

Developing Client/Server Applications

Version 4.3

October 1999

CONSTRUCT SPECTRUM™ SDK

Manual Order Number:**SPV430-021IBW**

Copyright © SAGA SOFTWARE, Inc., 1999. All rights reserved.

SAGA, SAGA SOFTWARE, the SAGA logo, Free Your Information, the FYI logo, CRIS, Construct, Construct Spectrum, Construct Spectrum SDK, iXpress, Sagacertify, Sagagallery, Sagavista, and Your Fastest Route to Enterprise Integration are trademarks or registered trademarks of SAGA SOFTWARE, Inc. in the U.S. and/or other countries. Adabas, Adabas Delta Save Facility, Adabas Fastpath, Adabas SQL Server, Adabas Vista, Adaplex+, Bolero, Com-plete, Entire, Entire Access, Entire Net-work, EntireX, EntireX DCOM, Entire Broker, Entire Broker SDK, Entire Broker APPC, Entire SAF Gateway, Natural, Natural Architecture, Natural Elite, Natural New Dimension, Natural Lightstorm, Natural Vision, New Dimension, PAC, Predict, Software AG, and Super Natural are developed by Software AG of Darmstadt, Germany and are distributed in the U.S., Latin America, Canada, Israel and Japan exclusively through SAGA SOFTWARE, Inc. and its subsidiaries and distributors. Adabas and Natural are registered trademarks of Software AG of Darmstadt, Germany. Except for Adabas and Natural, these products developed by Software AG of Darmstadt, Germany are either registered trademarks or trademarks of SAGA SOFTWARE, Inc., in the U.S. and/or other countries.

Other company or product names mentioned are used for informational purposes only and may be trademarks or servicemarks of their respective owners.

TABLE OF CONTENTS

PREFACE

Prerequisite Knowledge	16
How to Use this Guide	17
If You are Creating a New Client/Server Application	17
If You are Moving an Existing Application to a Client/Server Architecture	18
Conventions Used in this Guide	19
Related Documentation	20
Construct Spectrum SDK	20
Construct Spectrum	20
Natural Construct	21
Year 2000 Considerations	22

1. INTRODUCTION

What is Construct Spectrum?	24
Development Environments	25
Architecture of a Construct Spectrum Client/Server Application	28
Mainframe Server	29
Windows	31
The Development Process	33
Planning Your Application	33
Decide What to Show the User	33
Keep Window Design Simple	34
Number and Structure of Windows	34
Content of Each Window	35
Plan Your Code	35
Use a Consistent Style	35
Anticipate Translation Issues	35
Setting Up Your Application Environment on the Mainframe	36
Predict Definitions	36
Steplib Chains and Domains	36

Security for Domains, Steplibs, Users, and Groups	36
Generating Application Components	37
Using the Super Model.	37
Using Individual Models	37
Deciding Which Modules to Generate	37
Generation Process	37
Server Modules	38
Client Modules	39
Setting Up Your Project	39
Transferring Your Generated Code to the Project	39
Customizing Your Application and Environment.	40
Testing and Debugging Your Application.	40
Deploying Your Application.	41
2. USING THE DEMO APPLICATION	
Overview	44
Setting Up Prerequisites	45
Opening the Construct Spectrum Demo Project.	46
Understanding the Construct Spectrum Add-In	49
Understanding the Demo Project	50
Framework Components	50
Generated Modules.	53
Running the Demo Application	56
Application Interface	59
Menu Options	60
Toolbar Options	62
Application Workspace.	64
Status Bar	65
Additional Options	65
Error Notification Options	65
Remote Dispatch Service Options	66
Using the Demo Application	68
Opening a Business Object	68
Maintaining a Business Object	73
Validations	73
Business Data Types (BDTs).	75

Grids	78
Nested Grids	79
Nested Drop-Down Grids	79
Keyboard Shortcuts for Grids	81
Browsing For Business Object	81
Selecting Data With a Browse	82
Open a Business Object With a Browse	82
Open a Second Order to Work On	84
Open Foreign File Information	84
Specifying Browse Customization Options	86
Specifying Selection Options	87
Specifying Display Options	88
Troubleshooting	91
3. USING THE SUPER MODEL TO GENERATE APPLICATIONS	
Overview	94
Preparing to Generate with the Super Model	97
Using a Naming Convention	97
Understanding the Object Factory	99
Which Modules to Generate	100
Modules to Generate for a Maintenance Dialog	101
Modules to Generate for a Browse Dialog	102
Dependent Models	103
Generating with the Super Model	104
Using the Super Model Wizard in the Construct Windows Interface	104
Step 1 — Invoke the Super Model Wizard	104
Step 2 — Define General Package Parameters	106
Step 3 — Define Specific Package Parameters	109
Step 4 — Generate the Modules	113
Generating Modules from the Model Wizard	113
Generating Modules in Batch	113
Using the Super Model in the Generation Subsystem	114
Step 1 — Invoke the Super Model	114
Step 2 — Define General Package Parameters	115
Step 3 — Define Specific Package Parameters	117
Step 4 — Generate the Modules	119

What to Do If Something Goes Wrong	121
Transferring Your Application to the Client.	122
4. CREATING A CONSTRUCT SPECTRUM PROJECT	
Overview	124
Are You Ready?	126
Creating the Project	127
Prior to Downloading	130
Downloading the Generated Modules	131
Hand-Coding the Object Factory	133
What's Next?	134
Modifying the Dialogs	134
Testing the Application	134
Deploying the Application	135
Setting Up Security	135
5. CREATING AND CUSTOMIZING MAINTENANCE DIALOGS	
Overview of the Maintenance Dialog	138
Ways to Generate Maintenance Dialogs	139
The Process of a Maintenance Dialog	139
Are You Ready?	140
Using Individual Models to Generate Maintenance Modules	141
Generating the Object Maintenance Subprogram and PDAs	141
Generating the Maintenance Subprogram Proxy	142
Generating the Visual Basic Maintenance Object	142
Business Validations	142
Browse Functions	143
Generating a Maintenance Dialog	147
Downloading Client Modules	151
Integrating a New Maintenance Dialog	154
Strategies for Customizing a Maintenance Dialog	155
Doing the Predict Data Dictionary Work Up Front	155
Choosing an Appropriate Place to Add Hand-Written Code	156
Adding New User Exits	157
Making a Copy Before You Regenerate	158

Customizing on the Server	159
Deriving Variable Names	159
Deriving GUI Control Names	159
GUI Control Identifier	159
Object Identifier	160
Field Identifier	160
Deriving Label Captions for GUI Controls	160
Overriding GUI Controls	160
Step 1 — Search for GUI Keywords on Field Definitions	161
Generating a ComboBox Control to Display External Values	164
Step 2 — Search for GUI Keywords on Verification Definitions	164
Step 3 — Search for Business Data Type Keywords on Field Definitions	166
Step 4 — Use Default Derivation	167
Repeating Field Threshold	171
Option Button Threshold	171
Foreign Field Threshold	171
Setting Generation GUI Standards	172
Controlling the Size of a Maintenance Dialog	173
Overflow Conditions	174
Customizing on the Client	175
Creating Calculated Fields	175
Does a GUI Control Exist for the Calculated Field?	175
Coding the Calculation	176
Integrating Maintenance and Browse Functions	176
Validating Data Using the Visual Basic Maintenance Object	177
Tailoring the Maintenance Dialog	177
Working with Overflow Frames	179
Multi-column Layout	181
Tabbed Layout	182
State-Dependent Layout	183
Adding a New Field by Hand	185
Adding a Scalar Field by Hand	185
Adding a Regular Grid Column for a Field	189
Removing a Field by Hand	202
Using the Grid	202
Nested Grids	202

Nested Drop-Down Grids	204
Displaying Grids	205
Resizing Grids	206
Adding Sound to Error Notifications	208
Understanding How a Sound File is Associated With an Error	209
Multilingual Support for Maintenance Dialogs	211
Uploading Changes to the Server	212

6. CREATING AND CUSTOMIZING BROWSE DIALOGS

Overview of the Browse Dialog	216
About Browse Dialogs	216
The Browse Process	216
Browse Object Subprogram	217
Browse Object Subprogram Proxy	218
Visual Basic Browse Object	218
Data Cache	219
Framework Components	219
Creating a Browse Dialog	220
Setting up Predict for the Browse Dialog	220
Business Data Types	220
Descriptive Fields	220
Using the Construct Models to Generate Browse Modules	221
Generating the Browse Subprogram and PDAs	222
Generating the Subprogram Proxy	222
Generating the Visual Basic Browse Object	223
Defining Alternate Browse Data Sources	226
Downloading the Client Modules	229
Updating the Project	231
Extend Object Factory	231
Customizing On the Client	232
Adding Command Handlers	232
Customizing the Generic Browse Dialog	232
Understanding the BrowseManager Class	232
Display the Browse Dialog	232
Support a Browse Command Handler	233
Return a Specific Row of Data	233

Return All Rows of Data	233
Using the BrowseManager.	233
BrowseManager Methods.	237
Understanding Browse Command Handlers	238
Creating Browse Command Handlers.	240
Coding the Custom Browse Command Handler.	241
Enabling Commands on the Browse Toolbar and Menu	242
Coding the UICommandTarget() Method	242
Marking Updated Rows Using the UpdateListViewIcons Method	243
7. MOVING EXISTING APPLICATIONS TO CONSTRUCT SPECTRUM	
Overview	246
Moving Natural Construct Object Applications	247
Moving Non-Object Natural Construct Applications	248
Step 1 — Set Up Your Server Environment	248
Step 2 — Evaluate Your Application Data	249
Step 3 — Perform Optional Predict Set Up	249
Step 4 — Generate the Client/Server Modules.	249
Step 5 — Update Your Object Subprograms with Existing Business Rules.	251
Considerations for Implementing Business Rules	251
Step 6 — Set Up and Run Your Construct Spectrum Project	252
8. UNDERSTANDING AND CUSTOMIZING THE CLIENT FRAMEWORK	
Introduction to the Client Framework	254
About Box	257
Customizing the About Box	258
Application Preferences	260
Application Settings	262
Customizing the Application Settings.	262
Browse Support	265
Internationalization Support.	267
Maintenance Classes	268
Grid Support	268
Menu and Toolbar Support	270
Understanding Menu and Toolbar Command Handling	271
Class Summary.	272

Defining, Sending, and Handling Commands	273
Step 1 — Declare a Global Instance of the UICommands Class	273
Step 2 — Define the Commands	274
Step 3 — Code Menu and Toolbar Events to Send the Commands	274
Step 4 — Code the Command Handlers	276
Step 5 — Link the Commands to the Command Handlers	277
Updating User Interface Controls	278
Step 1 — Code Events to Update the Menu Controls	279
Step 2 — Code the Logic that Determines the State of a Command	281
Step 3 — Code Events to Update the Toolbar Buttons	282
Displaying a Disabled Bitmap	282
Displaying a Message	283
Update Cycles	284
Additional Methods For Command Handling	288
Unhooking Commands	288
Customizing the Menu and Toolbar in the Client Framework	289
Changing the Menu Structure	289
Example of Changing the Menu Bar and Its Menu Items	294
Changing the Toolbar Structure	298
Example of Adding Buttons to the Toolbar	299
Multiple-Document Interface (MDI) Frame Form	301
Object Factory	302
Understanding the Open Dialog	303
Understanding the Object Factory	304
Using the Object Factory	305
Example of Using the Object Factory	306
Customizing the Object Factory	307
Setting Up	307
Making your Application Aware of New Business Objects	311
Step 1 — Update the InitializeOpenDialog Procedure	312
Step 2 — Update the CreateForm Procedure	313
Step 3 — Update the GetBrowser Procedure	315
Step 4 — Update the BrowserExists Procedure	317
Spectrum Dispatch Client Support	319
Logon Dialog	321
Error Messages	322

Dispatcher Selection Dialog	322
Utility Procedures	323
9. VALIDATING YOUR DATA	
Overview	328
Basic Data Type Validation	328
Business Data Type Validation	328
Local Business Validation	328
Business Object Validation	329
Client Validation	331
Validation in Maintenance Dialogs	333
Using BDTs	333
Hand-Coded Validations in Generated Dialogs	333
Validation in Visual Basic Maintenance Objects	334
Adding Validations in the CLIENT-VALIDATION User Exit	334
Validations from Predict	335
Creating Verification Rules in Predict	336
Deciding Where To Implement a Validation Rule	336
Coding User Type Rules	337
Order of Precedence in Data Validation	339
Validation Error Handling	340
Framework Components	340
Handling Business Object Validation Errors	341
10. UNDERSTANDING THE BROWSE AND MAINTENANCE INTEGRATION	
Overview	344
Drill-Down Capabilities from a Browse Dialog	344
Active Help on Maintenance Dialogs	345
Primary Key Field Active Help	345
Foreign Field Active Help	346
Design Objectives	349
Application Component Independence	349
Simplified Generated Components	350
Overview of Foreign Key Field Relationships	351
Fields that can be Used in a Foreign Key Relationship	351

Simple Field	351
Repeating Field	352
When Not to Use a Foreign Field Relationship	353
List of Values is Static	353
List of Values is Small	353
List of Values Contains Two Choices Only	354
Foreign Field Support Provided By Maintenance Dialogs	355
GUI Control Representations of Foreign Fields	355
Foreign Fields On the Primary Part of a Maintenance Dialog	355
GUI Controls in a Grid	357
How Construct Spectrum Determines Which GUI Control to Use	358
Displaying Descriptions for a Foreign Field	359
Examples of Descriptive Fields	360
Supporting Multiple Descriptive Values and Derived Values	360
How Foreign Field Descriptions Are Refreshed	361
Supporting Code for Drop-Down lists	363
Initializing a Drop-Down List	363
Support for Value Selection	364
Supporting Code for Command Buttons	364
Initializing a Command Button	364
Click Events on the Command Button	365

11. INTERNATIONALIZING YOUR APPLICATION

Planning Your Internationalized Application	368
Internationalizing Using the Client Framework	369
Resource File Syntax	372
Text Values	372
Binary Values	373
Links	373
Using the Client Framework's Internationalization Components	374
Methods	374
GetResourceGroup	374
LocalizeForm	375
LoadBinaryResource	375
LoadStringResource	376
Message	376

MessageEx	376
SetDefaultMessageGroup	377
Properties	377
Language.....	377
LanguageRegistryKey	378
LanguageINIKey	379
ResourceFilePath	379
Hints for Developers.....	380
Automatically Setting the Language.....	380
Strategy for Using Resource Files and Groups.....	380
Starting an Application in a Specific Language.....	381
Associating Windows Locale Setting with a Language	383
Changing Language at Runtime	383
APPENDIX: MODIFYING SPECTRUM MODELS.....	385
VB-Maint-Dialog Model.....	386
VB API	388
Components of the VB API	388
How the VB API Works	389
GUI Controls with VB API	391
Parameter Data Area (PDA) Used	397

PREFACE

Developing Client/Server Applications is designed to help developers create and customize applications using the Construct Spectrum software development kit (SDK) and Visual Basic.

This preface will help you get the most out of the guide and find other sources of information about creating Construct Spectrum applications.

The following topics are covered:

- **Prerequisite Knowledge**, page 16
- **How to Use this Guide**, page 17
- **Conventions Used in this Guide**, page 19
- **Related Documentation**, page 20
- **Year 2000 Considerations**, page 22

Prerequisite Knowledge

Developing Client/Server Applications does not provide information about the following topics. We assume that you are either familiar with these topics or have access to other sources of information about them.

- Natural Construct
- Microsoft[®] Visual Basic[®]
- Predict[®]
- Natural[®] programming language and environment
- Entire Broker[™]
- Entire Net-Work[®]

How to Use this Guide

Developing Client/Server Applications describes how to create and customize client/server applications using Construct Spectrum SDK and Visual Basic. In particular, it provides information about:

- Creating new client/server applications
- Moving existing server-based applications to a client/server architecture

The following sections explain how to use this and related Construct Spectrum guides to perform these tasks.

If You are Creating a New Client/Server Application

If you want to use Construct Spectrum's tools to create a client/server application to run on Windows® 95 or Windows NT, we recommend that you first read the following chapters in *Construct Spectrum Programmer's Guide*:

- **Chapter 1, Introduction**
Contains an overview of the product, development process, and applications you can develop.
- **Chapter 2, Setting Up Your Application Environment on the Mainframe**
Contains detailed information on how to define domains and security options that control what data application users can access on the mainframe.

Developing Client/Server Applications contains detailed information on using the VB-Client-Server super model to generate all of your application's components. It explains how to set up a Visual Basic project, customize maintenance and browse dialogs, and internationalize your application.

As you customize and regenerate application components, you will find the following chapters in *Construct Spectrum Programmer's Guide* useful:

- **Chapter 6, Using the Subprogram Proxy Model**
- **Chapter 7, Using Business Data Types**
- **Chapter 8, Debugging Your Client/Server Application**
- **Chapter 9, Deploying Your Client/Server Application**

If You are Moving an Existing Application to a Client/Server Architecture

Before moving any existing server-based applications to the Construct Spectrum client/server architecture, we recommend that you gain familiarity with Construct Spectrum by creating a new application. The best way to become familiar with Construct Spectrum is described in **If You are Creating a New Client/Server Application**, page 17.

To learn how to migrate your existing server-based applications to the Construct Spectrum client/server architecture, refer to **Moving Existing Applications to Construct Spectrum**, page 245.

Conventions Used in this Guide

This guide uses the following typographical conventions:

Example	Description
Introduction	Bold text in cross references indicates chapter and section titles.
“A”	Quotation marks indicate values you must enter.
Browse model, GotFocus, Enter	Mixed case text indicates names of: <ul style="list-style-type: none"> • Natural Construct and Construct Spectrum editors, fields, files, functions, models, panels, parameters, subsystems, variables, and dialogs • Visual Basic classes, constants, controls, dialogs, events, files, menus, methods, properties, and variables • Keys
Alt+F1	A plus sign (+) between two key names indicates that you must press the keys together to invoke a function. For example, Ctrl+S means hold down the Ctrl key while pressing the S key.
CHANGE-HISTORY	Upper case text indicates the names of Natural command keywords, command operands, data areas, help routines, libraries, members, parameters, programs, statements, subprograms, subroutines, user exits, and utilities.
<i>Construct Spectrum Administrator's Guide, variable name</i>	Italicized text indicates: <ul style="list-style-type: none"> • Book titles • Place holders for information you must supply
[variable]	In syntax and code examples, values within square brackets indicate optional items.
{WHILE UNTIL}	In syntax examples, values within brace brackets indicate a choice between two or more items; each item is separated by a vertical bar ().

