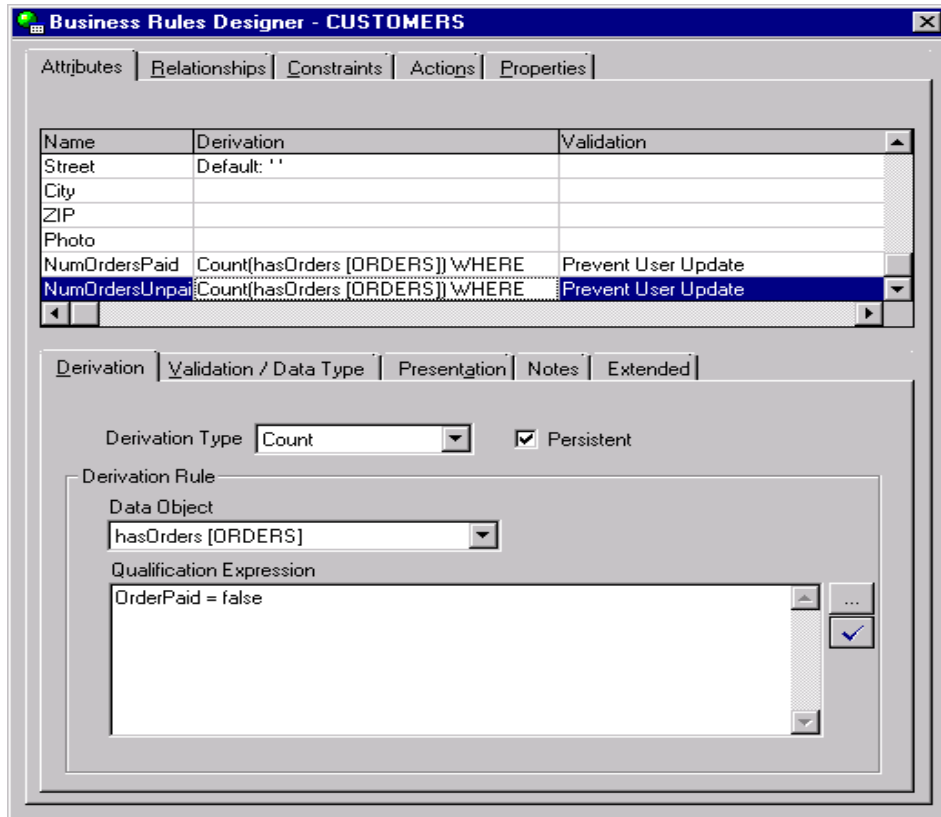


Figure 11-6 First rule to enforce business assertion derives unpaid order

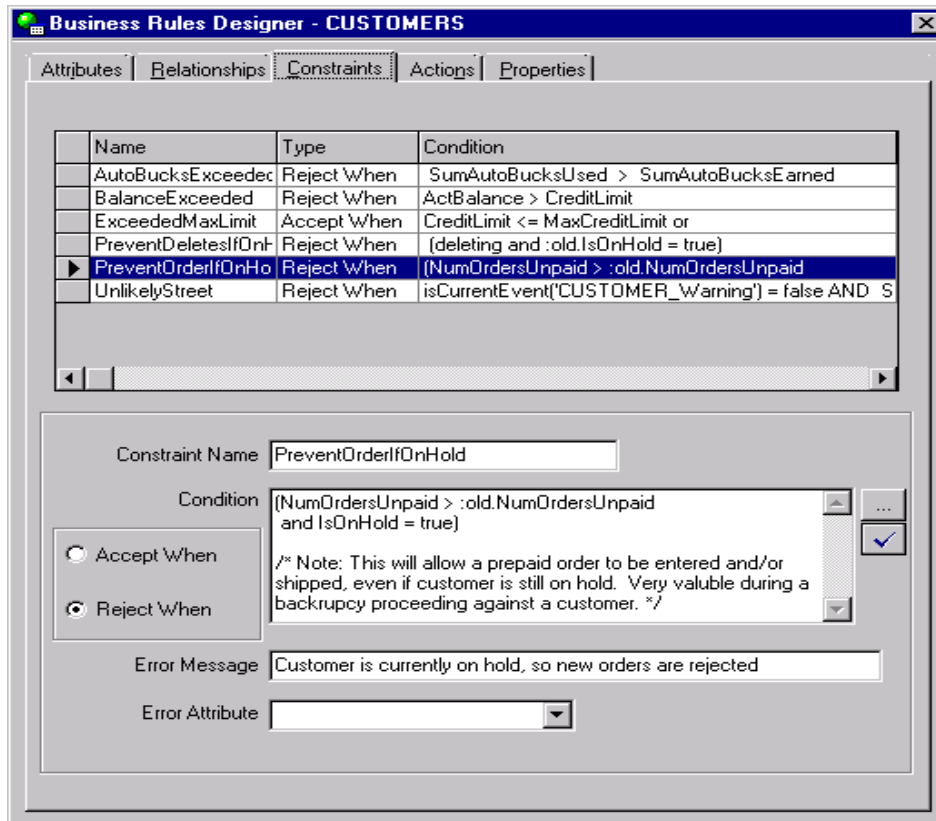


Figure 11-7 Second rule constrains transactions which violate the state

#### 4. Detailed class diagrams

Using the Conceptual Model as input (updated as Use Case design unfolds), the Data Model can be elaborated as Class Diagrams. The complete data model may include additional classes for building many to many relationships, introducing new attributes to derive values or for performance, and provisions for inheritance and Object-Relational mapping. Versata best-practice guidelines include many recommendations for conceptual and physical data modeling.

#### 5. Transfer business logic to the Versata Studio

In most cases, the detailed data model can be transferred to the Versata Studio from the modeling tool. This can be done through adapter technology provided by Versata, or by pushing the data model to a prototype relational database, where it can be introspected by the Versata Studio. (The introspection process was how the Trade application was begun.)

**Note:** The Versata Best Practice guide Re- Engineering Between Rational Rose and Versata Logic Studio, provides details on moving object models from Rational Rose to Versata and re-synchronizing with Rose when Versata models changes.

Rules can then be created based on Business Assertions and Requirement Interaction Diagrams.

## **6. Prototype the user case model using Versata (if desired)**

The final proof that a Use Case clearly understood is a working prototype. As we have seen, the Versata Presentation designer can be used to create a user interface from business objects.

If a simple Use Case Model is needed, a quick Versata application can be build for each Use Case. During this phase, page layout need not be refined - the focus is on basic flow, attributes and semantics.

Versata recommends that the “User Experience” be extracted from activity diagrams. Within Rational Rose, forms can be associated with data classes using input form stereotype. When defining the Versata application, the form stereotype will be used as the application page data source.