Related Documentation

The documentation sets for Construct Spectrum and Natural Construct consist of the following manuals:

Construct Spectrum SDK

- *Construct Spectrum Programmer's Guide*
This guide is for developers creating Natural modules and ActiveX Business Objects to support applications that will run in the Natural mainframe environment and a Windows environment and/or an internet server.
- *Developing Web Applications*
This guide is for developers creating the web components of applications. It explains how to use the Construct Spectrum wizards in Visual Basic to generate HTML templates, page handlers, and object factory entries. It also contains detailed information about customizing, debugging, deploying, and securing web applications.
- *Construct Spectrum Reference Manual*
This manual is for application developers and administrators who need quick access to information about Construct Spectrum application programming interfaces (APIs) and utilities.
- *Construct Spectrum Messages*
This manual is for application developers, application administrators, and system administrators who want to investigate messages returned by Construct Spectrum run-time and SDK components.

Construct Spectrum

- *Construct Spectrum Client Installation*
This manual describes how to install and set up the Construct Spectrum runtime and SDK components on the client.
- *Construct Spectrum Mainframe Installation*
This manual describes how to install and set up the Construct Spectrum runtime and SDK components on the mainframe.
- *Construct Spectrum Administrator's Guide*
This guide is for administrators. It describes how to use the Construct Spectrum Administration subsystem to set up and manage Construct Spectrum applications.

Natural Construct

- *Natural Construct Installation and Operations Manual for Mainframes*
This manual provides essential information for setting up the latest version of Natural Construct, which is needed to operate the Construct Spectrum programming environment.
- *Natural Construct Generation User's Manual*
This manual describes how to use the Natural Construct models to generate applications that will run in a mainframe environment.
- *Natural Construct Administration and Modeling User's Manual*
This manual describes how to use the Administration subsystem of Natural Construct and how to create new models.
- *Natural Construct Help Text User's Manual*
This manual describes how to create online help for applications that run on server platforms.

Year 2000 Considerations

We strongly recommend that you work with the preferred Y2K-capable date format of alphanumeric eight (A8) and numeric eight (N8). Although you can use alphanumeric six (A6) and numeric six (N6) date formats for some functionality, full functionality, inclusive of Y2K capability, is contingent upon the use of the A8 and N8 date formats. A8, A6, N8 and N6 formats are used in examples throughout the documentation.

Year 2000 capability as it relates to Natural Construct, Construct Spectrum, and Construct Spectrum SDK requires the use of the A8 and N8 date formats.

With regard to products or services, SAGA SOFTWARE defines Year 2000 “readiness” or “capability”, or the fact that a product or service is Year 2000 “ready” or “capable”, to mean that the product or service is capable of accurately processing, providing, and receiving data from, into, and between the 20th and 21st centuries, and that it will correctly create, store, process, and output information related to or including dates on or after January 1, 2000, provided that all other products, including hardware or software, used in combination with the product or service, properly exchange date information with the product or service.

INTRODUCTION

This chapter describes the components of Construct Spectrum and the architecture of the client/server applications you can create with the software development kit (SDK). An overview of the steps involved in developing an application prepares you for the detailed procedures in the chapters that follow.

The following topics are covered:

- **What is Construct Spectrum?**, page 24
- **Architecture of a Construct Spectrum Client/Server Application**, page 28
- **The Development Process**, page 33

What is Construct Spectrum?

Construct Spectrum comprises a set of middleware and framework components, as well as integrated tools, that use the specifications you supply to generate all the components of distributed applications.

Construct Spectrum works with other products in the following partnership:

- Natural is an open server that provides access to databases such as Adabas, DB2, and VSAM
- Predict provides a comprehensive repository
- Entire Broker provides message-oriented communication

You define and manage data and business rules for your application in a repository managed by Predict. Using Natural Construct, you can then generate the Natural modules that process data. Using Construct Spectrum SDK, you can also generate the Visual Basic client code and download the appropriate components to the client. You define the security privileges in the Construct Spectrum Administration subsystem and then deploy the application.

Construct Spectrum includes two components for delivering the performance and security that mission-critical applications require:

- Spectrum Dispatch Client (SDC) on the client
- Spectrum dispatch service on the mainframe server

When the client makes a communication request, the SDC translates the request into a compact, secure message and transmits it to the server via Entire Broker. On the server, the Spectrum dispatch service converts the incoming request for processing by the server application while enforcing multi-level security. Construct Spectrum then uses a similar technique to return the processed result to the client.

This guide describes how to generate and customize client/server applications using the Construct Spectrum SDK. Refer to *Construct Spectrum Programmer's Guide* for information about:

- Setting up your application environment on the mainframe
- Using business data types (BDTs)
- Debugging and deploying your application
- Creating client/server applications without the Construct Spectrum

Development Environments

As you develop applications, you will be working in at least three environments: the Construct Spectrum Administration subsystem, Construct Windows interface, and Visual Basic (using the Construct Spectrum Add-In).

You manage system and application data for your applications in the Construct Spectrum Administration subsystem:

```
BS_MAIN ***** Construct Spectrum Administration Subsystem ***** CDLAYMN1
Jul 30 - Main Menu - 10:14 AM

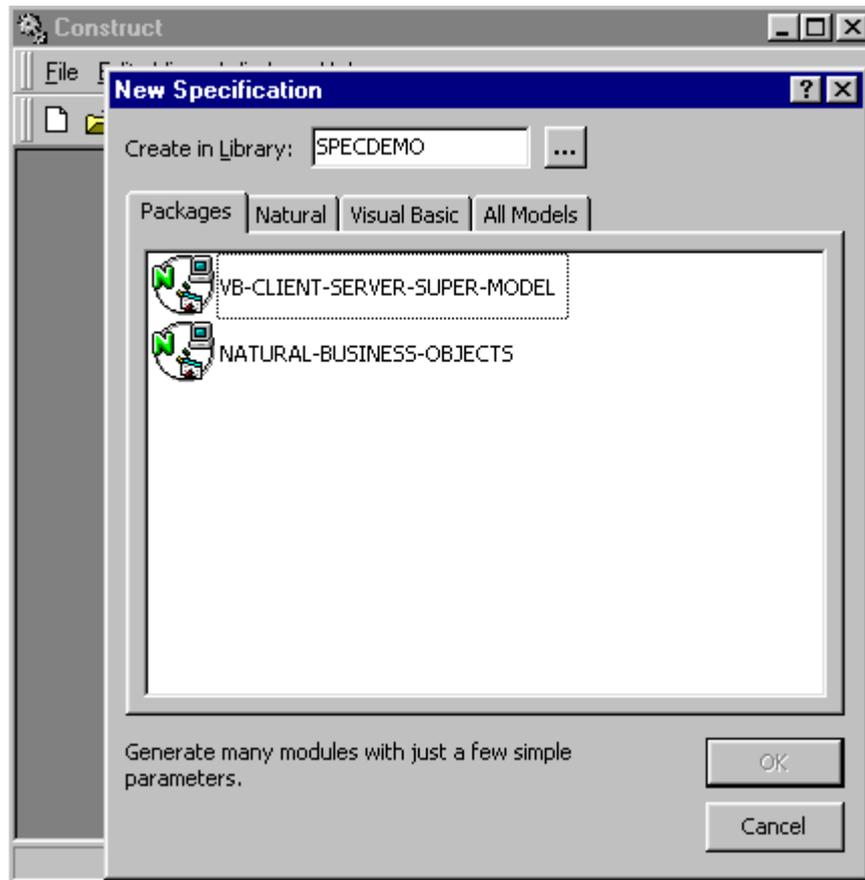
          Functions
          -----
          SA System Administration
          AA Application Administration

          ? Help
          . Terminate
          -----
Function ..... _

Command .....
Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      help retn quit          flip                               main
```

Construct Spectrum Administration Main Menu

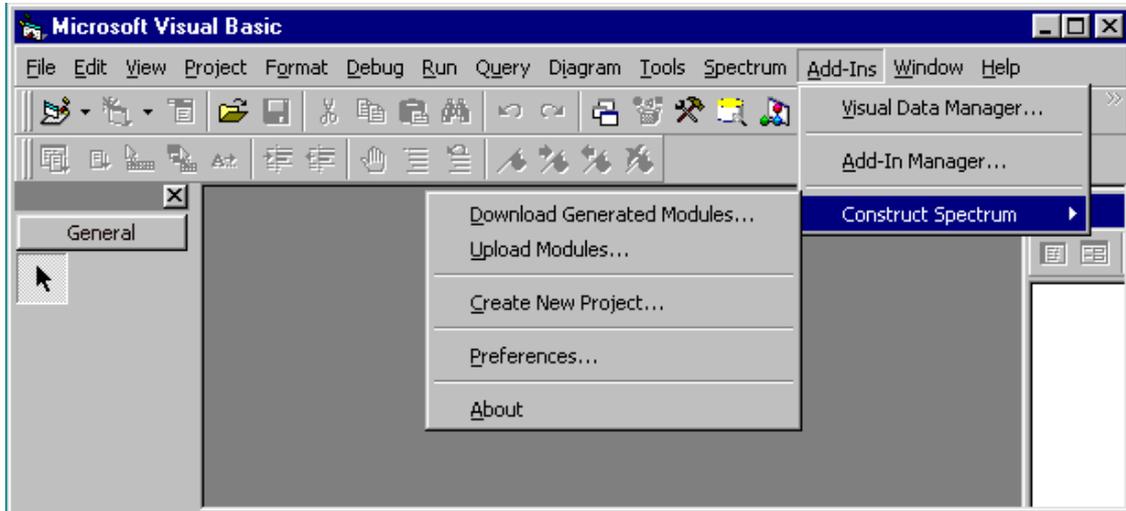
You use wizards to generate Natural and Visual Basic modules for your application in the Construct Windows interface on your PC:



New Specification Window in the Construct Windows Interface

The wizards available in the Construct Windows interface are also available in the Generation subsystem in your Natural Construct mainframe environment.

You use the Construct Spectrum Add-In in Visual Basic to create projects, download modules from the mainframe server, and set configuration options:

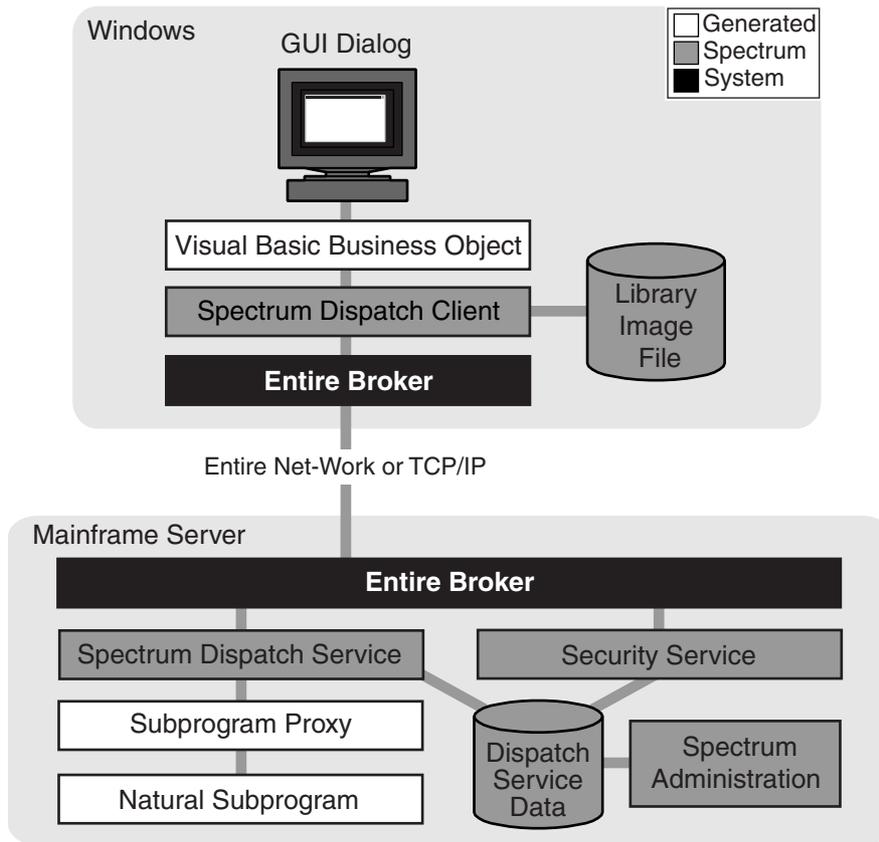


Construct Spectrum Options on the Add-Ins Menu

Information about how to access and use these environments is presented where you need it throughout the documentation.

Architecture of a Construct Spectrum Client/Server Application

Construct Spectrum generates high-performance, distributed components using COM-enabled clients to access Natural application servers. The following diagram shows the architecture of a Construct Spectrum client/server application:



Architecture of a Construct Spectrum Client/Server Application

The following sections explain these components according to the platforms on which the components run.

Mainframe Server

Component	Description
Natural subprograms	<p>Perform maintenance and browse functions on the mainframe server. The same set of business objects can be accessed from character-based Natural applications, client/server applications, and web applications. This ensures that the integrity of business data is preserved, independent of the presentation layer.</p> <p>Natural subprograms may be either written by hand or generated by Construct models. The VB-Client-Server-Super-Model, Object-Maint-Subprogram, and Object-Browse-Subprogram models generate subprograms and parameter data areas (PDAs) for client/server applications.</p>
Subprogram proxy	<p>Acts as a bridge between a specific subprogram and the Spectrum dispatch service. It performs a number of vital functions, including translating parameter data into a format that can be transmitted between client and server, issuing CALLNATs to subprograms, and validating the format and length of data received from the client.</p> <p>For more information, see Generating a Subprogram Proxy, page 132, in <i>Construct Spectrum Programmer's Guide</i>.</p>

Component	Description (continued)
Spectrum dispatch services	<p>Ensure that the current user is allowed to perform the requested function. Once the service has performed user authentication, it activates the correct Natural subprogram to handle the request. After the target subprogram finishes processing, the results are transferred back to the client. Depending on user options, the service may also be required to compress and decompress and/or encrypt and decrypt messages.</p> <p>For more information, see Defining and Managing Construct Spectrum Services, page 47, in <i>Construct Spectrum Administrator's Guide</i>.</p>
Dispatch service data	<p>Information defined and maintained in the Spectrum Administration subsystem and accessed by Spectrum dispatch services anywhere on the network via Entire Broker.</p>
Spectrum administration	<p>Allows system administrators, application administrators, and application developers to set up and manage system and application environments.</p> <p>For more information, see <i>Construct Spectrum Administrator's Guide</i>.</p>
Security service	<p>Checks client requests against the security settings defined in the Construct Spectrum Administration subsystem. This stand-alone service operates independently of the Spectrum dispatch services. This allows the security service to process, in one central location, the requests of several Spectrum dispatch services, which may be located on nodes throughout the network.</p> <p>For more information about security services and security settings, see <i>Construct Spectrum Administrator's Guide</i>.</p>
Entire Broker	<p>Transfers messages between Windows and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.</p>

Windows

Construct Spectrum client/server applications run on Windows or Windows NT.

Component	Description
Entire Broker	Transfers messages between the client and the Natural environment. Entire Broker can be configured to use either native TCP/IP or Entire Net-Work as the transport layer.
Spectrum Dispatch Client (SDC)	<p>Component Object Model (COM) middleware component that enables client/server applications to read from, and write to, variables in a Natural parameter data area (PDA) and to issue CALLNAT statements to Natural subprograms. Its main functions are simulating PDAs and CALLNATs, encapsulating Entire Broker calls, and controlling database transactions. As the client counterpart of Spectrum dispatch services, it is also responsible for data marshaling, encryption, compression, error-handling, and all Entire Broker communication.</p> <p>For more information, see Understanding The Spectrum Dispatch Client, page 243, in <i>Construct Spectrum Programmer's Guide</i>.</p>
Library image files	<p>Define information to the client component of a client/server application that it needs to assemble data and call the mainframe server. This file contains the following information:</p> <ul style="list-style-type: none"> • Parameter data area (PDA) definitions that specify information required for communication with the server. They are an image of the PDAs used by the Natural subprograms. • Application service definitions that specify to the client the names of the available subprograms. • Steplib definitions. The SDC allows chaining of library image files. The entries are used to point to other library image files in the same directory. The SDC checks all library image files in the chain for the required parameter or application service definition.

Component	Description (continued)
Visual Basic business object	Visual Basic class that acts as an intermediary between a dialog and the Spectrum Dispatch Client. This class invokes the methods of subprograms on behalf of dialogs and instantiates all the data areas required to communicate with a subprogram. Visual Basic business objects can also perform local data validation to provide immediate feedback to the user without involving a network call.
GUI dialogs	Represent graphical interface screens that communicate with the user and interact with the Visual Basic business objects and other framework components to implement business processes.

The Development Process

If you are creating a new application, you must decide on the nature of your application and its uses. If you are planning to reuse an existing application, evaluate the character screen displays and decide how to improve them using the power of a graphical user interface.

For more information about reusing existing Natural Construct applications, see **Moving Existing Applications to Construct Spectrum**, page 245.

- To develop an application:
- 1 Plan your application.
 - 2 Set up your application environment on the mainframe.
 - 3 Generate application components.
 - 4 Customize your application.
 - 5 Test and debug your application.
 - 6 Deploy your application.

The following sections describe these steps in detail.

Planning Your Application

Decide what the main purpose of your application is and what features you must provide to address it. After you determine the core features, consider the advanced functionality you may want to provide.

Decide What to Show the User

Before you begin creating a new application, decide on the purpose of your application, how it will be presented to the user, and how it will communicate with other applications. Decide what you want users to do with your application and determine what you need to provide in your application so that they can do it.

During the planning stages of your project, identify and itemize what the user will need to do. Then, design what the users will see when they use your application, such as the content, number, and order of windows in the application.