**Note:** Versata is developing a more direct linkage between Rational Rose forms and the Versata Presentation Designer.

### **11.2.3 Versata and the Construction phase**

There are typically two tasks that fall-outside of Versata automation — connectivity to legacy systems or third-party applications, and connectivity to non-Versata client-tiers. The two tasks will require the majority of the custom development and integration when using Versata-automated business logic.

#### **Custom connectivity**

Connectivity to legacy systems, including CICS, MQSeries, and any other system with a documented API is supported through the Versata Connector layer outlined in Chapter 4. Recall that the Versata Connector architecture is a generalized access framework for accessing relational databases (by leveraging WebSphere Services) and non-relational sources leveraging other technologies such as IBM's Common Connector Framework (which is evolving into the standard JCA.)

The connection type for a Versata business object is specified in the Versata Studio as part of the object's properties. Figure 11-8 illustrates, that the Customer object will persist its data, not to a database, but to an MQSeries queue.

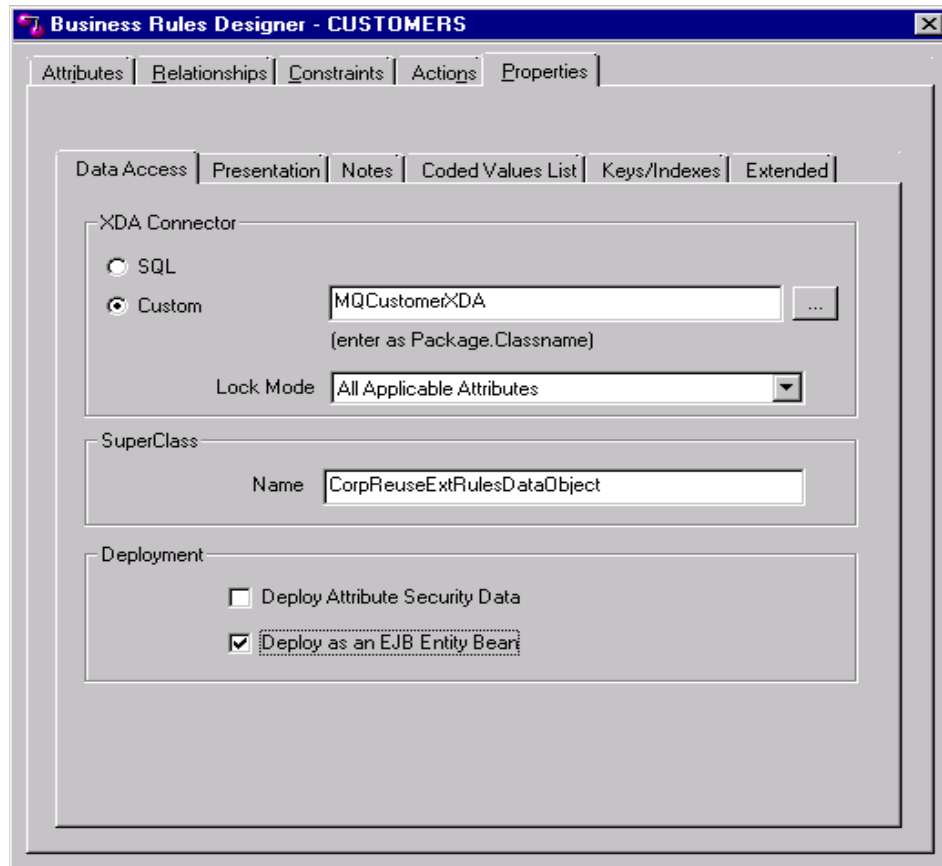


Figure 11-8 Connecting to a custom connector class

Building custom Versata connectors for business objects is outside the scope of this redbook. The reason for mentioning connectors here is that connector development may be scoped and planned for as part of the overall software engineering process.

The complexity of a connector will vary depending on the actual system to be integrated. from simple calls to CICS or IMS transactions to get data, to a generalized interface to push a variety of dynamic messages to several MQSeries queues.

Detailed Use Cases and Data Models will reveal how specific or generalized the connector must be, as well as its inputs and returns. Using these designs, a prototype implementation can be developed during the Elaboration Phase or left to the Construction Phase. If left to the Construction Phase, the initial prototype can be executed against a simpler data store (such as a relational database) and re-directed to the permanent connector using the Versata Studio.

### **Connectivity to non-Versata client-tiers**

The final area of construction and integration is connecting Versata Business Logic to a non-Versata client-tier. This is the process we observed in Chapter 9, when we connected the existing TRADE client JSP pages and servlets to the Versata Logic Server. We mention it again here to understand its sequence in the development process.

Using the RUP process, basic GUI prototypes are often produced Inception Phase to gain user buy-in. As IBM explains in the User-to-Business pattern, such prototypes usually contain a view of static contents (for example HTML pages) and an interaction controller built into the HTML code that provides basic navigational functions to fulfill the tasks of a GUI prototype. There is typically no business logic or model layer.

During the Elaboration Phase, the client application flow can be extracted from the activity diagrams. HTML page elements can be extracted from data classes. When using Rational Rose, this can be done using the same input form stereotype that is used to produce Versata-constructed applications.

With a well-elaborated model, business logic development within the Versata Studio and presentation-logic development in other tools can proceed in parallel.

When linking to a non-Versata client to Versata business logic during the prototype phase, it may be sufficient to replace the static HTML controller logic from the early prototype with Versata-provided JSP tags and beans from the Versata JSP toolkit. These tags and beans enable JSP pages to execute dynamic business object queries or pass data to Versata Logic Server for further processing.

If a more sophisticated architecture is being prototyped, such as the TRADE Model-View-Controller example, an interaction controller function for each use case will be need to be created. This is usually done with a servlet, and supporting Java classes.

As we saw when porting the TRADE Client, typically a controller function will connect to the Versata Logic Server using the Versata Client Library, submit a query, insert or update command, and pass the results to the appropriate JSP for display or further processing.

Interaction diagrams and class diagrams provide the client-tier developer with details on the object names and attributes available through the Versata Client Library.

**Note:** Although we consider only HTML-based clients in this Self-Service pattern, any component capable of calling the Versata client libraries or EJB interfaces can access rule-based logic. For instance, the IBM Redbook ***Web Services Wizardry with WebSphere Studio Application Developer*** (SG24-6292) shows how to use the Versata Logic Server, Versata's XML interface and Versata's WebServices toolkit with new capabilities in WebSphere Studio Application Developer. Its WebService scenario shows how to use Versata rules to automate sophisticated B2B logic that is wrapped with a client proxy using the WebService Description Language (WSDL).

## 11.2.4 Versata and the Transition phase

The Transition Phase turns the constructed and integrated system to the user community for acceptance testing.