Plan to help your users, who will have varying degrees of experience with your client environment. Consider providing online help tailored for the level of knowledge of your typical user. You may choose to include all three types of online help:

- context-sensitive help
- task help
- window-level help

Provide customized error messages that are clear and informative. If you've planned your application, the chances of error are reduced. Since you cannot plan for every possibility, plan how your application will inform users about an action it cannot interpret. For example, you may want to display a message if a user tries to exit a file without saving the changes made during an edit session.

Note: Many error messages provided by Construct Spectrum will be available to your users. However, you must provide error messages for application-specific windows.

Design windows that are clear and intuitive. Try to give users all information they require to complete a task. Provide meaningful prompts and labels on GUI windows. To help minimize the amount of information your users need to provide, pre-set default values.

Keep Window Design Simple

When designing windows for your application, keep the window design simple. First determine the number and structure of windows, then determine the content.

Number and Structure of Windows

When designing the number and structure of windows, consider the following tips:

- Have one main window from which the user can initiate all of the main tasks.
- Provide secondary windows for additional information the user must specify to complete a task.
- Avoid a lot of nested windows, which can:
 - make a simple task look complex
 - clutter the user's screen (especially if the more than one application is open)
 - cause the user to become lost

Content of Each Window

When determining the content of each window, consider the following tips:

- Group related information together
- Use graphic images and icons to identify tasks or complement the words
- Position information in a neat, logical manner
- Position common information in the same place throughout your application. This makes it easier for your users to navigate.

Plan Your Code

After designing the application windows, decide what code is required to support actions users will perform with your application. When you generate your application, Construct Spectrum supplies many actions and default values. While several routine tasks are predefined and contain default attributes, you must explicitly set others for your application.

Use a Consistent Style

To help your users learn to navigate through your application, use consistent terminology. To help minimize confusion, use consistent mnemonics in all application windows.

Anticipate Translation Issues

When planning your application, consider whether the user interface will be translated into other languages. Construct Spectrum supplies translation facilities to support translation.

To minimize the effort required for translation, anticipate any issues when designing your application. For example, you may have to change mnemonic characters for different languages (if you are using mnemonics) or translation may change the size requirements for window text (such as text boxes, labels, and command buttons). Frequently, translated text is longer than the original text.

For more information, see **Internationalizing Your Application**, page 367.

Setting Up Your Application Environment on the Mainframe

Before you can create a Construct Spectrum application, ensure that Predict definitions, steplib chains, domains, users, groups, and security settings are defined.

Predict Definitions

Set up file and field definitions in Predict for all database applications generated using Construct Spectrum. This includes your application files and their intra- and inter-object relationships. For more information, see **Setting Up Predict Definitions**, page 47, in *Construct Spectrum Programmer's Guide*.

Steplib Chains and Domains

Define the steplib chains and domains for your applications. The application environment includes users, application libraries, business objects, and associated modules. Users are combined into larger entities called “groups”. Application libraries, business objects, and associated modules are combined into larger entities called “domains”. For more information, see **Define a Steplib Chain**, page 52, and **Define a Domain**, page 54, in *Construct Spectrum Programmer's Guide*.

Security for Domains, Steplibs, Users, and Groups

Define user IDs for users of your application, the groups to which each user belongs, and security privileges for each user. Then, assign users and security privileges for each group. Finally, grant groups applicable access to the domain for your application. Granting access to a domain enables users to access the objects and methods within the domain. For more information, see **Define Security for the Domain**, page 57, in *Construct Spectrum Programmer's Guide*.

Generating Application Components

After planning your application and setting up your environment, use the Construct Spectrum models to generate the application-specific components of your application. These components interact with the client framework components to form your complete application. To generate your application modules, use either the VB-Client-Server-Super-Model or the individual models.

Using the Super Model

Use the VB-Client-Server-Super-Model to quickly create a new application or add a graphical front-end to an existing application. For more information, see **Using the Super Model to Generate Applications**, page 93.

Using Individual Models

Use the individual models to fine-tune your application. Using individual models provides more opportunity to create unique model specifications. Additionally, you can add user exit code to further refine your application modules. For more information, see **Creating and Customizing Maintenance Dialogs**, page 137, and **Creating and Customizing Browse Dialogs**, page 215.

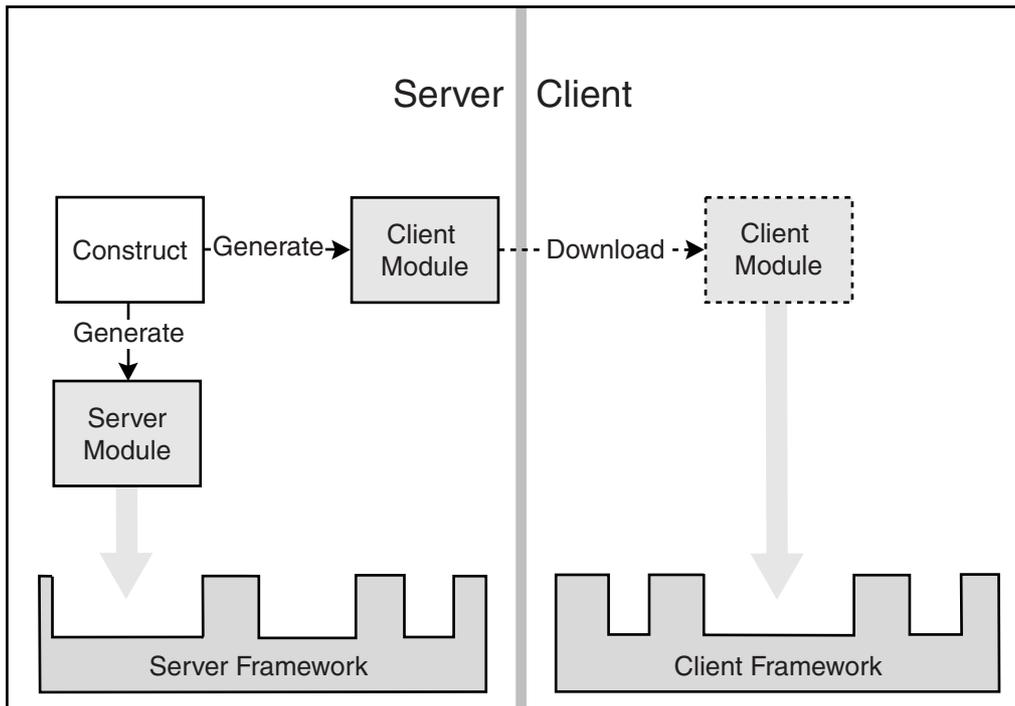
Deciding Which Modules to Generate

Regardless of how you generate your application modules, the same modules must exist to create a client/server application. These modules are grouped by function: maintenance or browse. To application users, these functions appear as either a window or dialog.

For a description of the modules that must be generated for either a maintenance or browse function, see **Using the Super Model to Generate Applications**, page 93.

Generation Process

The following diagram illustrates the process of generating application modules. The following sections describe how the server and client modules function.



Server Modules

Modules for the server portion of your application are generated in Natural, leveraging the existing Natural Construct object methodology. You can reuse existing Natural Construct modules generated using the Object-Maint-Subp or Object-Browse-Subp model as components of a client/server application.

For more information about moving existing applications, see **Moving Existing Applications to Construct Spectrum**, page 245.

Client Modules

Modules for the client portion of your application are also generated on the server. These modules are generated as Visual Basic code and stored as text members in the Natural library in which you generate them. When you are ready to set up your application on the client, use the Construct Spectrum Add-In to download the Visual Basic source code from the generation library to your client.

As you become more experienced in developing Construct Spectrum applications, you will want to create modules (or regenerate existing ones to add customizations) using individual models. The two types of objects you will create with Construct Spectrum are Visual Basic maintenance objects and Visual Basic browse objects.

You can access the models that generate application components either in the Generation subsystem on the server or in the Construct Windows interface. In both cases, modules are generated on the server.

For more information about using the Super model, see **Using the Super Model in the Generation Subsystem**, page 114.

For more information about generating with individual models, see:

- **Creating and Customizing Maintenance Dialogs**, page 137
- **Creating and Customizing Browse Dialogs**, page 215
- **Generating a Subprogram Proxy**, page 132, in *Construct Spectrum Programmer's Guide*

Setting Up Your Project

When you create a new project using the Construct Spectrum Add-In in Visual Basic, Construct Spectrum automatically adds the client framework components to a standard Visual Basic project. For more information, see **Creating a Construct Spectrum Project**, page 123.

Transferring Your Generated Code to the Project

Use the Construct Spectrum Add-In from the Visual Basic Add-Ins menu to download your generated components to the client. The components are added to your Construct Spectrum project, which includes the client framework components.

After integrating the generated components into your project, you can modify them and test your application. The following section describes this in more detail.

For more information about transferring your application to the client, see **Downloading the Generated Modules**, page 131.

Customizing Your Application and Environment

After creating your application, use Visual Basic on the client to tailor the user interface for your application.

For more information about customizing your application, see:

- **Understanding and Customizing the Client Framework**, page 253
- **Creating and Customizing Maintenance Dialogs**, page 137
- **Creating and Customizing Browse Dialogs**, page 215

Testing and Debugging Your Application

As your application becomes more stable, thoroughly test each component. In your test plan, include tests for each of the objects and their associated actions, each form, all local validations, and all remote methods.

While you can fix many errors you may encounter while creating your application on the client, you must fix others on the server. Construct Spectrum supplies methods that help track the origin and reason for errors. For more information, see **Debugging Your Client/Server Application**, page 201, in *Construct Spectrum Programmer's Guide*.

Once satisfied with the appearance and robustness of your application, you can begin to deploy your application for users. The following section describes how to make your application accessible to users.

Deploying Your Application

Deploy your Construct Spectrum applications in the same way as you deploy any Visual Basic application.

- To deploy your client/server application:
 - 1 Create the executable file.
 - 2 Collect the files to be installed.
 - 3 Create a set of installation disks.
 - 4 Install the client application on the user's PC.
 - 5 Run the application.

Note: To run the application, the Construct Spectrum runtime environment must be installed on the user's PC.

For more information, see **Deploying Your Client/Server Application**, page 237, in *Construct Spectrum Programmer's Guide*.

1

Developing Client/Server Applications

USING THE DEMO APPLICATION

This chapter provides a guided tour of a demo application created using Construct Spectrum. It also describes the underlying structure of the demo application. Use this chapter to familiarize yourself with the basic features available for client/server applications created with Construct Spectrum.

The following topics are covered:

- **Overview**, page 44
- **Setting Up Prerequisites**, page 45
- **Opening the Construct Spectrum Demo Project**, page 46
- **Running the Demo Application**, page 56
- **Using the Demo Application**, page 68
- **Troubleshooting**, page 91

Overview

The demo application is a Customer Order Maintenance program. This application is designed to demonstrate the features and functions of a typical application created with Construct Spectrum. As a demo application, certain “real-world” features, such as ensuring invoice numbers are sequential or order numbers are not duplicated, have been left out. You can add this type of application-specific checking when customizing your applications. Use the demo application to become familiar with using the application controls and components. Understanding the potential of Construct Spectrum is crucial to planning and developing an application that meets your needs.

Construct Spectrum is a flexible tool and your generated applications can be as simple or complex as you require. Additionally, you can implement features, such as a browse lookup, in many different ways. Therefore, you can give your applications a look and feel that is best suited to your organization’s needs.

Setting Up Prerequisites

Ensure the following items are in place before you begin generating applications using Construct Spectrum:

- ❑ Installation and configuration is complete.
Ensure that all client and server software has been installed as described in the *Construct Spectrum and SDK Client Installation Guide*.
- ❑ Entire Net-Work kernel is running on your PC (if you are using Entire Net-Work).
An Entire Net-Work kernel that enables communication between the client and server must be running on your PC.
- ❑ Your PC is attached to an Entire Broker node.
Your PC must be attached to an Entire Broker node that enables access to the demo database files and modules on the server. Use Spectrum Service Manager to configure the Entire Broker node.
- ❑ The demo project's AppSettings.bas file is set up correctly.
The AppSettings.bas file must specify the database ID (DBID) and file number (FNR) of the FUSER file in which you installed the Construct Spectrum demo application on the server. The default Natural library name for the demo application is SPECDEMO.

The AppSettings.bas file for the demo project is located in the same directory as your demo application files. You can modify this file using a text editor, such as the Windows Notepad editor.

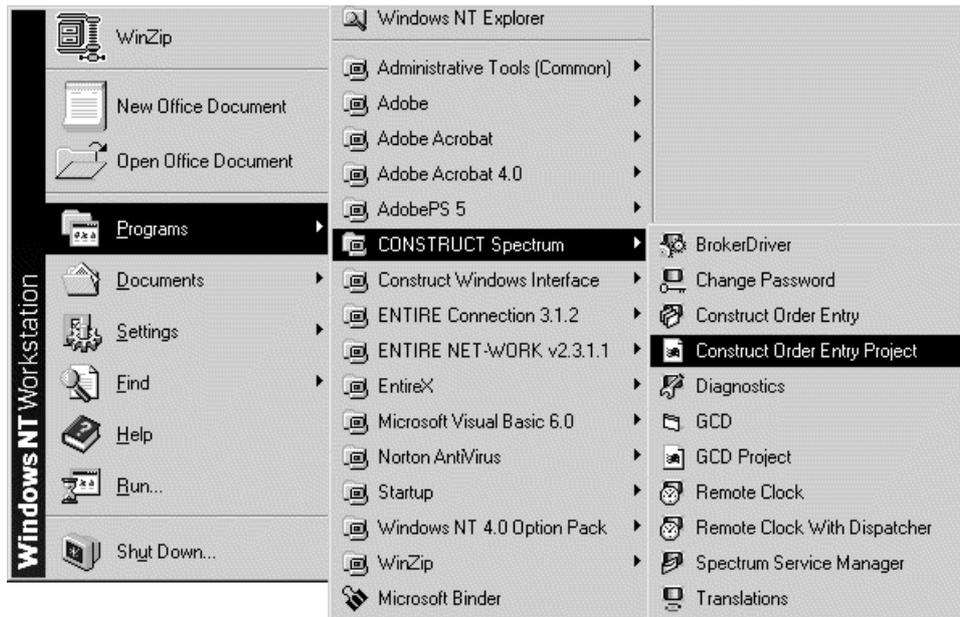
Consult with your system administrator to ensure that all of the listed prerequisites have been met before using the Construct Spectrum demo project.

Opening the Construct Spectrum Demo Project

This section describes how to open the Construct Spectrum demo project. A project is a container for all of the components required in the client portion of your application. All Construct Spectrum projects, including the demo project, are created using the Construct Spectrum Add-In in Visual Basic. Use this add-in to create the project and, if necessary, to download the required components from the server. For the demo application, these two steps are done for you. The demo project and the Construct Spectrum Add-In are described in more detail later in this section.

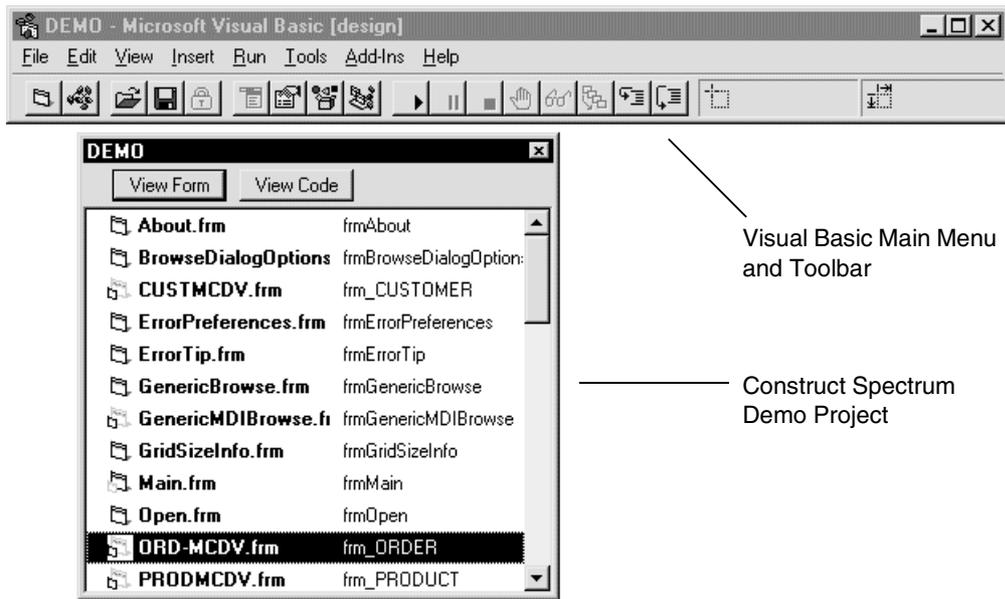
*Note: Ensure that all of the prerequisites described in **Setting Up Prerequisites**, page 45, have been met before opening the demo project.*

- To open the demo project:
 - 1 On the Start menu, click **Programs**.
 - 2 On the Programs submenu, click **Construct Spectrum** and then **Construct Order Entry Project Demo**, as shown in the following diagram:



Opening the Demo Application

The Construct Spectrum demo project opens. If Visual Basic is not running, it also opens. The project window contains references to all of the components required to compile and run your demo application:



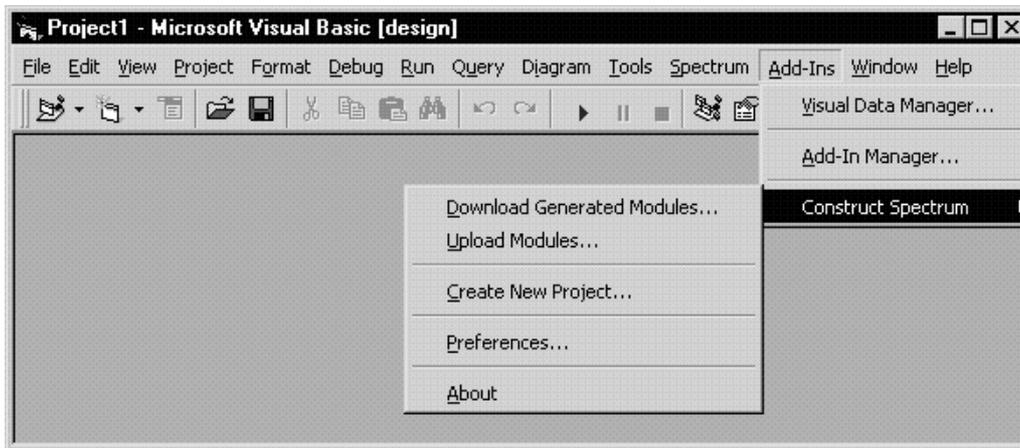
Construct Spectrum Demo Project

Tip: You can set up an icon or shortcut to open the Construct Spectrum demo project. For information, refer to your Windows help.