The marriage of UML and Rules should assist in gaining user acceptance, or when modifying the system when the implementation falls short of the users' expectations.

First, the UML Policy and Business Assertion and Rule extensions proposed by Versata enable traceability directly from the intended goal of the system to its rule-automated implementation.

Next, UML Rules provide a common language that bridges the gap between business analysts and developers. Analysts view rules as a business language; Developers view them as an implementation language.

Additionally, the rule-based repository provides its own level of detailed documentation. Detailed reports of all business and application objects and logic are available from the directly from the Versata Studio. These reports are guaranteed to be up-to-date since their XML-definitions were used to automatically create the system.

Finally, changes to business logic can typically be made with high-level changes to rules. Changed rules are inserted into the system, immediately re-synchronized with exiting rules and immediately re-integrated with existing Versata-created components. Upon redeployment, the system is automatically re-optimized for the new business logic.

## 11.3 Conclusion: Rule-based design and development

Most Versata users report that the biggest payback in rule-based development is not obtained in the first iteration of the system, but in subsequent iterations and maintenance. As we've seen in this chapter, this reflects the relatively high percentage of time that must be dedicated to analysis and design at the beginning of any project. Using RUP, this is in the critical Elaboration Phase.

However, logic automation greatly reduces construction, testing, and subsequent iterations by making careful design and specification directly executable as the business logic tier of WebSphere applications.







## A Versata Case Study: American Management Systems

In the competitive enterprise software arena of Enterprise Resource Planning (ERP), AMS has specialized for the past 25 years in understanding the needs driving the purchasing, revenue, and administrative operations for over 250 customers across state and local government as well as in higher academic institutions. In a move by state and local government (just as in almost all other industries) to bring key internal and external processes online such as vendor self-service and procurement, AMS is moving to update its ADVANTAGE Suite of Administrative Solutions to the next generation based on a thin-client, Java 2 Enterprise Edition (J2EE) model.

As they migrate this application from a client/server COBOL-based system to a distributed Web-based and Java-based model, AMS also has to ensure that its solution can meet the following design objectives such as:

- ▶ Ensure that performance meets enterprise requirements
- ▶ Provide the application flexibility to alter functionality based on changing business requirements
- ▶ Protect its existing customer base by providing richer baseline functionality and enhancing the user interface and overall experience of the application

In addition to these objectives, there is a substantial need to lower implementation costs. According to Dan Keene, Vice President of State and Local Solutions at AMS, the supply of the advanced technology skills in state and local governments to build and maintain applications can often be limited. Keene notes, "There is huge overhead required to maintain Web and distributed applications that many people don't factor into their acquisition of technology."

Finally, ERP application requirements are fairly sophisticated, unique, and continually changing within each organization. They typically involve complex business logic that drives the end-to-end procurement process such as workflow and routing, collaborative buying, and vendor self-service.

## 12.1 The technical decision process

For a solutions provider like AMS, making a "bet-your-business" technology decision is not a task to be underestimated. In a decision that involved the CTO's office, the Head of Product Engineering, and the General Manager of the ADVANTAGE Line of Business, Keene states that there were several critical business requirements that influenced the technology direction for AMS. These included:

- ▶ Reducing AMS' cost of development, which meant reducing actual development and training time and leveraging the domain experience of existing staff to implement new, Java and Web-based solutions
- ▶ Reducing product time-to-market to stay ahead of the competition
- ▶ Relying on a strong partnership with best-of-breed technology providers who understand AMS customer requirements such as flexibility, customization, and performance

Unlike other ERP solutions where the vendor's business logic, or rules and processes, are uniformly implemented in all organizations who purchase the product, the AMS design philosophy required that their solution could conform to the unique business practices of a particular institution. The ADVANTAGE Suite of Administrative Solutions Version 3.0 also required that:

- ▶ Business requirements could be defined and configured by business analysts so that changes could be made without needing advanced technology specialists
- ▶ Moving to a Web-based solution would not create huge overhead to support the application
- ▶ Enterprise level scalability and performance supporting complex, data-intensive transactions could be met

Based on these requirements, AMS knew they needed to take a declarative business logic approach to get that level of flexibility and abstraction from the underlying technology infrastructure. AMS initially attempted to build their own development, rules-based framework to automate and host the ADVANTAGE application. They quickly eliminated this option due to its high cost. Keene estimates that it would have taken twelve months to build the framework alone; this timeframe didn't include the 16 to 18 months required for building the application itself. Simply, Keene stated, "AMS would not get to market quickly enough to be competitive and to meet our customer demands."

## 12.2 The Versata decision

After looking at several development solutions, the Versata Logic Server was chosen as the best option. Not only did Versata address the enterprise-level software requirements of an ERP solution, they offered a viable partnership that suited AMS' requirements for their next major release. According to Keene, "Due to the nature of our application, the relationship between AMS and Versata called for more than the standard acquisition of technology. Versata's engineering team met with us to collaborate on performance benchmarking and feature enhancements. Ultimately, this relationship is what closed the deal and gave us the assurance to move forward with such an important decision."

In addition to the previously mentioned requirements, the ADVANTAGE application had specific criteria in the area of performance, scalability, transaction management, and architecture. Because AMS has a wide range of customers, the ADVANTAGE application needed to scale both vertically within the application as well as horizontally across multiple application servers. For example, a typical configuration for the ADVANTAGE application is one with 3,000 signed on users and 800 concurrent users. Through a series of benchmarks and performance milestones, Versata met these requirements.

Finally, AMS required an open, standards-based J2EE-compliant architecture that initially supported IBM WebSphere, with the potential for supporting other application servers in the future. Supporting IBM WebSphere, Versata ensured that AMS could meet its customers' requirements. AMS was also looking for a partner to take on the responsibility for supporting the application server in order to avoid costly application rewrites. Using Versata enabled AMS to avoid the common pitfalls in J2EE such as upgrading from a J2EE 1.2 compatible application server to a J2EE 1.3 compatible release. Often, a new version of the Enterprise JavaBeans (EJB) specification is different from the previous version; therefore firms often must budget for recoding and retesting their EJBs.

## 12.3 The project

Based on meeting the key design criteria, AMS officially approved the use of the Versata Logic Server for the ADVANTAGE 3.0 project, and as of January 2000, the Versata Logic Server became the declarative business logic solution for automating the business logic development and management in the application server infrastructure of AMS' state and government practice.

### 12.3.1 The team

The initial AMS development team was comprised of 120 developers for the first three months and 60 developers for the remainder of the project. Thirty percent were Java and Object-Oriented developers. Seventy percent were developers with PowerBuilder or COBOL backgrounds. Until attending class, few team members had experience with declarative business logic, let alone the Versata Logic Server. According to Mike Titmus, CTO and Vice President of AMS, "The development staff new to Java was able to come up to speed using Versata in three to four weeks and were completely proficient in three to four months."

Included on the team were several technical architects focused on both infrastructure using the Versata Logic Server and the IBM WebSphere Application Server, as well as integration issues with the core application. Originally, AMS also had a team of user interface (UI) experts. But with the level of automation Versata provided for the application, AMS quickly narrowed this down to one UI expert and a business analyst that would gather the key requirements needed to drive the user experience.

## 12.3.2 The application

With over 3.7 million lines of code comprising the existing COBOL-based ADVANTAGE application, AMS wasn't under-estimating the vendor commitment that they required for supporting the project.

Table 12-1 ADVANTAGE application statistics

Item	Statistics
# Tables	1200
# Data Objects	833
# Query Objects	117
# Forms or Pages	849
# Coded Value Lists	273
# Business Logic Statements	12,000 (replaces 3.7 million lines of code)

Possibly one of the largest enterprise IBM WebSphere applications in existence, let alone enterprise Java applications in existence, the ADVANTAGE application's complexity is best illustrated by:

- ▶ The type of information it processes
- ▶ The type of transactions it manages
- ▶ The overall scale of the application

For example, a transaction in the ADVANTAGE Solution requires processing a very large set of data to complete a transaction. Like a physical paper file that stores volumes of information, one particular document may require information stored in multiple tables. Much like TurboTax where data is stored and held each time you enter and exit the system, the ADVANTAGE application has a multi-step validation process where information is stored in multiple tables. Any user action could result in updating data in 10 tables, or in some cases 250 tables. During one of the performance tests with Versata, AMS tested a complex, online update involving more than 750 rules.

Additionally, because of the nature of procurement, the ADVANTAGE application has significant integration requirements with other key back-end systems found in state and local government. These systems include mainframe data on OS/390 as well as with XML exchanges needed to integrate with key partner systems.

According to Keene, without a declarative approach to automate the business logic and user interface, building this application by hand would have been a daunting task. Keene estimated that the Versata Logic Server automated over 85 to 90 percent of the business logic and 98 to 99 percent of user interface. Most of the hand-coded components relate to the architectural decisions, batch processing, and other common modules across the application.

### 12.3.3 A specific look at performance

For an institution to entrust its business operations to the ADVANTAGE solution, AMS needed a vendor with reliable technology and a commitment to performance.

*Table 12-2* ADVANTAGE performance statistics

Item	Statistics
# Signed on Users	3000
# Concurrent Users	800
Response Time	2.8 seconds

With a call for a heavy investment in performance analysis and tuning, AMS needed to work shoulder-to-shoulder with a partner to meet these requirements. The enterprise requirements for the ADVANTAGE solution called for:

- ▶ Supporting 3,000 signed on users
- ▶ 800 concurrent users
- ▶ 2.8 second response time for small and medium sites

Transaction loads vary from client to client. According to Titmus, "Because performance and throughput for each client varies, the application load must scale both vertically throughout the application and horizontally across multiple application servers. For example, a medium size client might have 7,500 transactions in an online, 8 hour day and a smaller size client might have a much smaller number of transactions over that same period. The application must be flexible to support these different requirements." Additionally, the transactions themselves are diverse and complex. One transaction may consist of hundreds of inserts, updates, and deletes to the underlying relational database.

AMS and Versata managed this demand by conducting a series of benchmarks to ensure that specific milestones were being met. In fact, AMS recently tested their application at the IBM Test Center to certify these measurements. The characteristics of the benchmarks included the number of pages accessed, the response time, and the number of screens for both small and medium sites.

*Due to the nature of our application, the relationship between AMS and Versata called for more than the standard acquisition of technology. Versata's engineering team met with us to collaborate on performance benchmarking and feature enhancements. Ultimately, this relationship is what closed the deal and gave us the assurance to move forward with such an important decision.*



## 12.3.4 System architecture

The diagram in Figure 12-1 depicts the ADVANTAGE system architecture with IBM's WebSphere as the application server foundation for distributed, enterprise computing offering services such as data caching, load balancing, session management, connection pooling, and failover.

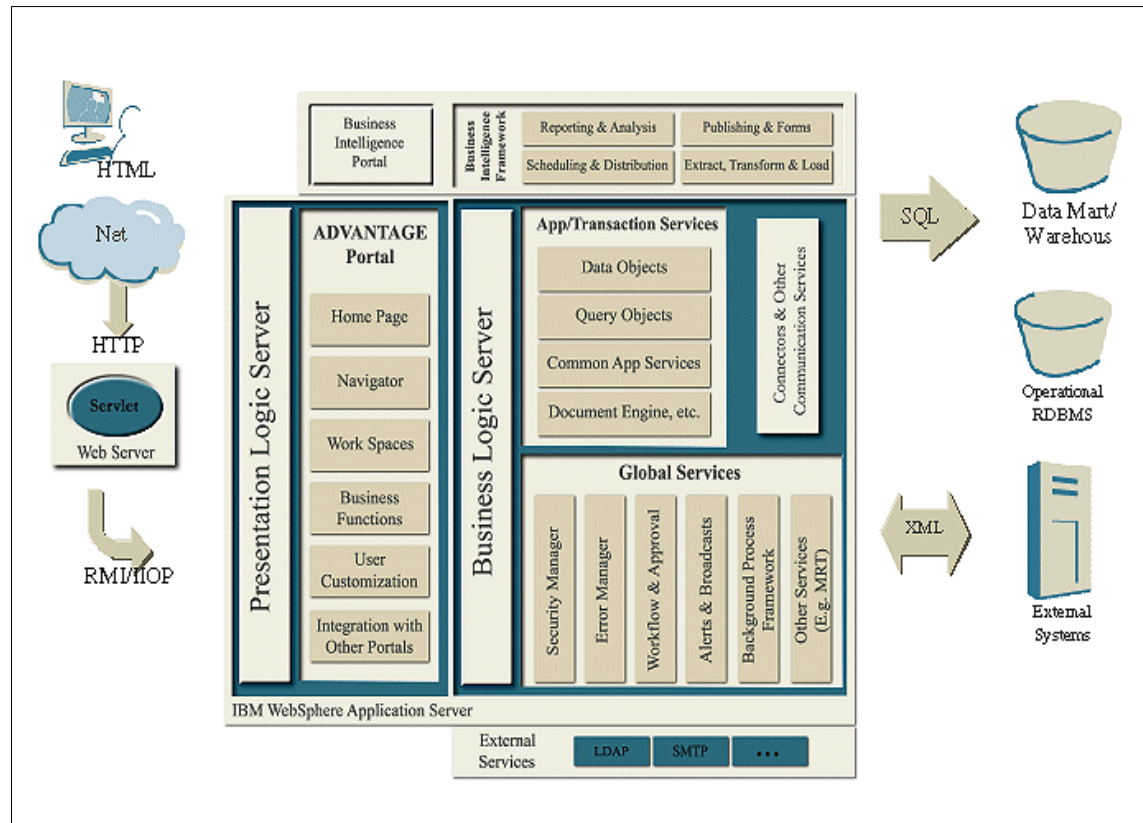


Figure 12-1 ADVANTAGE system architecture

The Versata Logic Server's Transaction Logic layer provides the application definition and transaction services required by the ADVANTAGE application as well as the global services such as security, workflow, and alerts. AMS estimates that 85 to 90 percent of the specific business logic is automated by the Versata Logic Server.