Once you have opened the project, you must run it to create a working application. This is described in **Running the Demo Application**, page 56.

Understanding the Construct Spectrum Add-In

Use the Construct Spectrum Add-In to manage the development of the client portion of your application. The add-in is available from the Add-Ins menu on the Visual Basic menu:



Construct Spectrum Add-In

Each Construct Spectrum Add-In option is described in the following table:

Add-In Option	Description
Download Generated Modules	Downloads generated modules from the server to your application project. For the demo application, this has already been done.
Upload Modules	Preserves user exit code that has been added on the client. For example, if you add user exit code to a Visual Basic maintenance object on the client, use this option to upload the business object module to the server so the code is preserved upon future regenerations of the business object.

Add-In Option	Description (continued)
Create New Project	Creates a project for your Construct Spectrum client/server application. For the demo application, this has already been done.
Preferences	Allows you to select a remote dispatch service. To allow access to the mainframe for downloading, enter your user ID and password in the appropriate fields.
About	Identifies the Construct Spectrum version level you are using and contains PC resource information, such as available memory.

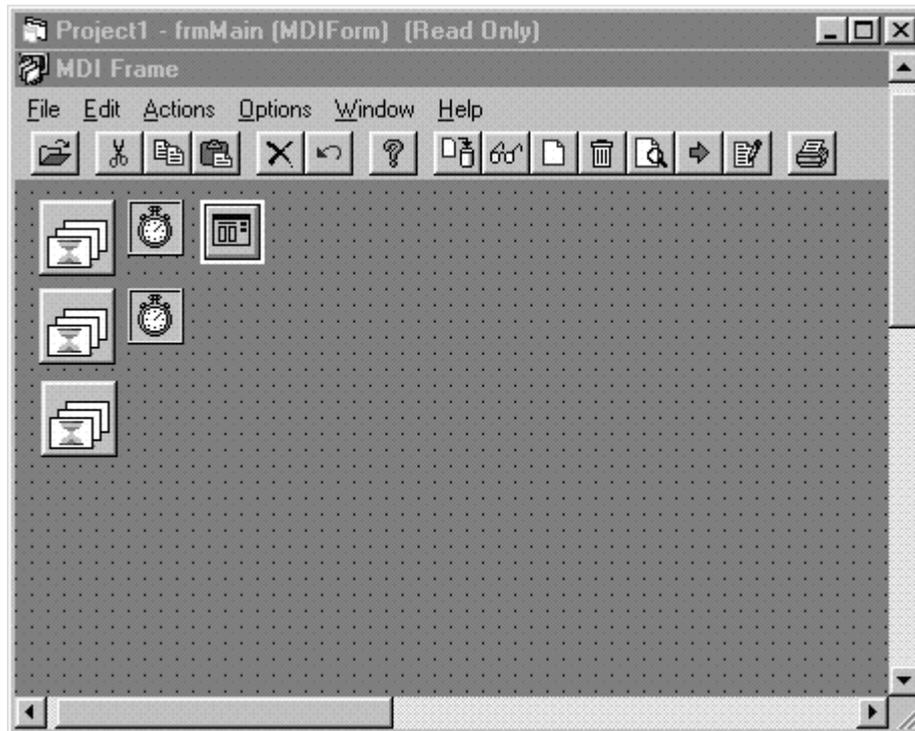
Understanding the Demo Project

The demo project contains all client components required to make a fully functional client/server application. The client components consist of framework components and generated modules. These are briefly described in the following sections. Also included in the following sections are diagrams showing both a framework component and a generated module as they appear before and after the project is run.

Framework Components

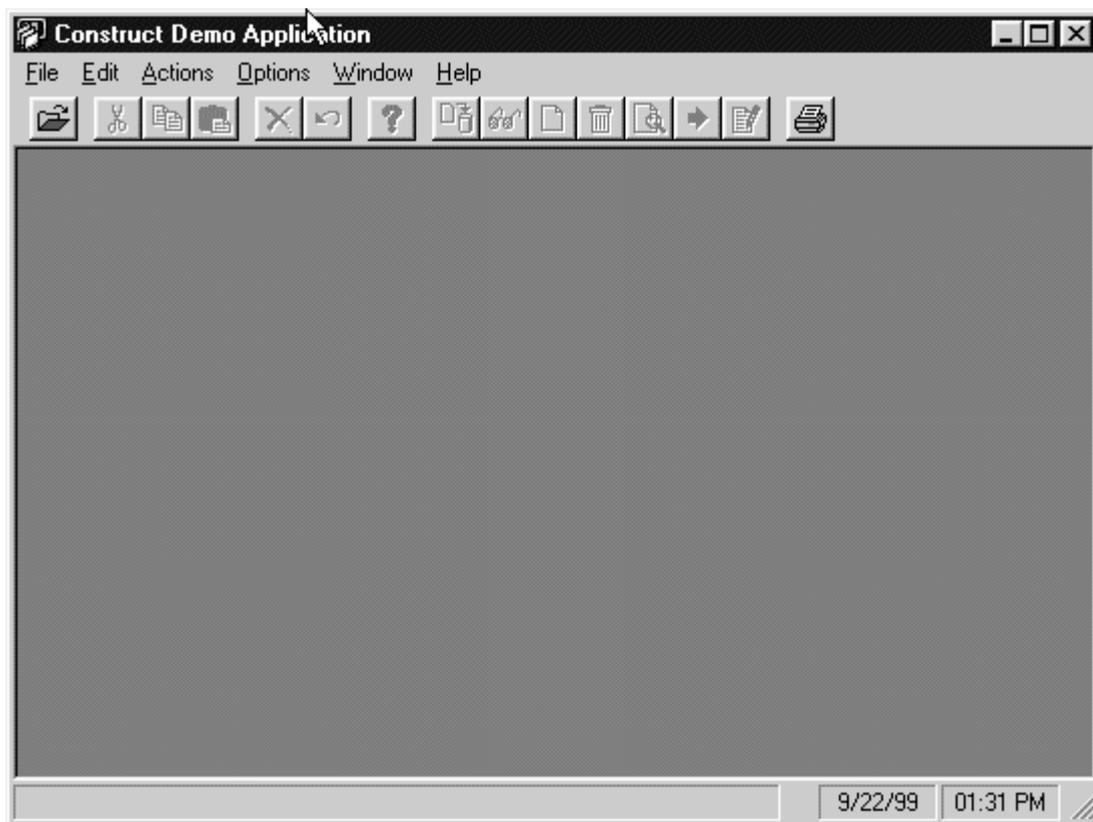
Framework components are reusable application components. These components provide a skeleton of functionality that interacts with generated and hand-coded Construct Spectrum modules to create a client/server application. When you create a project using the Create New Project option on the Construct Spectrum Add-In menu, framework components are automatically included in your project.

The diagram on the following page shows one of the framework components in your project: the Construct Spectrum Multiple Document Interface (MDI) frame.



MDI Frame Before Running Project

When you run the project to create your demo application, this frame looks similar to the diagram on the following page:



MDI Frame After Running Project

Use this window to access standard options, such as Open, Close, business objects and maintenance actions such as update, delete, move, next. For more information about framework components, see **Understanding and Customizing the Client Framework**, page 253.

Generated Modules

Generated modules are specific to your application. For example, the demo has a number of dialogs to maintain customer orders and products. Other generated modules include, but are not limited to, Visual Basic maintenance and browse objects, subprogram proxies, and PDA definitions. Generated modules are created on the server; those required on the client are downloaded to your Construct Spectrum project using the Download Generated Modules option on the Visual Basic Add-Ins menu.

The generated modules required for the demo have already been downloaded for you. The following diagram shows one of the generated components in your demo project: the Order Maintenance dialog.

The screenshot shows a Windows-style window titled "Project1 - frm_ORDER (Form)". Inside the window is a form titled "Order Maintenance". The form has a dotted background and contains the following fields and controls:

- Order Number:
- Order Amount:
- Order Date:
- Customer Number:
- Warehouse Id:
- Invoice Number:
- Delivery Instructions:
- Product:

*		

At the bottom of the form is a large, empty rectangular area. The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

Order Maintenance Form Before Running the Project

When you run the project to create your demo application, the dialog looks similar to the following diagram:

The screenshot shows a window titled "Construct Demo Application" with a menu bar (File, Edit, Actions, Options, Window, Help) and a toolbar. The main area is titled "Order Maintenance (123)" and contains the following fields:

- Order Number: 123
- Order Amount: \$300.00
- Order Date: 9/16/99
- Customer Number: 123
- Warehouse Id: A02 (Warehouse A02)
- Invoice Number: 34
- Delivery Instructions: Confirm delivery with Jack Fendler

Below the fields is a "Product:" section with a table:

	Product Id	Line Description	Quantity	Unit Cost
1	12	Stainless steel railings	10	
2				
3				
4				
5				

At the bottom right of the window, the date and time are displayed as 9/22/99 02:16 PM.

Order Maintenance Form After Running the Project

Using this dialog, you can maintain customer order information for your demo application. For more information about generated components, see **The Development Process**, page 33.

Running the Demo Application

This section describes how to run the demo application to add, delete, and update records to your Customer Order demo application. Experiment to become familiar with the user interface and various features that you get with any Construct Spectrum application.

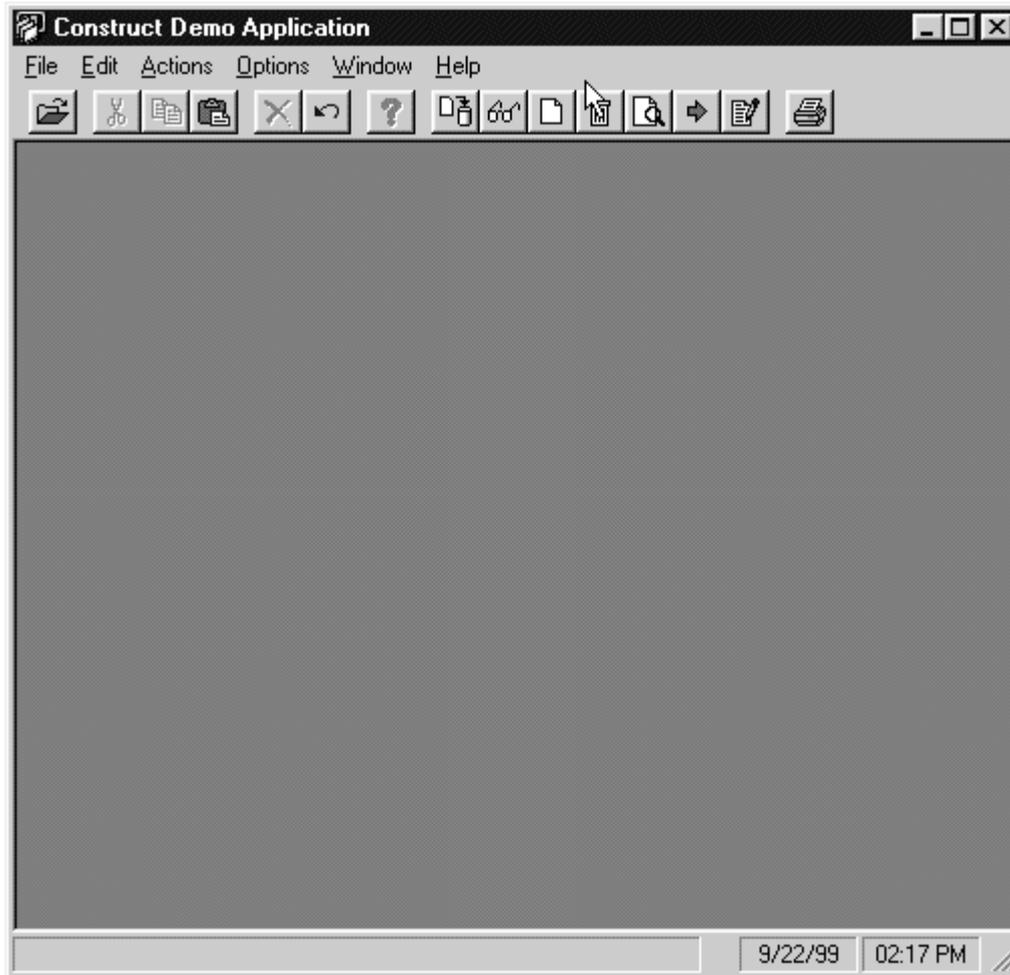
This section also contains information about some of the standard features that you get with every application developed with Construct Spectrum.

➤ To create the demo application:

- 1 Open the demo project as described in **Opening the Construct Spectrum Demo Project**, page 46.
- 2 On the Run menu, click **Start**.
- 3 Click **OK**.

Do not type a user ID or password in this dialog; the default user ID for the demo application is SYSTEM and there is no password.

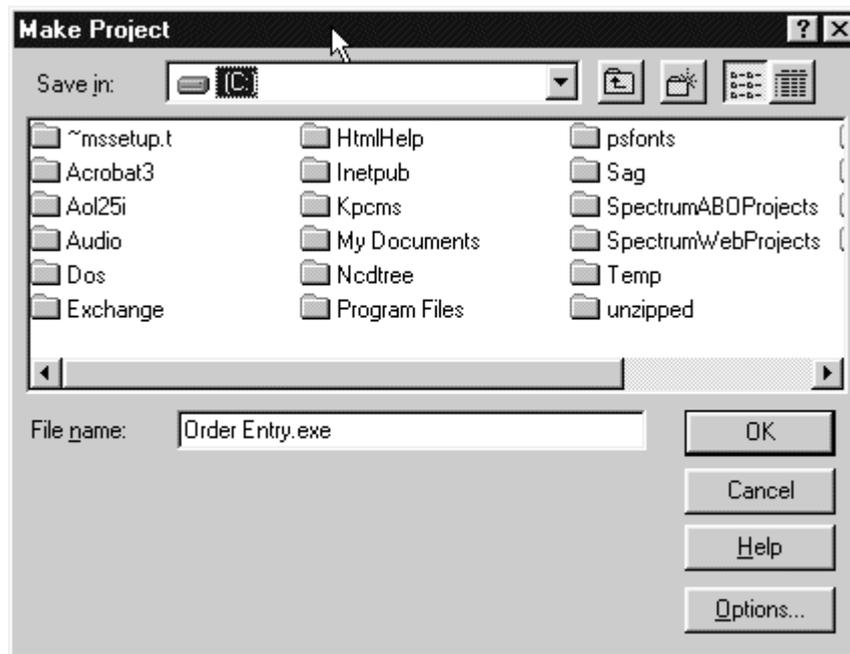
When the project successfully compiles, the MDI frame is displayed:



MDI Frame

You can use the demo application as long as the Visual Basic development environment is running. Steps 4 to 7 describe how to create an executable file from which you can use the demo application independent of the Visual Basic development environment.

- 4 On the Run menu, click **End**.
The MDI frame closes.
- 5 On the File menu, click **Make EXE File**.
The Make Project dialog is displayed:



Make Project Dialog

By default, the executable file (DEMO.exe) is saved in the ConstructOrderEntry directory in your Demo folder.

- 6 To save the executable file to another directory or with a different name, type new information in this dialog. When you are ready to replace the existing executable file, click **OK**.
The executable file is compiled and saved.
- 7 Locate and execute the file using the Run option on the Taskbar.
Alternatively, you can create a Windows shortcut to the file and double-click the shortcut icon.
When the Logon dialog is displayed, click **OK** to start the demo application.

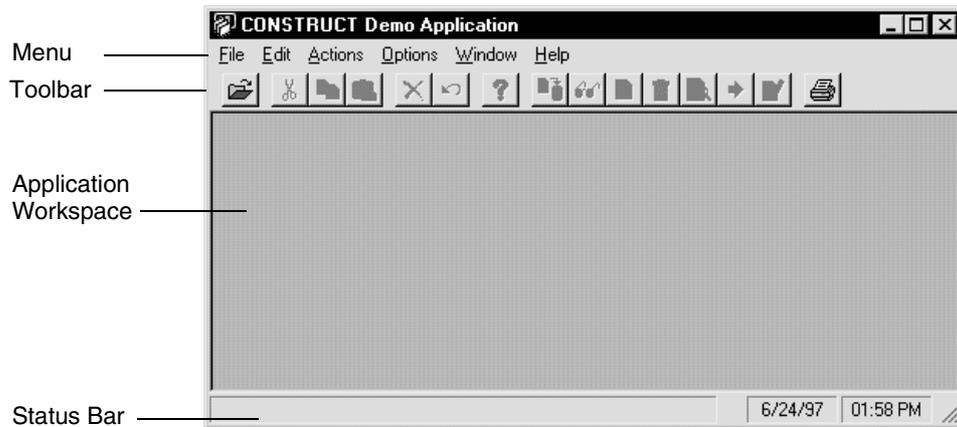
Application Interface

This section describes the user interface provided by default with all Construct Spectrum applications. The first dialog displayed when you start the demo is the Construct Spectrum Multiple Document Interface (MDI) frame. This is the workspace from which you manage your business objects, such as:

- Order object
- Customer object
- Product object
- Warehouse object
- Province object

Note: The Province object is a table in a Predict validation rule.

On the MDI frame, you can select an object for an action, such as to open it to maintain or browse records. The MDI frame consists of the components shown in the following diagram:



MDI Frame Dialog

Menu Options

The following table describes each menu option on the MDI Frame dialog

Menu Option	Description
File	Contains options to open or close a business object, log off, or exit the application.
Edit	Contains options to cut, copy, paste, undo or delete typing. Also contains options to add or delete rows of information; for example, when maintaining a customer order, you can add or delete rows of order information.
Actions	Contains methods for working with your application, for example, methods to add, delete, or get an object record. The methods available from this menu correspond to the methods associated with the business object.