The User Interface level is automated by the Versata Presentation Engine, which incorporates the frames or archetypes to support specific business functions. With 849 pages, AMS needed a template-based approach as it would have been too cumbersome to layout each specific page. Using an iterative design approach allowed them to review design with business analysts earlier in the cycle and support continuous change. The distinct advantage of automation is that the system implementers can change aspects of the application, and the UI automatically reflects these changes. This layer provides the key navigation, business functions, and user customization needed by the ADVANTAGE portal. AMS estimates that 98-99 percent of this functionality is automated by the Versata Logic Server. Furthermore, using a thin-client model that separated the interface from the business logic allowed the business logic to be re-used across multiple user interfaces such as Java, HTML and XML.

One of the key benefits of the Versata Logic Server that AMS passes on to its clients is the ability to easily customize and configure methods in the ADVANTAGE application. AMS includes an ADVANTAGE Studio modeled from the Versata Logic Server so that users can:

- ▶ Redefine data model and business logic
- ▶ Create and modify business objects
- ▶ Declaratively specify business logic at a business level as opposed to at the hand-written Java code level
- ▶ Store all application information in a standards-based XML repository so that it can be easily shared across the application and across teams.

### 12.3.5 The schedule

While AMS calculated that Versata saved them 12 months in building their own logic-based development platform, there was still the difficult task of working with the very complex ADVANTAGE application, which called for enterprise-capable design and development procedures.

Table 12-3 ADVANTAGE schedule

Date	Tasks
October 1999	Design begins
January 2000	Versata class Development begins with 120 people
August 2000	Development scales back to 60 people
Ongoing	Iterative development - build test release Build 38 implemented
January 2002	Deployment

With initially 120 developers on the project and the design work already completed, AMS launched into development in January 2000. Versata conducted an immersion-style training program for the first wave of 15 developers. After that, AMS adopted a "train the trainer" approach due to the size of its development team and moved the training in-house.

AMS estimates they could have implemented the application more quickly if they had not scaled back effort midstream due to other business priorities. Therefore, AMS calculated that Versata saved them at least 12 months in the development process in terms of both providing the server for the application's business logic as well as in the construction of the application user interface itself.

One of the most valuable and critical aspects of the development process for AMS was the ability to review key functionality and enhancements with its customer base. Nearly a year into development, AMS reviewed its initial implementation with its customers. It was determined that significant changes needed to be made to key functionality as well as to the overall "look-and-feel" of the application. Due to the high degree of flexibility derived from Versata's declarative approach, the new changes and a complete new design were implemented and put into user testing in only three months' time.

This iterative development cycle, a concept that Versata directly enables, was a key risk mitigation strategy for AMS. According to Keene, "making critical changes even during development is far easier to make with Versata regardless of how far down the path you are."

### **12.3.6 Development/deployment issues**

Due to the nature and sophistication of the AMS application, the relationship between AMS and the Versata engineering team was critical particularly in addressing performance benchmarks, high volume transaction requirements, and feature enhancements based on feedback from the AMS team. Because of AMS' unique transactional model (for example, partial rollback), AMS required very close interaction with Versata's engineering team to implement that level of customization. Being able to finely tune and test the ADVANTAGE application was critical. Keene stated, "Both Versata and IBM stepped up to the plate in insuring that we met the levels required by the application."

## **12.4 The bottom line and the future**

Not only has AMS received significant benefits from using the Versata Logic Server in their ADVANTAGE application, Keene believes that using the Versata Logic Server in its next application for Human Resources and Payroll will generate even more significant savings. AMS predicts the effort with Versata to equal nine months worth of development time instead of the 18 months previously scoped, meaning AMS expects to realize a 50 percent savings in future development efforts.

Looking forward, AMS also sees the importance of building more institutional knowledge into the system where business analysts can have greater access to the business logic. With the median age of state and local employees rising, institutional knowledge on how operations and processes work are at risk of being lost as they are not documented in any system. This is critical in an environment where legislative or local mandates require significant changes to the application. According to Keene, "There is at least 20% that can't be planned for as new rules and requirements continually impact business processes. Business analysts will need to have greater access to the business logic directly and be able to make changes as needed. The declarative business logic approach documents this knowledge and makes it accessible across the organization."

AMS believes that the Versata Logic Server enabled them to meet their original objectives of reducing development costs and insuring their customers achieve a level of flexibility in easily making changes to the application. Titmus offered that "by using Versata to upgrade ADVANTAGE, we were not only able to use our existing staff resources, but also to get to market quickly in order to remain competitive and are able to easily change and customize the application to meet customer demands."

**Note:** Reprinted by permission of:

Versata, Inc. 300 Lakeside Drive, Suite 1500, Oakland, CA 94612 USA

[www.versata.com](http://www.versata.com)

toll-free 1.800.984.7638

ph 510.238.4100

fx 510.238.4101





**A**

## **Benchmark results**

IBM provides the WebSphere Performance Benchmark Sample (Trade2 application) to bring a robust model application to the industry for two purposes. One reason is to give customers the ability to model their own J2EE platform technology-compliant applications against a sample for best-practice approaches to writing applications. Another reason was to provide customers and ISVs with an application to test the performance of systems.

At IBM's suggestion, Versata ported the business logic layer of the Performance Benchmark Sample from the set of components created by IBM with VisualAge for Java to an equivalent set of components created automatically by Versata. The purpose of the port, and subsequent benchmark activities, was to test and compare scalability and throughput of the two approaches.

# Benchmark configuration

The benchmark was conducted in September 2001 on the following hardware and software configuration.

## Application

IBM Trade Benchmark 2.5.3.

## Configuration 1

The IBM primary mode was used. The application was accessed through the TradeAppServlet (client tier). Server logic included the Trade session bean and CMP entity beans. A variety of EJB optimizations were employed including VisualAge for Java Access Beans to EJBs, caching of JNDI look-ups and course-grained objects access. Components are detailed in the presentation “High Performance Web Apps” (trade2Performance.pdf) provided with the benchmark download from IBM.

## Configuration 2

The Versata primary model was used. The application was accessed through the TradeAppServlet (client tier). Server logic included Versata Logic Server components automatically compiled from rules. Versata components were deployed with optional EJB-interfaces. Access was through Versata-client libraries.

## Hardware and software

The test was run on WebSphere Application Server, Advanced Edition, Version 3.5.4. The WebSphere platform was a 4 CPU Dell PowerEdge 6300 running NT 4.0 SP5. The WebSphere Application Server accessed a single database system with the same configuration (4 CPU Dell PowerEdge 6300).

The benchmark was driven by 2, 2 CPU Dell PowerEdge 2400's executing LoadRunner Controller 6.5 software to simulate the client load.

Twenty-minute performance intervals were used for each configuration. The recommended IBM procedure was followed before each run to reset the Trade runtime to a clean starting point.



## Results

The results displayed in Figure A-1 show less than a 3% difference between the IBM primary mode (Configuration #1) and the Versata primary mode (Configuration #2). The average throughput for Configuration #1 was 258 transactions per second. The average throughput Configuration #2 was 250 transactions per second.

The results suggest that the performance and scalability of the WebSphere Application Server is equally effective regardless of whether the application logic was developed and optimized by hand, through VisualAge for Java, or created automatically by Versata. See Figure A-1.

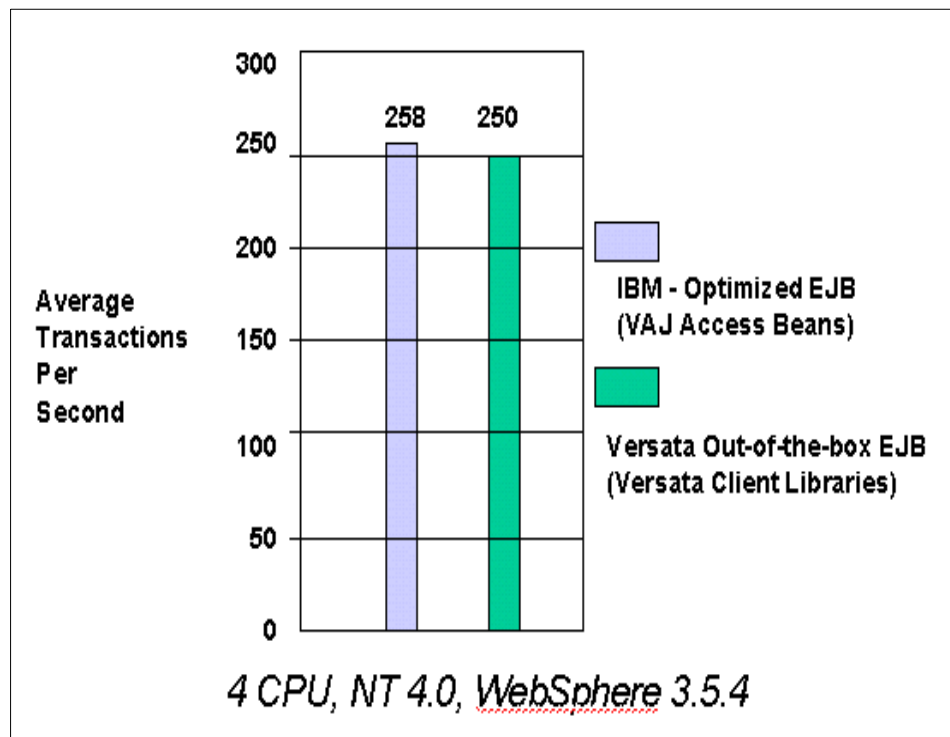


Figure A-1 Benchmark results

## Extrapolation to extended Trade2 logic

Although the WebSphere Performance Benchmark Sample was designed to be more realistic than other Web benchmarks in the industry, the business logic is still rather simple. Extending that logic, as we do in this Redbook, will involve significantly more interactions between EJB components. This intra-object logic is optimized is currently optimized by Versata through its local Java class “shadows” for EJBs.

Therefore, we can extrapolate that Versata-based applications may not experience the performance hit that is typically incurred by EJB-intensive applications. This performance advantage may persist, at least until the EJB 2.0 specification (with local object capabilities) is fully implemented by the application server.



# B

## Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

### Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246510>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246510.

### Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
<b>SG246510.zip</b>	Zipped Code Samples

## System requirements for downloading the Web material

The following system configuration is recommended:

<b>Hard disk space:</b>	20MB minimum
<b>Operating System:</b>	Windows 2000
<b>Processor:</b>	Pentium 3 or higher
<b>Memory:</b>	512MB or more

## How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Refer to Chapter 10, “Integrating Versata Logic Suite with WebSphere Studio Application Developer” on page 167 for information in how to use the Web material.

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 231.

- ▶ *Web Services Wizardry with WebSphere Studio Application Developer*, SG24-6292

## Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ Versata Corporation  
<http://www.versata.com>
- ▶ IBM Patterns for e-business  
<http://www.ibm.com/developerworks/patterns>
- ▶ Trade Application  
[http://www.ibm.com/software/webservers/appserv/wpbs\\_download.html](http://www.ibm.com/software/webservers/appserv/wpbs_download.html)
- ▶ Business Rules Group  
<http://www.businessrulesgroup.org>

## How to get IBM Redbooks

You can order hardcopy Redbooks, as well as view, download, or search for Redbooks at the following Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

You can also download additional materials (code samples or diskette/CD-ROM images) from that site.

## **IBM Redbooks collections**

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

# Index

## A

A constraint to limit margin selling 140  
Account attributes for display 90  
Account Record source properties - HTML object tab 89  
Account RecordSource on the home page 84  
Account relationships and referential integrity 34  
Action example 30  
activity diagrams 195  
Add relationship rules 62  
Adding business object method to the remote interface 162  
Adding new data object 122  
Adding transaction attributes and validations 123  
Agent pattern 8  
Allowing one Holding row to be inserted 100  
Alternative for JSP access - Versata JSP Toolkit 163  
American Management Systems 211  
AMS 211  
AMS - ADVANTAGE system architecture 219  
AMS - application 216  
AMS - project 215  
AMS - specific look at performance 217  
AMS - team 215  
AMS - technical decision process 213  
AMS - Versata decision 214  
Analyze interesting state changes 200  
Application and Runtime patterns 3  
Application pages before transitions 102  
approaches for developing in Versata when using RUP 194  
Archetypes - a brief overview 83  
Architecture of the Versata Logic Server within WebSphere 39  
As-is host pattern 7  
Assigning data objects to the TradeX data server 111  
Attaching the enterprise application to the server 184  
Attaching to server configuration 185  
automatically optimized for runtime 24