Menu Option	Description (continued)
Options	<p>Contains notification options for handling errors when they are encountered. For example, when an error occurs, you can chose to be notified by a sound, an error message, or both.</p> <p>Also contains a Services option to select between different dispatch services. For an example, see Additional Options, page 65.</p>
Window	<p>Contains options to manage the windows that are currently open on your MDI frame. For example, you can move between open windows using this menu.</p>
Help	<p>Contains options to access help for your application. Also contains an About option from which you can display standard information about the application as well as standard system resource information.</p>

Toolbar Options

Toolbar button options are available for the most commonly used menu options. These are described in the following table.

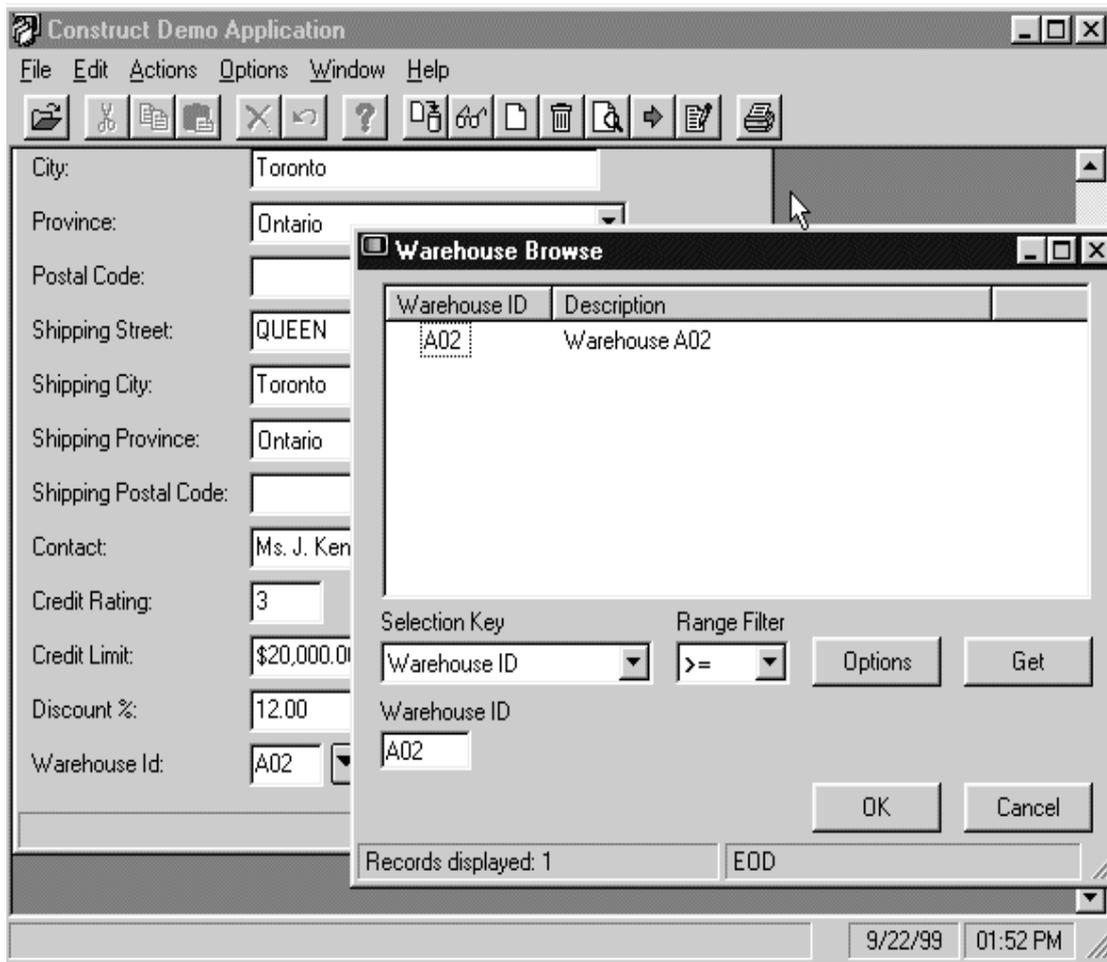
Note: To display the name of a toolbar button, place your cursor on the button for at least two seconds; a tooltip containing the name is displayed.

Toolbar Button	Description
	Displays the Open dialog, where you can select a business object and one of its associated actions for opening.
	Cuts the selection to the Windows Clipboard.
	Copies the selection to the Windows Clipboard.
	Pastes the selection to the Windows Clipboard.
	Deletes the selected characters.
	Undoes the last typing sequence you did; for example, if you delete a line of information using the Backspace key, clicking this button restores the line of information.
	Displays online help for Construct Spectrum.
	Adds a new record.
	Retrieves a listing of records from the server. You can select a record from the list to do some further action to it.
	Clears the currently displayed record from your desktop. If there are unsaved changes, you will be asked to save them; otherwise, changes will be lost.

Toolbar Button	Description (continued)
	Deletes the current record.
	Retrieves the specified record.
	Retrieves the next record. If there are unsaved changes to the currently displayed record, you will be asked to save them; otherwise, they will be lost.
	Updates the currently displayed record to the server database.
	Prints the selected object in the MDI frame.

Application Workspace

The Application workspace is where you work with your business objects. When you open one or more business objects, such as a customer order or a warehouse browse object, they are displayed on this workspace:



Open Documents on the Application Workspace

The MDI frame is a parent window to all business objects. You can manage your business objects through the MDI frame. For example, you can move between open objects using the MDI window menu commands. The previous diagram depicts a number of open objects on the application workspace.

Status Bar

The status bar displays messages and information about the current state of your application. For example, if you attempt an action that is not currently available, the status bar displays the following message:



Status Bar

Additional Options

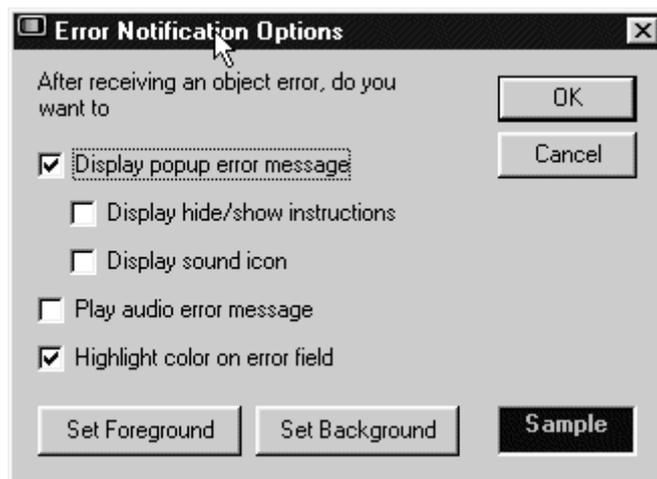
The following sections describe additional options available from the MDI frame:

- Error notification options
- Remote dispatch service options

Error Notification Options

Users can specify how they are to be notified when an error is encountered while using an application. For example, users can specify that the text box containing the error be highlighted and that information about the error be displayed immediately or only when the text box is selected.

- To modify error notification options:
- 1 Start the Demo.exe file created in **Running the Demo Application**, page 56. The Logon dialog is displayed.
 - 2 Click **OK**. The Construct Demo Application MDI frame is displayed.
 - 3 On the Options menu, click **Validation Errors**. The Error Notification dialog is displayed:



Error Notification Options

- 4 Select the check box(es) corresponding to the error notification options you want to enable.
- 5 If you selected the Highlight color on the error field option, choose the highlight colors by clicking the **Set Foreground** and **Set Background** buttons.

Later in this chapter, you will experiment with text box validations by entering incorrect values in a text box. At this point, try experimenting with your error notification options. Text box validations are described in **Validations**, page 73.

Remote Dispatch Service Options

Spectrum dispatch services can be set up for distinct units in your organization. For example, you could have one Spectrum dispatch service for your inventory control users and another one for your payroll users. Users who have been set up to access multiple Spectrum dispatch services do so by selecting the appropriate service from the MDI frame. For more information about Spectrum dispatch services, see the *Construct Spectrum Administrator's Guide*.

- To select a remote dispatch service:
- 1 Start the Demo.exe file that you created in **Running the Demo Application**, page 56.
 - 2 Click **OK**.
The Construct Demo Application MDI frame is displayed.
 - 3 On the Options menu, click **Service**.
The Select Remote Dispatch Service dialog is displayed:



Select Remote Service Dispatch Services Dialog

- 4 Select the Spectrum dispatch service you want to use.
Any open dialogs on the MDI frame are closed and you are prompted to save any unsaved changes.
You can now access the business objects available from the specified Spectrum dispatch service.

Using the Demo Application

This section describes many of the features and functions of the demo application by taking you on a guided tour of the customer order maintenance and browse functions. Some of the features and functions are provided by default with every application developed with Construct Spectrum, while others are provided based on the Predict setup of your application files and fields on the server. In each of the following sections, those features provided by default or provided based on your Predict set up are identified. For those features provided based on your Predict set up, you are also provided with information about the particular Predict set up that was required to make these features available.

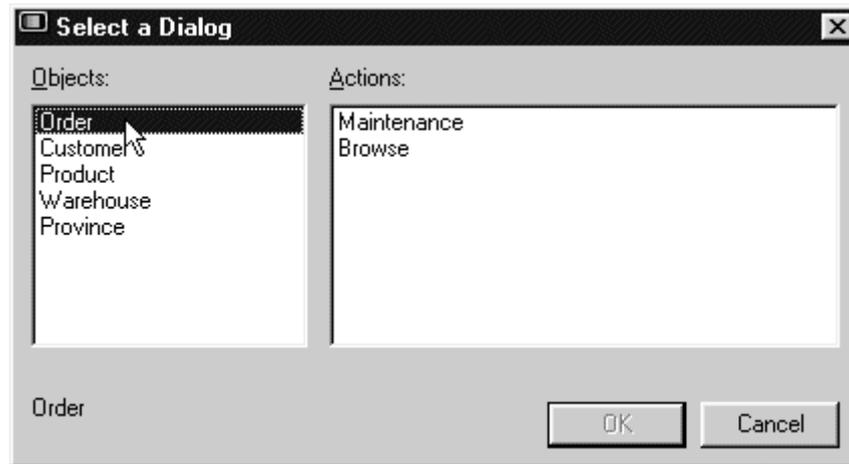
This section also contains a listing of the standard keyboard shortcuts available with all Construct Spectrum applications.

Your tour of the demo application involves working with customer orders. You will maintain and browse customer orders. As you do this, you will perform various tasks to give you an idea of what the application can do. You should be able to develop applications that are at least as functionally rich as the demo application. At this point, do not worry about the details of how things work, but try to get an understanding of what features you can provide in your own application.

Opening a Business Object

In this section, you will open a order business object. The order business object will be used to demonstrate most of the Construct Spectrum features described in the remainder of this chapter.

- To open a customer order business object:
 - 1 Start the Demo.exe file created in **Running the Demo Application**, page 56. The Logon dialog is displayed.
 - 2 Click **OK**.
The Construct Demo Application MDI frame is displayed.
 - 3 On the File menu, click **Open**.
Alternatively, you can click the Open toolbar button. The Select an Object/Action dialog is displayed:



Select an Object/Action

- 4 In the Object list box, click **Order**.
The available actions for the Order object are displayed in the Actions list box.
- 5 In the Actions list box, click **Maintenance** and then click **OK**.
Alternatively, you can double-click Maintenance. The Order Maintenance dialog is displayed:

Construct Demo Application

File Edit Actions Options Window Help

Order Maintenance

Order Number:

Order Amount:

Order Date:

Customer Number: ▼

Warehouse Id: ▼

Invoice Number:

Delivery Instructions:

Product:

	Product Id	Line Description	Quantity	Unit Cost	Total Cost
1					
2					
3					
4					
5					

	Cost Center	Acct	Project	Dist Amount
1				
2				

9/22/99 02:23 PM

Order Maintenance Dialog

Note: This procedure assumes that you are opening a business object to perform a maintenance function. To browse for a record or records, select Browse from the Actions list.

- 6 On the File menu, click **Next**.
Alternatively, you can click the Next toolbar button. The first customer order record is displayed.

Order Maintenance (512)

Order Number:

Order Amount:

Order Date:

Customer Number: ▼

Warehouse Id: ▼

Invoice Number:

Delivery Instructions:

Product:

	Product Id	Line Description	Quantity	Unit Cost	Total Cost
1	1111	Iron railings	24	12.00	288.00
2					
3					
4					
5					

Distribution (2):

	Cost Center	Acct	Project	Dist Amount
1	tc	231	5	120.00

9/23/99 02:12 PM

Order Maintenance Dialog With an Open Order

In the next section of this demo, you will learn about some of the standard features of the demo application by using an order object.

Maintaining a Business Object

This section demonstrates some of the standard features available to help users maintain their business objects. This section covers:

- Validations
- Business data types (BDTs)
- Grids

The features described in this section are demonstrated using the customer order object that you opened in **Opening a Business Object**, page 68. In addition to experimenting with the features described in this section, try adding, deleting and updating customer orders.

Validations

When a LostFocus event is triggered on a text box, it is validated. For example, when you type a value in a text box and tab to the next text box, a LostFocus event is triggered and the text box is validated. There are four types of validations that occur on the client:

- Basic data type
- Business data type
- Local business type
- Foreign field type

Basic data type validations verify that the format and length of an entered value is acceptable for the particular field.

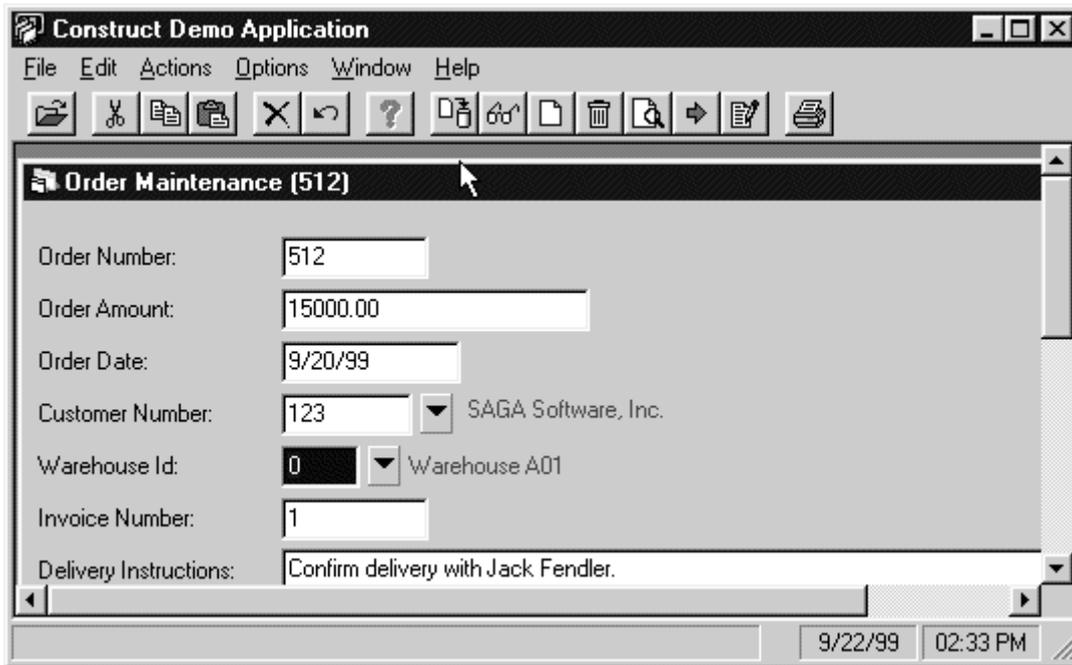
Business data type (BDT) validations ensure that data is formatted consistently and in a way that is easily understood. For example, if all dates in your organization should be formatted with forward slash (/) delimiters, you can assign a BDT to format such values. If a user enters a valid date without forward slash delimiters, the BDT formats the date when a LostFocus event occurs on the date text box. BDT validations are described in **Business Data Types (BDTs)**, page 75.

A local business validation performs more complex validations based on your business rules. For example, a local business validation can ensure that one of a finite set of valid values is allowed in the field, such as a valid province code. A more complex local business validation could calculate the provincial tax amount on an order based on the province code entered.

When a field on a maintenance dialog is a key field in a foreign file, Construct Spectrum generates code to validate the field using the foreign file. For example, the Order maintenance dialog in the demo application has a Warehouse ID text box which is a key field in a foreign file: the Warehouse file. When a LostFocus event is triggered on the Warehouse ID text box, Construct Spectrum verifies that the Warehouse ID entered is a valid ID.

For more information about basic data type, business data type, and local business type validations, see **Validating Your Data**, page 327. For more information about foreign field validations, see **Understanding the Browse and Maintenance Integration**, page 343.

- To test how a validation works:
 - 1 Open a Order object.
For information, see **Opening a Business Object**, page 68.
 - 2 Type an invalid warehouse ID in the **Warehouse ID** text box.
 - 3 On the Actions menu, click **Update**.
Alternatively, you can click the Update toolbar button. The Warehouse ID text box is highlighted and, depending on how your Error Notification options are set up, an error message is displayed. Or you can select the highlighted text box to display the message. For more information about Error Notification options, see **Error Notification Options**, page 65.



Validation in the Warehouse ID Text Box

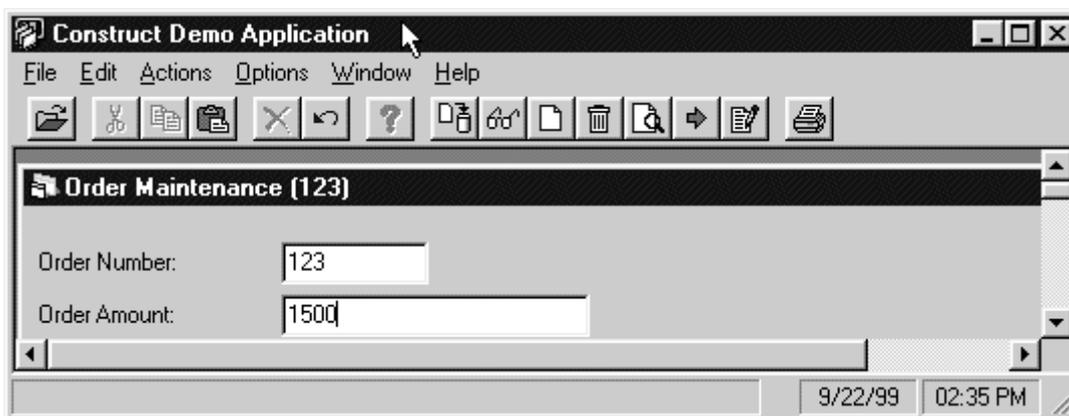
- 4 To correct the problem, type a valid warehouse ID in the text box (or select a valid warehouse from the drop-down list box).
- 5 On the Actions menu, click **Update**.
Alternatively, you can click the Update toolbar button.

Business Data Types (BDTs)

Business data types (BDTs) help to ensure that information is displayed in a way that is consistent and easy to understand. For example, a BDT can automatically reformat a telephone number that was entered without dashes or round a numeric value.

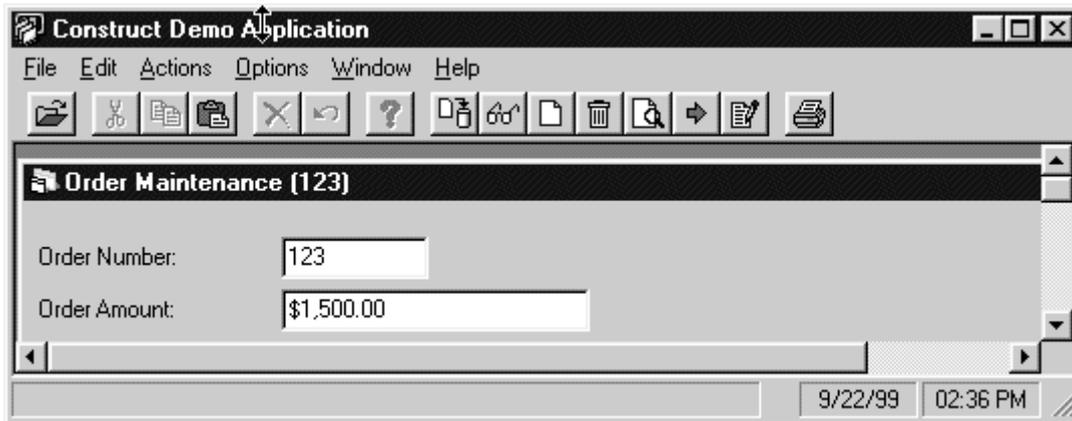
Construct Spectrum comes with a number of predefined BDTs you can customize and attach to any field based on your business requirements. When a user enters a value in the field, formatting is applied automatically when a lost focus event occurs (for example, when the user selects another field or option).

- To test how a BDT works:
- 1 Open a Customer Order object. If you need help opening a business object, see **Opening a Business Object**, page 68.
 - 2 Place your cursor in the **Order Amount** text box and type “1500” as shown in the following diagram:



Value Before BDT Formatting Occurs

- 3 Click outside the **Order Amount** text box.
The value you entered is formatted with a decimal and two trailing zeros:



Value After BDT Formatting Occurs

Enter an alphabetical character in the Order Amount text box to see what happens. In this case, the BDT for the Order Amount text box was set up to convert alphabetical characters to zeros. Another option could be to display an error if alphabetical characters are entered.

Grids

Grids display rows of related information about a business object. The Order object contains the Product grid, which displays the individual lines for a particular customer order. Each row corresponds to a separate order line. In the following diagram, there is one order line, Cat Nuggets, for the customer order:

	Product Id	Line Description	Quantity	Unit
1	187361	CAT NUGGETS	10	
2				
3				

Grid Showing Order Lines for a Customer Order

Experiment with the grid by adding and deleting additional order lines.

- To add an order line:
 - 1 Open a Customer Order object.
For information, see **Opening a Business Object**, page 68.
 - 2 Place your cursor on an empty order line and complete the cells.
Use the horizontal scroll bar to access additional information on the grid.
 - 3 On the Actions menu, click **Update**.
Alternatively, you can click the Update toolbar button.
- To add a new order line between two lines:
 - 1 Select the row immediately above the location where you want to add a new row.
 - 2 On the Edit menu, click **Insert Row**.
An empty row is added below the selected row.
- To delete an order line:
 - 1 Select the order line.
 - 2 On the Edit menu, click **Delete Row**.
The selected order line is deleted.

Grids can also be linked to nested grids and browse functions. Nested grids are described in **Nested Grids**, page 79. Using a browse from a grid is described in **Browsing For Business Object**, page 81.

Nested Grids

Nested grids show additional information related to a row or a single cell in a grid. The Order object has a nested grid containing the distribution information for each order line. The distribution grid is nested to each order line in the Product grid. In the following diagram, two lines of distribution have been set up for our order of Cat Nuggets.

	Cost Center	Acct	Project	Dist Amount
1	11	521	A4	1500.00
2	14	333	A4	75.00
3				
4				

Nested Grid Showing Distribution for an Order Line

Select the first order line in your order object and then another order line; notice that the distribution grid changes depending on the order line you select. This is because you can have multiple lines of distribution for each order line. To accomplish this, the distribution grid was set up as a nested grid.

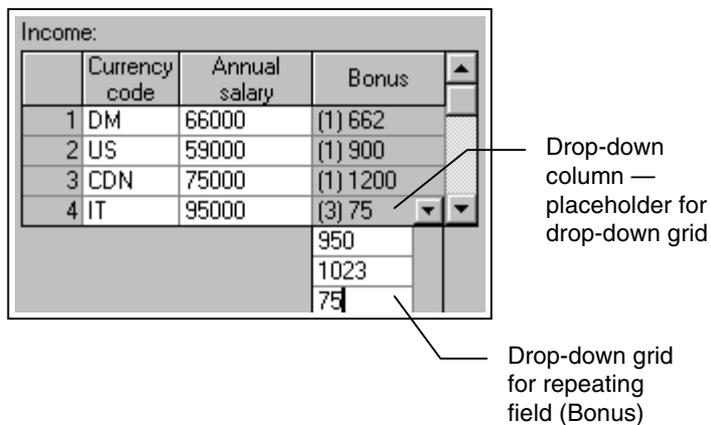
Nested Drop-Down Grids

You can set up a nested grid to drop-down for a cell within a grid. When a user selects the drop-down grid, additional information is displayed in a drop-down list box. For example, suppose you had a grid showing customer accounts and one of the cells in this grid showed the first of up to five lines of the customer's address. You could set up a nested grid containing the remaining lines of address information.

The demo application does not have a drop-down grid. The following procedure contains a diagram of a sample drop-down grid to show you what one looks like.

Tip: Cells containing drop-down grids are identified with gray shading and an occurrence number in brackets () for each repeating value in the grid.

- To display a drop-down grid:
- 1 Select the cell containing a drop-down grid.
A down arrow is displayed in the cell.
 - 2 Select the down arrow.
The drop-down grid is displayed:



Sample Drop-Down Grid

To learn more about working with drop-down grids, see **Keyboard Shortcuts for Grids**, page 81.

Keyboard Shortcuts for Grids

The first two keyboard shortcuts apply to a selected grid row. Select a grid row by highlighting the number to the left of the grid row. The remaining shortcuts apply only to nested drop-down grids.

Keystroke	Action
Del	Deletes the selected row of information from a grid. If the row has child grids, these are also deleted.
Ins	Inserts a blank row above the selected row. If the selected row has child grids, these are inserted as well.
Alt+Down Arrow	Displays the drop-down grid.
Alt+Up Arrow or Esc	Hides the drop-down grid.
Shift+Alt+Down Arrow	Displays the next value in a drop-down column without displaying the entire drop-down grid.
Shift+Alt+Up Arrow	Displays the previous value in a drop-down column without displaying the entire drop-down grid.

Browsing For Business Object

Browses enable you to search for and select records. For example, if you want to update an existing order but do not remember the order number, you can locate and select the order using the order browse. Construct Spectrum provides you with a number of methods to browse for a business object. Browsers can be initiated from a menu option or from a maintenance dialog. This section describes some of the ways users can browse for data, as well as some of the features available to customize a browse. This section covers:

- Selecting a business object with a browse
- Specifying browse customization options

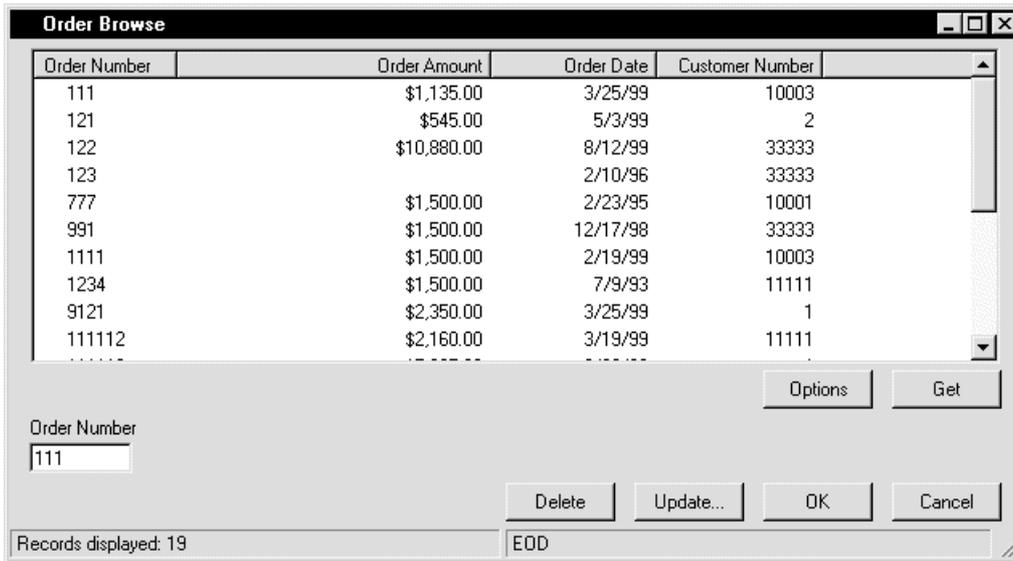
Selecting Data With a Browse

This section describes a number of ways to search for and open records with a browse. Browsers can be opened directly, from the **File** menu, or from a maintenance dialog.

Open a Business Object With a Browse

In the following procedure, you open an Order Browse dialog from the File menu.

- To open a browse from the File menu:
 - 1 On the File menu, click **Open**.
Alternatively, you can click the Open toolbar button. The Select an Object/Action dialog is displayed.
 - 2 In the Object list box, click **Orders**.
The available actions for the Order object are displayed in the Actions list box.
 - 3 In the Actions list box, click **Browse** and then **OK**.
Alternatively, you can double-click Browse. The Order Browse dialog is displayed:



Order Browse Dialog

- 4 Click **Get**.
A list of orders is displayed in the browse dialog.
- 5 Click an order.
- 6 Click **Update**.
The Order Maintenance dialog is displayed with the selected order.

The Update and Delete options shown in the previous browse dialog were created by adding update and delete commands to the demo application's command handler.

For information about adding these and other commands, see **Creating and Customizing Browse Dialogs**, page 215.

Open a Second Order to Work On

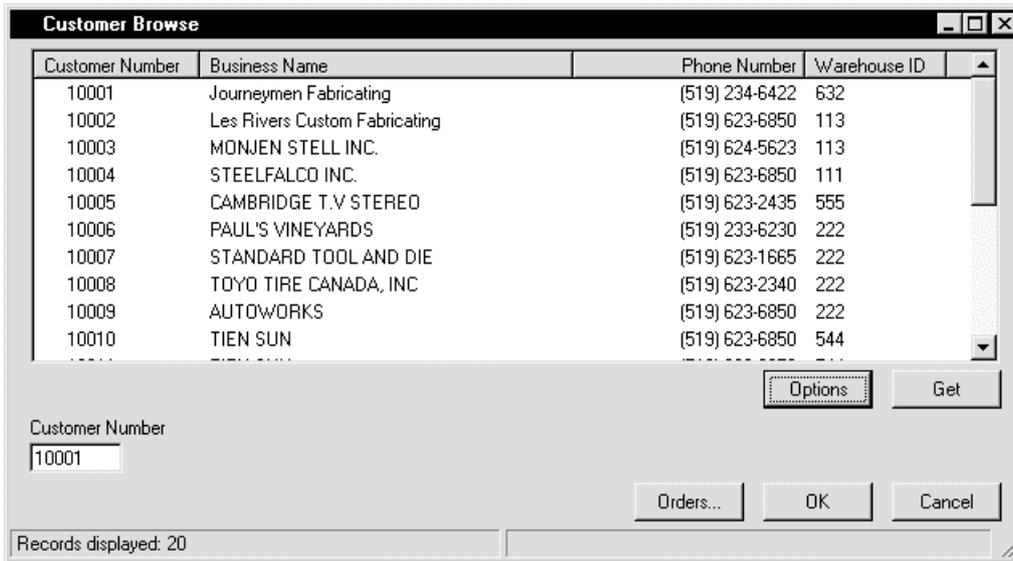
In this procedure, you will browse for and open a second order while the first business object is currently open.

- To browse for and open a second order:
 - 1 Open an Order object to perform a maintenance activity.
For information, see **Opening a Business Object**, page 68.
 - 2 On the Actions menu, click **Browse**.
Alternatively, you can click the Browse toolbar button. The Order Browse dialog is displayed.
 - 3 Click **Get**.
A list of customer orders is displayed in the browse dialog.
 - 4 Select an order (a different order than the one currently displayed on the Order Maintenance dialog).
 - 5 Click **OK**.
The selected order replaces the order currently displayed on the maintenance dialog.

Open Foreign File Information

When a maintenance dialog contains text box or grid information that is defined in a another file (foreign file), Construct Spectrum automatically adds a browse function to the foreign field or grid information. For example, the Order Maintenance dialog includes the Customer Number text box, which is defined in a foreign file: the Customer file. You can browse on this field to locate and select a customer number for your order. In the following procedure, you will open the Customer browse from the Order Maintenance dialog.

- To open a browse from the Order Maintenance dialog:
 - 1 Open the **Order maintenance** dialog.
 - 2 Select the Down arrow to the right of the **Customer Number** text box.
The Customer Browse dialog is displayed:



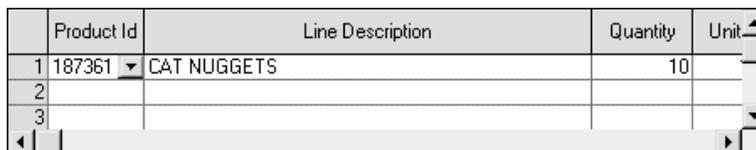
Customer Browse Dialog

- 3 Click **Get**.
A list of customer records is displayed in the Customer Browse dialog.
- 4 Select a customer number.
- 5 Click **OK**.
The selected customer number is displayed in the Customer Number text box on the Order Maintenance dialog.

The Order Maintenance dialog also has a browse linked to the Product grid. Use this browse to select a product.

➤ To open the Product browse from the Product grid:

- 1 Click a cell in the first column of the grid.
A Down arrow is displayed:



	Product Id	Line Description	Quantity	Unit
1	187361 ▾	CAT NUGGETS	10	
2				
3				

Grid with Down Arrow Displayed

- 2 Click the Down arrow.
The Product browse is displayed with a list of products.
- 3 Click a product.
- 4 Click **OK**.
The selected product is displayed in the Product grid on the Order maintenance dialog.

Specifying Browse Customization Options

Construct Spectrum browses include options that enable you to narrow your search criteria and to customize the information displayed on your browse dialog. The following topics are covered in this section:

- Specifying selection options
- Specifying display options

Specifying Selection Options

You can specify selection options in your browse to display as many or as few records as you want.

- To specify selection options:
- 1 Open the Order Browse dialog.
For information, see **Selecting Data With a Browse**, page 82.
 - 2 Click **Options**.
The Browse Options dialog is displayed with the Key Options tab selected:

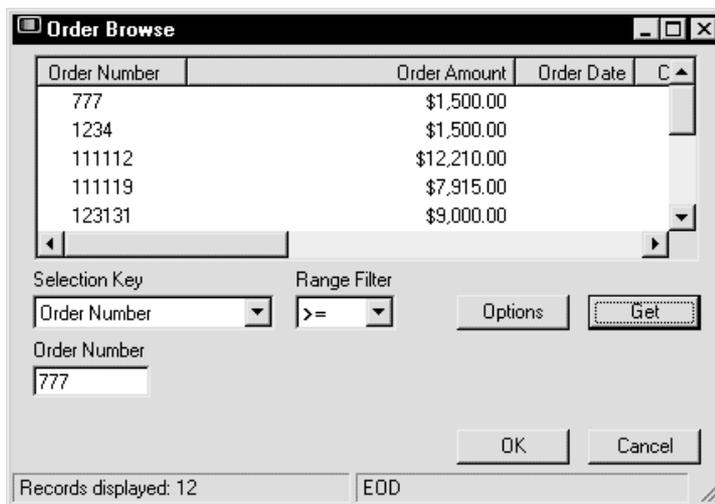


Browse Options - Key Options Tab

Ensure that the Show Selection Key and Show Range Options check boxes are selected, as shown in the previous diagram.

- 3 Click **OK**.
The Browse Options dialog closes and the Order Browse dialog is displayed.
- 4 Select **Customer Number** from the **Selection Key** drop-down list box.
- 5 Select the greater than symbol (>) from the **Range Filter** drop-down list box.

- 6 Type “777” in the **Order Number** text box.
- 7 Click **Get** (or press Enter).
The Order Browse dialog displays all customer order numbers greater than 777:



Order Browse

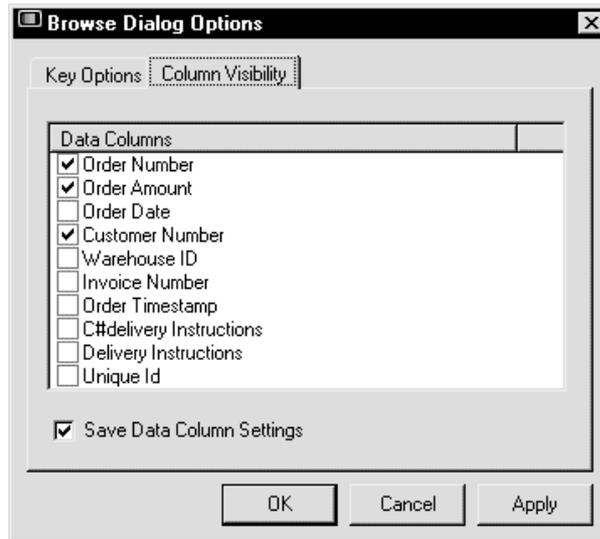
Specify your own selection options by experimenting with Selection Key, Range Filter, and Order Number.