## B

Basic business logic in Trade 17  
Beginning application design 82  
Beginning client application deployment 114  
Beginning the getPortfolio() method 159  
Beginning to deploy the business logic tier 105  
Benchmark - Application 226  
Benchmark - Configuration 1 226  
Benchmark - Configuration 2 226  
Benchmark - Hardware and software 226  
Benchmark - Results 227  
Benchmark configuration 226  
Benchmark results 225  
Building the query for the home page 93  
Business logic automation using rules 19  
Business object deployed as local classes with EJB faces 47  
Business object deployment 104  
Business patterns 2, 4  
Business uses of rules 31

## C

Calling a method to return the commission 138  
Changes to TradeAltAccess to accommodate Versata 153  
Characteristics of rules 24  
Choosing Account/Holding client coordination 96  
Choosing displayed attributes for Holding 98  
Choosing the application style 82  
Choosing the archetype for Holdings 97  
Choosing the archetype for the Account Record source 85  
Choosing the archetype for the home page 86  
Classification of declarative logic (rules) 26  
Classpath variable setup 172  
Client application deployment 114  
client libraries 56  
Collaboration pattern 4  
Compiling the Versata application in debug mode 170  
Completing the application design 102  
Completing the business object deployment process 105

- Completing the Relationship rule 63
- Component directories in the Application Deployment Wizard 115
- Composite patterns 2
- Computing a new GrossProfit attribute in the query 145
- conceptual class diagrams 197
- Concluding the TradeX extended business logic 149
- Configuring the server instance 185
- Configuring the server to test the application 182
- Confirming application deployment 116
- Connecting to a custom connector class 206
- Connectivity to non-Versata client-tiers 207
- Connector layer 50
- connectors 56
- Constraint example 29
- Construction Phase 194
- Container managed EJBs 16
- Copying the required files 169
- Create a new repository 61
- create and test Servlets, JSPs, and Enterprise Java Beans 168
- Creating "QueryObjects" 142
- Creating a classpath variable for the Versata install directory 171
- Creating a non-persisted attribute for the transaction amount 67
- Creating a server instance and configuration 182
- Creating data server properties 110
- Creating the Account/Holding relationship 62
- Creating the Portfolio page 101
- Creating the Profile page 101
- Creating the source EAR file 170
- Custom connectivity 205
- Customized presentation to host pattern 7

## D

- data objects 55
- Data objects (left) and account attributes and rules (right) 43
- Database deployment 106
- Database schema 15
- Database scheme 15
- Debit and credit methods added to the Account Java source 70
- Debiting the account balance in a transaction 130
- Debugging your application 190

- declarative 24
- Declaring the buy() method 157
- Decomposition pattern 7
- Defining the action to debit the user's Account 71
- Defining the data objects to be joined in a QueryObject 143
- Defining the QueryObject sort order 146
- Defining the transaction and holding relationship 125
- Defining the ValidTransTypes as a coded value list 127
- Defining TradeX users to the logic server 112
- Defining transaction primary and foreign keys 124
- Deploying the TradeX application 103
- deployment to database 107
- Derivation example 28
- Deriving AccType from total assets and number of transactions 136
- Deriving QtyOnHand of a holding 134
- Design and runtime environment 21
- Designing an HTML client application 73
- Designing the Home page 84
- Designing the QuoteBuy page 94
- Detail class diagrams 204
- Details of the Trade EJB implementation 15
- Developing with UML and rules 193
- Directly integrated single channel pattern 6

## E

- EAR import 177
- EAR module import 179
- Editing transition properties 92
- EJB 2.0 - Container Managed Relationships (CMR) 52
- EJB 2.0 - local interfaces 53
- Elaborate key use cases 200
- Elaboration Phase 194
- End-to-end processing 17
- Enhancing TradeX business logic 119
- Example of a rule 22
- Executing an application from the Versata Studio 117
- Executing deployed applications 117
- Exporting the application from WSAD 191
- Extended Enterprise 4
- Extrapolation to extended Trade2 logic 228



## **F**

Firewall and hot backup configuration 42  
For the logic server EJB itself 55  
Four service objectives 55

## **G**

Generating business and application logic reports 118  
Granting access to TradeX users 112  
Granting permissions to the TradeX role 113  
GUIDE Business Rule Project 27

## **H**

Handling potential exceptions 158  
high-level policy model 196  
Home 148  
Home page 78  
Home page tab is based on the "TransProfit" Query-Object 148

## **I**

IBM Patterns for e-business 2  
Identify additional rules 64  
Import an object model 60  
Import DB2 schema and view data objects and attributes 61  
Import the application into the repository 191  
Importing applications into WSAD 171  
Importing EAR jar file 173  
Importing modified application into Versata 191  
Importing the Versata Logic Server code 172  
Importing your Versata application EAR file into WSAD 177  
Inception Phase 194  
Inference example 29  
Information Aggregation pattern 4  
Initializing the VSSession, VSQuery and VSResultSet 157  
Integrated testing with WebSphere Application Server 168  
Integrating the IBM Trade2 client 151  
Integrating Versata Logic Suite with WebSphere Studio Application Developer 167  
Integration patterns 2  
Introduction to the Versata Studio Business Logic Design 58  
Iterating over the ResultSet 159

## **J**

J2EE applications 168  
Java Connector Architecture (JCA) 53  
Just-in-time objects 49

## **L**

logic for multiple objects 24  
logic server EJB 55  
Login page 76  
Look to the future - EJB 2.0 and JCA 51

## **M**

Main properties tab for the Account Record source 91  
Managing the Versata Logic Server in WebSphere 41  
Manifest class path 178  
map the business assertion to two rules 202  
Method 1 - Using the Versata client libraries 152  
Method 2 - Utilizing EJB interfaces 160  
Model for processing a customer order 195  
Model-View-Controller (MVC) architecture 11  
Model-View-Controller architecture 11  
Modified client application using new business logic 142  
Most frequent question - What is it really? 40  
Multiple runtime modes 12–13  
MVC architecture of the Presentation Engine 75  
MVC architecture of the Versata Logic Server 45

## **N**

New application using a style 82  
New project name setting 175  
New requirements 120  
Note on project approaches and roles 58

## **O**

ODBC system data source name 107  
Other J2EE standards used by the logic server 53  
Overview of self-service pattern 1  
Overview of the completed application 76

## **P**

Passing attributes to a date comparison method 141  
Patterns for e-business 3

- pecifying the pop-up for viewing Quotes 99
- Persistence as a layer in the server MVC 45
- Portfolio page 80
- Potential enhancements to Trade business logic 18
- Preparing the EAR file with source 169
- Preparing Versata application for import into WSAD 169
- process model 195
- Processing Trade client requests to Versata business objects 153
- Profile page 81
- Project name input 174
- Properties for Holding Record Source on the Portfolio page 101
- Properties for the home page 88
- Properties for vlsBeans55\_EJB 176
- Prototype the user case model using Versata 205