Specifying Display Options

You can customize your browse dialog to show as many or few columns of browse information as required.

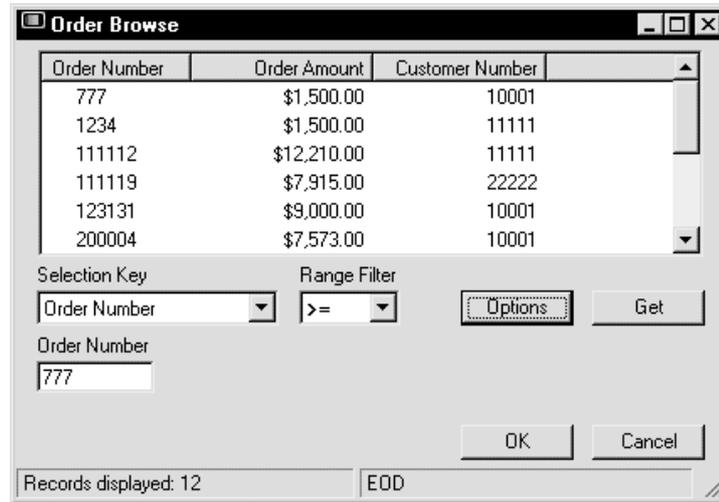
- To customize the display options on your browse:
 - 1 Open the **Order Browse** dialog.
For information, see **Selecting Data With a Browse**, page 82.
 - 2 Click **Options**.
The Browse Options dialog is displayed.

- 3 Select the **Column Visibility** tab:



Browse Options - Column Visibility Tab

- 4 Clear all of the check boxes except for the **Order Number**, **Order Amount**, and **Customer Number** check boxes.
- 5 Clear the **Save Data Columns** check box.
Optionally, if you want to save your column selections on closing the Order Browse dialog, click this check box.
- 6 Click **OK**.
The Order Browse dialog is displayed with the Order Number, Order Amount, and Customer Number columns only:



Order Browse After Specifying Display Options

Specify your own display options by experimenting with the Column Visibility tab on the Browse Options dialog.

Troubleshooting

If you encounter errors while using the demo application, ensure that all prerequisites listed in **Setting Up Prerequisites**, page 45, have been met. Your system administrator can help you with this.

In diagnosing the problem, refer to the *Construct Spectrum and SDK Client Installation Guide* and *Construct Spectrum and SDK Mainframe Installation Guide* to ensure that the client and server components have been installed correctly.

USING THE SUPER MODEL TO GENERATE APPLICATIONS

This chapter describes how to generate all of the application modules required to create a Construct Spectrum client/server application using the super model (VB-Client-Server-Super-Model).

The following topics are covered:

- **Overview**, page 94
- **Preparing to Generate with the Super Model**, page 97
- **Preparing to Generate with the Super Model**, page 97
- **Generating with the Super Model**, page 104
- **What to Do If Something Goes Wrong**, page 121
- **Transferring Your Application to the Client**, page 122

Overview

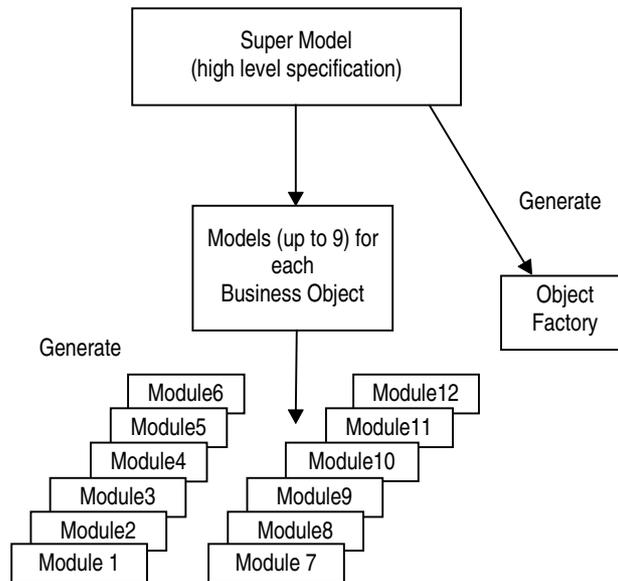
The super model (VB-Client-Server-Super-Model) is designed to be used as part of a rapid application development (RAD) process, where it is important to be able to generate a working client/server application from a minimum of input parameters.

The super model drives the generation of all the required modules for a client/server application using a single high-level model specification. For example, given a set of database file names defined in Predict, all the client and server modules required for fully functioning maintenance and browse services can be generated.

A single super model specification can generate all of the maintenance and browse modules required for up to 12 packages. A package contains the modules required to provide both browse and maintenance services for a business object. For example, the modules that make up the maintenance and browse services for a Customer Order business object are referred to as a package.

If you are creating a new application or adding a graphical front-end to an existing application, the fastest way to do this is by using the super model. The super model invokes each of the models necessary to produce your application.

Tip: The super model does not allow you to specify user exits. To specify user exits, regenerate using the specific model which supports the desired user exit.



Super Model Generation Overview

Using the super model, you can specify one or more high-level specifications. Each high-level specification corresponds to a business object such as a Customer Order object. Together, these specifications define the business objects in your client/server application. Next, select the models to run for each high-level specification. These models, using information derived from the business object's Predict file and field definitions, supply the specifications required to produce the Visual Basic and Natural modules for your application.

Because the super model requires few specifications, it uses many default values. If necessary, you can fine-tune and customize a module by re-generating it individually. Re-generating with the individual model enables you to override default values, add additional specifications, and add user exit code.

Another advantage to using the super model is that you can select to create an object factory module that defines all business objects within the application. The object factory performs many functions, for example, it enables you to use the Open dialog box by providing the names of all business objects within the application along with the actions they support.

Tip: If your application requires more than 12 packages, generate with the super model as many times as necessary to create all of the required modules.

Typically, you will use the super model to generate application modules when you develop the first iteration of your application. As you refine your application, you will likely need to regenerate certain application modules. In most cases, you will regenerate these modules separately using the individual models. Step-by-step instructions for generating application modules with the individual models are provided in the following chapters:

- **Generating a Subprogram Proxy**, page 132
- **Object-Maint Models**, page 479

Tip: The super model does not allow you to specify user exits. To specify user exits, regenerate using the specific model which supports the desired user exit.

Preparing to Generate with the Super Model

Before using the super model, do some planning and research to make your generation procedure go smoothly. The preparation you need to do includes:

- Establishing a naming convention
- Determining the domain name
- Understanding the object factory
- Determining the Predict default values
- Deciding which modules to generate

These tasks are described in the following sections.

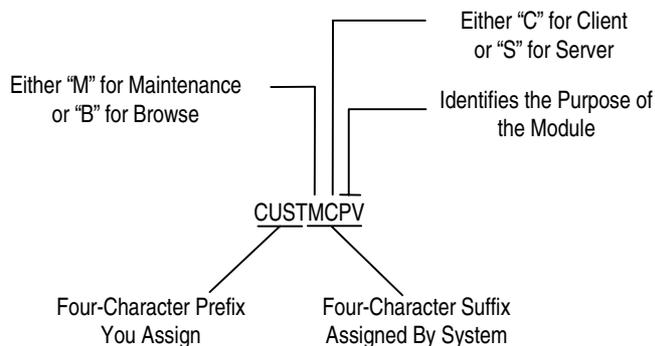
Using a Naming Convention

Establishing a naming convention is important because modules for up to 12 packages can be created with the super model at one time. Knowing the naming convention allows you to easily identify the package a module belongs to and what type of module it is.

If you use the super model, all the modules belonging to a package are given the four-character prefix you assign. If you assign a prefix that is less than four characters, the prefix is padded with dashes.

The module name suffix is defaulted by the super model. The suffix identifies the module type and can be up to four characters in length.

The default naming conventions applied to the module name are illustrated in the following diagram:



Naming Convention for a Generated Module

Note: These naming conventions apply only to modules generated by the super model.

Each suffix name that the super model uses as a default is described in the following table:

Default Module Suffix	Module Description
MSD	Object maintenance subprogram and two parameter data areas: object PDA and restricted PDA.
MSA	
MSR	
MSP	Object maintenance subprogram's proxy.
MCPV	Visual Basic maintenance object.
MCDV	Maintenance dialog.
BSO	Object browse subprogram and three parameter data areas: key PDA, row PDA, and restricted PDA.
BPRI	
BROW	
BKEY	

Default Module Suffix	Module Description (continued)
-----------------------	--------------------------------

BSP	Object browse subprogram's proxy.
BCPV	Visual Basic browse object.

Note: The parameter data areas MSA and MSR are created by the Object-Maint-Subp model. The parameter data areas BPRI, BROW, and BKEY are created by the Object-Browse-Subp model. The PDAs cannot be individually selected for generation.

Tip: You can override a default name by typing over the default value in the super model.

Understanding the Object Factory

Each Construct Spectrum application contains a module called the object factory. The purpose of the object factory is to make an application aware of its objects and the actions, such as a maintenance or browse action, associated with the objects. Each application also has an Open dialog (Open.frm) that enables users to select an object and one of its corresponding actions. When a user displays the Open dialog, the object factory populates it with a list of the application objects and their associated actions.

The super model allows you to generate an object factory. During subsequent iterations of your application, you have the option of regenerating an object factory with the super model or modifying the existing object factory by hand.

Tip: Because the super model can generate modules for up to 12 business objects at a time, you must generate with the super model multiple times if your application includes more than 12 business objects. In this situation, generate a unique object factory each time you generate with the super model. Later, do some coding to merge each object factory into a single object factory module.

For more information, see **Customizing the Object Factory**, page 307.

Which Modules to Generate

A package consists of two groups of modules, each bundling services for either a maintenance or browse function. For either group of services to be complete, all the modules belonging to a group must be generated and deployed. The modules are generated on the server but are deployed to either the server or the client.

You may choose to generate only certain modules. For example, if you already have an existing maintenance subprogram and you only want to generate a maintenance dialog, generate the following models: Subprogram-Proxy, VB-Maint-Object, and VB-Maint-Dialog. Later, if you decide to generate only a browse dialog, select only the Object-Browse-Subp, Subprogram-Proxy, and VB-Browse-Object models.

Tip: You must generate browse modules for a package if you want to allow users to browse the business objects in the package. Additionally, browse modules must be generated for the package if the business object is linked by a foreign field relationship to another business object. Foreign field relationships enable a user to browse and select key field values for foreign fields on a dialog.

For more information about foreign field relationships, see **Understanding the Browse and Maintenance Integration**, page 343.

Modules to Generate for a Maintenance Dialog

The following table shows the modules that you must generate to implement a client/server maintenance dialog. When you generate these modules individually, rather than using the super model, generate them in the order shown.

Module	Model Name	Result
Object maintenance subprogram, object PDA, restricted PDA	Object-Maint-Subp	Subprogram used to maintain a business object. This model also generates the PDA and restricted PDA for the object.
Object maintenance subprogram proxy	Subprogram-Proxy	Proxy used to communicate information between the Spectrum Dispatch Service and an object maintenance subprogram.
Visual Basic maintenance object	VB-Maint-Object	Visual Basic class instantiated by a maintenance dialog to encapsulate calls to the Spectrum Dispatch Client and implement local validations.
Visual Basic maintenance dialog	VB-Maint-Dialog	Dialog that provides graphical interface between the maintenance application and the user.
Object factory	VB-Client-Server-Super-Model	Visual Basic code module that identifies all business objects within an application and instantiates objects upon request.

Modules to Generate for a Browse Dialog

The following table shows the modules that you must generate to implement a client/server browse dialog. When you generate these modules individually, rather than using the super model, generate them in the order shown.

Module	Model	Result
Object browse subprogram, key PDA, row PDA, restricted PDA	Object-Browse-Subp	Natural subprogram used to encapsulate access to data on the server and return records as a series of rows. The parameter data areas (PDAs) communicate information to and from an object browse subprogram.
Object browse subprogram proxy	Subprogram Proxy	Proxy used to communicate information between the Spectrum Dispatch Service and an object browse subprogram.
Visual Basic browse object	VB-Browse-Object	For each object browse subprogram on the server, you must generate a supporting Visual Basic class. This class describes the object browse subprogram to the BrowseBase class, which in turn provides information to a browse dialog that is configured at runtime.
Object factory	VB-Client-Server-Super-Model or hand coded	Visual Basic code module that identifies all business objects within an application and instantiates objects upon request.

Unlike maintenance subprograms, which use a specific Visual Basic form for each maintenance dialog, all generated browse subprograms use the same underlying browse form. This browse dialog form communicates with a BrowseBase class to obtain information needed to configure itself for a particular browse subprogram and to retrieve data from the BrowseBase class.

Although many objects interact to produce a typical browse dialog, the important thing to know is that most of these are standard, reusable client framework components. For more information about browse processes, see **Creating and Customizing Browse Dialogs**, page 215.

Dependent Models

Some of the models used to generate the individual modules have dependencies on one another. This means you have to generate individual modules in an established order.

Note: If you use the super model to generate all the modules for a client / server object, the order of generation is managed for you.

The following table shows the dependencies between models:

Model and Module	Prerequisite Module
Object-Maint-Subp Object maintenance subprogram	None
Subprogram-Proxy Object maintenance subprogram proxy	Object maintenance subprogram
VB-Maint-Object Visual Basic maintenance object	Object maintenance subprogram proxy
VB-Maint-Dialog Visual Basic maintenance dialog	Visual Basic maintenance object
Object-Browse-Subp Object browse subprogram	None
Subprogram-Proxy Object browse subprogram proxy	Object browse subprogram
VB-Browse-Object Visual Basic browse object	Object browse subprogram's proxy

Generating with the Super Model

Generating with the super model involves four main tasks:

- 1 Invoking the super model to create a new specification.
- 2 Defining general parameters.
- 3 Defining specific package parameters.
- 4 Generating the modules.

Each task is described in the following sections, along with the steps you must follow to complete the task.

The super model is available in both the Generation subsystem on the server and in the Construct Windows interface. If you are using the model wizard, see **Using the Super Model Wizard in the Construct Windows Interface**, page 104. If you are using the model on the server, see **Using the Super Model in the Generation Subsystem**, page 114.

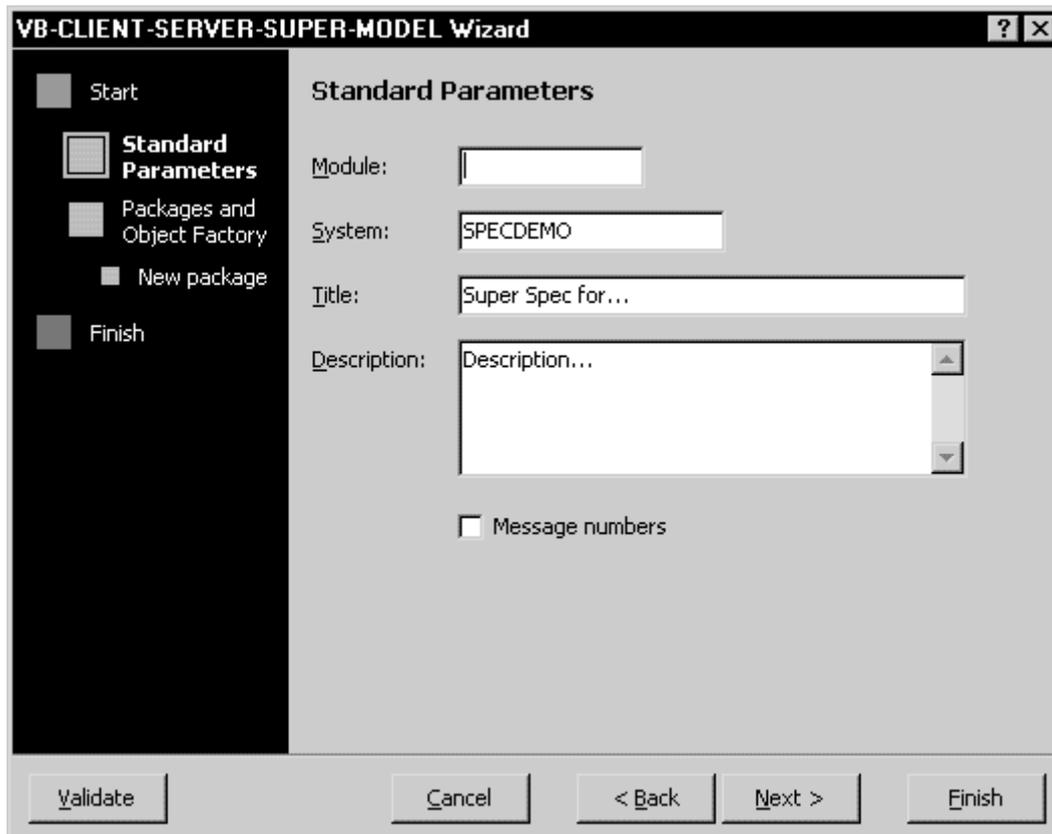
If you encounter problems, see **What to Do If Something Goes Wrong**, page 121.

Using the Super Model Wizard in the Construct Windows Interface

A model wizard is provided for the super model in the Construct Windows interface. For general information, see **Using the Construct Windows Interface**, page 77, in *Natural Construct Generation User's Manual*.