## Q

- Query object in Trade shows Profit\_Loss attribute 44
- query objects 56
- QuoteBuy page 79
- QuoteBuy page and Quote Pick List 79

## R

- Rational Rose 194
- Rational Rose use case for a Versata-automated functions 198
- Rational Unified Process (RUP) 193
- Recap of the Versata logic services 54
- Redbooks Web site 231
  - Contact us xi
- Refine the page properties 87
- Relationship example 27
- Requirement 1
  - Sell partial holdings 121
- Requirement 2
  - Customize rules based on account type 135
- Requirement 3
  - Calculate commissions based on account type 137
- Requirement 4
  - Limit margin selling 139
- Restricting a QueryObject with an expression 144
- ResultSet access and just-in-time object instantiation 47
- ResultSets of objects 48

- Retrieving the first row of holdings 159
- Return the array of HoldingObjects 160
- Returning the new account balance 158
- Review of the steps 72
- Reviewing or setting the data server 108
- Router pattern 7
- Rule builder is used to “point-and-click” the rule 68
- rule constrains transactions 204
- Rule example 28
- rule specifying the value of ACTIVE\_HOLDING 23
- Rule that counts number of transactions for a holding 131
- Rule that restricts short selling 133
- rule to enforce business assertion 203
- rule to get price from quote object 65
- Rule-based development 57
- Rule-enabled objects as WebSphere components 46
- Rules and EJB domains 25
- Rules can automate data validation 31
- Rules can automatically synchronize related attributes in different objects 34
- Rules can enforce operational policies and respond to change when policies change 35
- Rules can identify interesting data 35
- Rules can initiate asynchronous events 36
- Rules can preserve the association between related objects when they change 33
- Run on Server 189
- Running and debugging the application 187
- Running your HTML application 188
- RUP phases and iterations 194

## S

- scope 199
- Self-service application patterns 6
- Self-service business pattern 5
- Self-service pattern 4
- Self-service pattern and the Trade application 8
- Sending updated ResultSet to logic server 158
- server instance creation 183
- Servlet icon 188
- Setting module visibility 184
- Setting the build properties for the application EJB project 180
- Setting the build properties for the application Web project 181
- Setting the build properties for the vlsBeans55\_EJB

- project 175
- Setting the module visibility 183
- Setting the sequence number inside an event 132
- Setting the values for the new row 157
- Setting up an ODBC Data Source Name (DSN) 106
- Specifying a business logic report 118
- Specifying the Account Record source on the QuoteBuy page 95
- Stand-alone single channel pattern 6
- Standards used for WebSphere version 3.5 54
- Starting the server 187
- State diagram for an order entity 201
- Supported functionality 164
- System policy model the Versata SampleDB example 197

## T

- tables to be created in database 108
- Tag library overview 165
- The TradeAltAccess class from IBM 152
- The TradeVFC buy() method 156
- The TradeVFC getPortfolio() method 159
- The TradeXv2 application 147
- The TradeXv2 repository 120
- Trade 12
- Trade application controller implementation 12
- Trade application functionality 10
- Trade application overview 9
- Trade client design using MVC 11
- Trade interface 154
- Trade user's home page 10
- TradeVFC method summary 154
- TradeVFC.java 154
- TradeX application model 74
- TradeX home page 78
- TradeX login page 76
- TradeXv2 home page 147
- Transaction and holding relationship details 126
- transaction independent 24
- Transaction rules - basis for automated business logic 22
- Transfer business logic to the Versata Studio 204
- Transition Phase 194

## U

- UML and rules 193
- UML and the Rational Unified Process 194
- Unique key autonumbered 66

- Universal Modeling Language (UML) 194
- unordered 24
- updateProfile method 161
- use cases and business assertions 197
- Use of copy helper access beans 17
- Using patterns for e-business 4
- Utilize sequence diagrams 201

## V

- Validating transaction types from the coded value list 128
- Validation 32
- Validation from a cached list of values 32
- Value-based access through Versata client libraries 49
- Versata 14
- Versata - a new option 14
- Versata and the Construction phase 205
- Versata and the Elaboration phase 199
- Versata and the Inception phase 194
- Versata and the Transition phase 208
- Versata business objects 43
- Versata Case Study
  - American Management Systems 211
- Versata design and execution environments 21
- Versata Logic Server 14
- Versata Logic Server classes 44
- Versata Logic Server Components created by installation 40
- Versata Logic Server console 109
- Versata Logic Server EJBs and servlet engine in WebSphere 41
- Versata Logic Server within WSAD 169
- Versata Presentation Designer 74
- Versata projects - Application developers 59
- Versata projects - Systems Architects 59
- Versata projects - Systems Level programmers 59
- Versata projects - Web technology specialists 59
- Versata repository 60
- Violation of attribution validation rule on Transaction.Quantity 149
- VSQuery 155
- VSRResultSet 156
- VSRow 156
- VSSession 155

## W

- WebSphere console after TradeX business logic de-

ployment 104  
WebSphere Performance Benchmark Sample 225  
WebSphere Studio Application Developer 168  
WebSphere Studio Application Developer (WSAD)  
167  
What are patterns 2  
What Versata Logic Suite Is not 36



## Application Development Using the Versata Logic Suite for WebSphere

(0.5" spine)  
0.475" <-> 0.875"  
250 <-> 459 pages







**Redbooks**

# Application Development Using the Versata Logic Suite for WebSphere

**Learn options for  
automating business  
logic in the EJB-layer**

**Explore declarative  
logic design using  
rules**

**Understand Versata  
Logic services in  
WebSphere**

Patterns for e-business are a group of proven, reusable assets that can help speed the process of developing applications. This IBM Redbook demonstrates a method of developing and managing the business logic in the “self-service business pattern” (formerly known as the user-to-business pattern).

The book describes the process of developing a stock trading application, based on the IBM “Trade” benchmark, using business logic rules to automate the construction and interaction of the transactional (EJB) components. It demonstrates substantially enhancing the business logic of the application through rule changes.

Two methods of constructing the presentation layer of the application are examined. The first uses Versata presentation automation techniques. The second adopts the Model-View-Controller (MVC) framework of the existing IBM Trade application.

The book demonstrates how to use the JSPs, servlets, and Java beans of the existing Trade application to interface to the EJB-based business logic and explains the role of the runtime Versata logic services installed into the WebSphere Application Server.

**INTERNATIONAL  
TECHNICAL  
SUPPORT  
ORGANIZATION**

**BUILDING TECHNICAL  
INFORMATION BASED ON  
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:  
[ibm.com/redbooks](http://ibm.com/redbooks)**

SG24-6510-00

ISBN 0738424218