Step 1 — Invoke the Super Model Wizard

- To invoke the wizard:
- 1 On the **File** menu, click **New**, or click  on the toolbar. The **Create New Specification** dialog box is displayed.
 - 2 On the **Packages** tab, double-click VB-CLIENT-SERVER-SUPER-MODEL. The model wizard is displayed.
 - 3 Click **Standard Parameters** in the wizard navigator to view the **Standard Parameters** step:



VB-CLIENT-SERVER-SUPER-MODEL Wizard — Standard Parameters

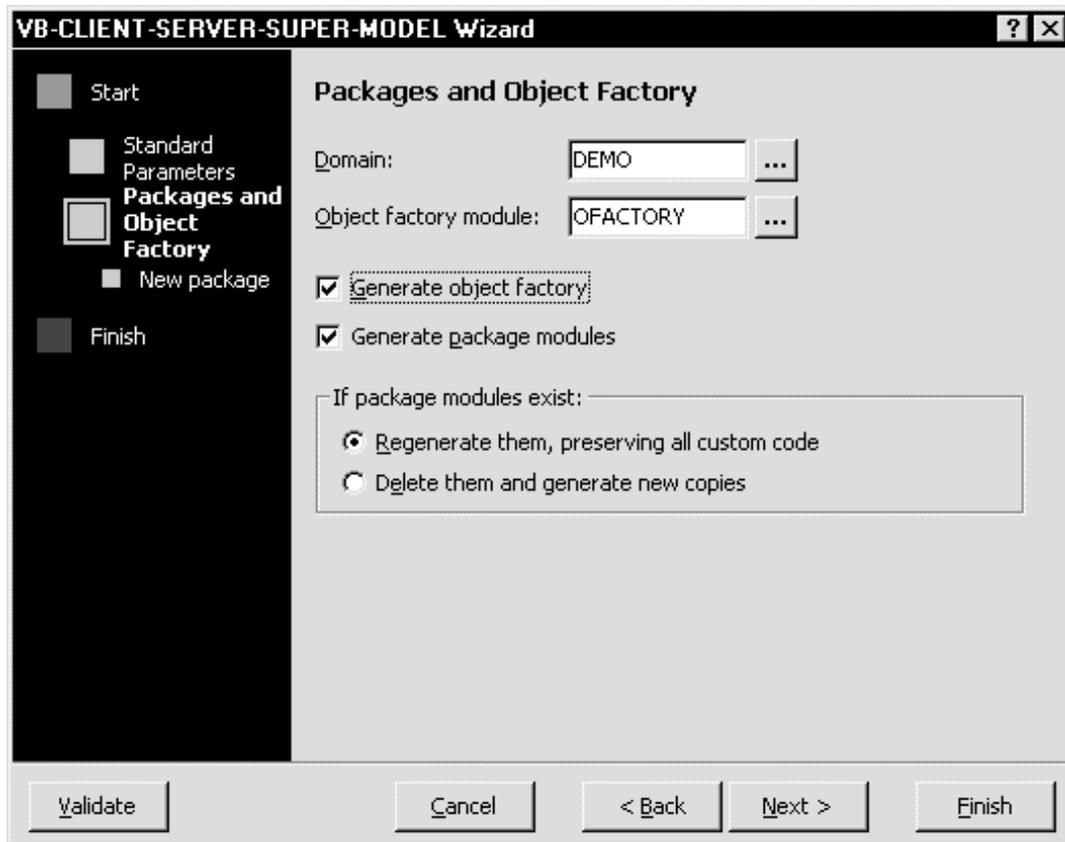
The **Standard Parameters** step is similar for all model wizards. The parameters in this step are described in **Standard Parameters Wizard Step**, page 269, in *Natural Construct Generation User's Manual*.

Click **Message Numbers** to use message numbers rather than message text for all REINPUT and INPUT messages in the generated subprogram.

Click **Next** to proceed to Step 2.

Step 2 — Define General Package Parameters

In the **Packages and Object Factory** step, identify the domain, object factory, and generation preferences for your application:



VB-CLIENT-SERVER-SUPER-MODEL — Packages and Object Factory

➤ To complete this wizard step:

- 1 Enter or select a domain.
For more information, see **Understanding the Object Factory**, page 99.
- 2 Type “OFACTORY” in the **Object factory module** text box.
- 3 Click **Generate object factory** to generate the object factory.

Tip: Use “OFACTORY” to identify your object factory as this is the default name used by the client framework.

Note: If you do not generate an object factory module, you must code it by hand on the client. This procedure is described in **Customizing the Object Factory**, page 307.

For more information about the object factory, see **Understanding the Object Factory**, page 99.

- 4 Select **Generate package modules** to generate the package modules.
- 5 Do one of the following:
 - If you are creating a new specification, click **Next** to proceed to the next step.
or
 - If modules already exist for the super model specification, select one of the following options:
 - By default, **Regenerate it, preserving all custom code** is selected. When you regenerate existing modules, any modified parameters in the specification will not be used during the regeneration. However, the model will:
 - Keep user exits
 - Apply updates from Predict (such as a new field or a BDT keyword)
 - Apply updates that have been added to the model’s code frames
 - To replace all existing modules with newly generated ones, click **Delete it and generate a new copy**.

In the next step, you can select the modules you want to regenerate or replace.

-
- Tip:** If you are regenerating some but not all modules for a package and have added custom actions that need to be reflected in the object factory:
- 1 Regenerate the modules.
 - 2 In a separate procedure, regenerate the object factory. Similarly, if you are adding modules to an existing package, for example, adding modules to support a browse service where currently only a maintenance service is provided, generate the new modules first and, in a separate procedure, regenerate the object factory.
 - 3 When you regenerate the object factory, select **Generate object factory**, but do not select **Generate package modules**.
 - 4 When you define the specific package parameters in the next step, **Defining Specific Package Parameters**, select all of the modules in your package so that the object factory is updated with all of the required information about your package.
-

Step 3 — Define Specific Package Parameters

In this step, specify details for each package in your application:

VB-CLIENT-SERVER-SUPER-MODEL Wizard

Start

Standard Parameters

Packages and Object Factory

NCST-ORDER-

Finish

Package prefix:

PREDICT view: ... Defaults

Primary key: ...

Hold field: ...

Description: Add Delete

Package modules:

Module	Gen.	Model	G/R/O	Library
ORD-MSO	<input checked="" type="checkbox"/>	Object Maintenance Subprogram	R	? ?
ORD-MSP	<input checked="" type="checkbox"/>	Spectrum Maintenance Proxy	R	? ?
ORD-MCPV	<input checked="" type="checkbox"/>	VB Maintenance Class	R	? ?
ORD-MCDV	<input checked="" type="checkbox"/>	VB Maintenance Form	R	? ?
ORD-BSO	<input checked="" type="checkbox"/>	Object Browse Subprogram	R	? ?
ORD-BSP	<input checked="" type="checkbox"/>	Spectrum Browse Proxy	R	? ?
ORD-BCPV	<input checked="" type="checkbox"/>	VB Browse Class	R	? ?

Check

Validate Cancel < Back Next > Finish

VB-CLIENT-SERVER-SUPER-MODEL Wizard — New Package

If you are working on an existing super model specification, the packages are displayed in the wizard navigator. Click a package in the wizard navigator to view it, or click Next to proceed through the packages.

- To add a new package, do one of the following:
 - While viewing the last package, click **Next** or **Add**.
 - or
 - Click **New Package** in the wizard navigator.

Note: The maximum number of packages you can create at one time is 12.

- To delete an existing package, go to that package and click **Delete**.
- To define specific package parameters:
 - 1 Provide a package prefix, which will be used to identify each module generated for the package. The prefix can be up to four characters long and should enable you to easily identify the package to which the generated modules belong. The importance of establishing a logical naming convention is explained in **Using a Naming Convention**, page 97. Once you provide a prefix for a new package, the **Package modules** grid is populated.
 - 2 Specify a Predict view.
 - 3 Provide the primary key, hold field, and object description.

Tip: Click **Defaults** to use default values for these three fields. You can also specify your own default override values using Predict keywords. Rather than typing these values directly, set up your file definition in Predict to default the required values.

For more information, see **Setting Up Predict Definitions**, page 47, in *Construct Spectrum Programmer's Guide*.

- 4 Determine which package modules to generate. The Package modules grid contains the following information:

Column	Description
Module	All of the modules that can be generated with the super model are listed. Each module is identified by the package prefix, followed by the standard suffix for the module type. For more information about suffixes, see Using a Naming Convention , page 97.
Gen.	Use the Generate check boxes to specify which modules will be generated. For more information about selecting modules, see Which Modules to Generate , page 100.
Model	Individual models the super model invokes to generate the package modules. Although seven models are listed, up to 12 modules can be generated. The Object-Browse-Subp model generates a subprogram, Key PDA, Row PDA, and Restricted PDA. The Object-Maint-Subp model generates a subprogram, Object PDA, and Restricted PDA.
G/R/O	<ul style="list-style-type: none"> • “G” indicates that modules do not currently exist in source form and will be generated and saved in the current library. • “R” indicates that modules currently exist in source form and will be regenerated and saved in the current library. This status occurs when you select Regenerate it, preserving custom code in Step 2. • “O” indicates that modules currently exist in source form and will be overwritten and saved in the current library. This status occurs when you select Delete it, and generate a new copy in Step 2.

Column	Description (continued)
Library	<p>Displays any of the following information:</p> <ul style="list-style-type: none"> • A question mark (?) indicates that you must click Check to determine if there is existing source or compiled (object) code for the module. • No content indicates that a check has been made, but there is no existing code for the module. • “S” indicates that source code exists. If the “S” is black, the source code is in the current library. If the “S” is red, the source code is in another library. To view the location of the source code, place the mouse pointer over the “S.” A pop-up window shows the library or libraries. <p>“C” indicates that compiled (object) code exists. If the “C” is black, the source code is in the current library. If the “C” is red, the source code is in another library. To view the location of the source code, place the mouse pointer over the “C.” A pop-up window shows the library or libraries.</p>

5 When you have finished specifying the parameters for all packages, click **Finish**.

6 In the **Finish** step, either:

- Click **Finish** to proceed to the Code window, where you can view the specification lines. The super model does not allow you to specify user exits. To specify user exits, regenerate using the specific model which supports the desired user exit. When you have finished viewing the Code window, proceed to Step 4.

Or

- Click **Generate** to proceed to Step 4.

Step 4 — Generate the Modules

You have two options for generating modules using the super model: you can generate in batch or you can generate from the model wizard.

Tip: If you are generating a number of modules, generate in batch to avoid tying up system resources.

Generating Modules from the Model Wizard

When you click **Generate** in the **Finish** step, the following process occurs as the super model generates:

- The super model specification is saved.
 - All the specifications for the individual modules are created and saved.
 - The Generate window is displayed. The **Module** pane provides information such as the module name, type, and action status. The **Message** pane provides a scrollable list of status messages from the server regarding the generation process. The **Message** pane displays the word “Done” when generation is complete.
- To terminate the generation process, click **Cancel**.

Generating Modules in Batch

- To generate in Batch:
- 1 On the **File** menu, click **Save** to save the specification.
 - 2 In a mainframe session, log on to the library where the specification is saved.
 - 3 Use the NCSTBGEN utility in batch to generate, specifying the name of your super model specification and the model name: VB-Client-Server-Super-Model.

For information about using this utility, see **Multiple Generation Utility**, page 1009, in *Natural Construct Generation User's Manual*.

Using the Super Model in the Generation Subsystem

Step 1 — Invoke the Super Model

- To invoke the super model:
 - 1 On the Natural Construct Generation Main menu, type “M” in the Function field.
 - 2 Type an eight-character name for the super model specification in **Module**. This name identifies the super model specification you are creating. The name should be descriptive so you can easily identify it as the super model specification for the application you are creating.
 - 3 Enter “VB-Client-Server-Super-Model” in the Model field. Alternatively, you can enter “VB-C”. The Standard Parameters panel is displayed:

```

CUSSMA                VB-CLIENT-SERVER-SUPER-MODEL Multi-module          CUSSMA0
May 28                Standard Parameters                               1 of 3

Module ..... OE-SPEC_
System ..... DEMO_____

Title ..... Multi-Object spec_____
Description ..... Order Entry demo system for Spectrum_____
_____
_____

Message numbers .... X

Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
right help  retrn quit                                           right main

```

Super Model Multi-Module — Standard Parameters Panel

The Standard Parameters panel is similar for all models. For information about the fields on this panel, refer to **General Model Specifications**, page 263, in *Natural Construct Generation User's Manual*.

Step 2 — Define General Package Parameters

On the General Package Parameters panel, identify the application packages for which you want to generate modules. You can generate up to 12 packages:

```

CUSSMB          VB-CLIENT-SERVER-SUPER-MODEL Multi-module          CUSSMB0
May 28          General Package Parameters                          2 of 3

Domain ..... DEMO_____ *
Gen object factory ..... X Object factory module ..... OFACTORY *
Only gen object factory .... _ Replace existing modules ... _

Package prefix      Predict view
ORD-                NCST-ORDER-HEADER_____ *
CUST                NCST-CUSTOMER_____
PROD                NCST-PRODUCT_____
WH--                NCST-WAREHOUSE_____
_____
_____
_____
_____
_____
_____
_____
_____

Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
right help  retrn quit                                left  right main

```

Super Model Multi-Object Specification General Package Parameters Panel

Tip: If you are regenerating some but not all modules for a package and have added custom actions that need to be reflected in the object factory:

- 1 Regenerate the modules.
- 2 In a separate procedure, regenerate the object factory. Similarly, if you are adding modules to an existing package, generate the new modules first, and in a separate procedure, regenerate the object factory.
- 3 When you regenerate the object factory, select the Gen object factory field and the Only gen object factory field.
- 4 When you define the specific package parameters, select modules in your package so the object factory is updated with all required information.

- To define general package parameters:
 - 1 Type the domain name for this application in the Domain field.
To display a list of domains from which to select a value, place the cursor in the field and press PF1. You must enter a value in this field. For more information about domains, see **Understanding the Object Factory**, page 99.
 - 2 To generate an object factory module, mark **Gen object factory**.

Note: If you do not generate an object factory module, you must code it by hand on the client. This procedure is described in **Customizing the Object Factory**, page 307.

For more information about the object factory, see **Understanding the Object Factory**, page 99.

- 3 Type “OFACTORY” in the Object factory module field.

Tip: Use “OFACTORY” to identify your object factory as this is the default name used by the client framework.

To generate only an object factory module, without regenerating any other modules, mark the Only gen object factory field. You must also select the package modules for which the object factory will be generated in the next step, **Step 3 — Define Specific Package Parameters**, page 117.

- 4 If you are using the super model to regenerate modules, you must decide whether you want to replace or regenerate existing modules. If you select the Replace existing modules option, the super model will replace any existing modules, including their user exit code. If you do not select this option, it will regenerate the existing modules but not the user exit code.

When you regenerate an existing module, any modified parameters in the specification will not be used during the regeneration. However, the model will:

- Keep user exits
- Apply updates from Predict (such as a new field or a BDT keyword)
- Apply updates that have been added to the model’s code frames

- 5 Type the prefix that will be added to each module generated for this package in the Package prefix field.
The prefix can be up to four characters in length and should enable you to easily identify the package to which the generated modules belong. The importance of establishing a logical naming convention is explained in **Using a Naming Convention**, page 97.
- 6 Type the primary file name for which the package is being generated in the Predict view field.
This is the file that represents your business object. This file must exist in Predict.
- 7 When you have added all of the primary files to be included in your application, together with a prefix name for each of the files, press Enter or PF11 to display the Specific Package Parameters panel.

Step 3 — Define Specific Package Parameters

In this step, specify generation details for each package included in your application.

```

May 28                VB-CLIENT-SERVER-SUPER-MODEL Multi-module          CUSSMCO
                        Specific Package Parameters                      3 of 3

>> 01  Package prefix ..... ORD-

Predict view ..... NCST-ORDER-HEADER_____ *
Primary key ..... ORDER-NUMBER_____ *
Hold field ..... ORDER-TIMESTAMP_____ *
Description ..... Order_____

---- Modules to Generate -----
      Model                Module      Source  Object  G/R/O
X Maint Object Subp      ORD-MSO_ *
X Maint Object Proxy    ORD-MSP_
X Maint VB Object       ORD-MCPV
X Maint VB Dialog       ORD-MCDV
X Browse Object Subp    ORD-BSO_
X Browse Object Proxy   ORD-BSP_
X Browse VB Object      ORD-BCPV

Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
      help  retrn quit      selct      bkwrdr frwrdr      left      main

```

Super Model Multi-Module — Specific Package Parameters Panel

Note: You must complete this panel for each package in your application.

This panel allows you to scroll through the packages in the application. The Package prefix field automatically shows the prefix defined in the General Package Parameters panel for the first package. The >> field shows which package is currently displayed.

- To scroll between packages, either:
 - Press PF8 (frwr) and PF7 (bkwr).
- Or
 - Enter a package number in the field following the two angle brackets (>>).
- To define specific package parameters:
 - 1 Specify a Predict view.
 - 2 Specify the primary key, hold field, and object description of your package file in the Primary key, Hold field, and Description fields, respectively.

Tip: Based on how the file is defined in Predict, the super model attempts to provide default values for these fields. You can also specify your own default override values using Predict keywords. Rather than typing these values directly, set up your file definition in Predict to default the required values.

For more information, see **Setting Up Predict Definitions**, page 47, in *Construct Spectrum Programmer's Guide*.

- 3 If you are generating both a maintenance and a browse function for this package, press PF5 (selct) to select all modules. Otherwise, mark the ones that you want to generate.

For information about determining which modules to generate, see **Which Modules to Generate**, page 100.

Note: Although only seven models are displayed on this screen, up to 12 modules can be generated. The Browse-Object-Subp model creates three additional modules: Key PDA, Row PDA, and Restricted PDA. The Maint-Object-Subp model generates two additional modules: Object PDA and Restricted PDA.

If you marked the Replace existing modules field on the General Package Parameters panel, any existing modules marked for generation will be replaced, including user exit code. For these modules, “O” is displayed in the G/R/O field. If you did not mark this field, existing modules will be regenerated and user exit code will be preserved. For these modules, “R” is displayed in the G/R/O field.

- 4 Press PF8 (frwrđ) to display the next package in your application. Complete the panel as described in steps 1 and 2. When you have entered specifications for all of your packages, return to the Natural Construct Generation Main menu.
- 5 Save your super model specification. You are now ready to generate the modules.

Step 4 — Generate the Modules

You have two options for generating modules using the super model: you can generate in batch or you can generate from the main menu.

Tip: If you are generating a number of modules, generate in batch to avoid tying up system resources.

- To generate from the Natural Construct Generation main menu:
- 1 Type “R” in the Function field.
 - 2 Type the name of the super model specification in the Module field and press Enter. This reads the super model specification into Natural Construct.
 - 3 Enter “G” in the Function field.

The following steps occur as the super model generates:

- The super model specification is saved.
- All the specifications for the individual modules are created and saved.
- The standard generation status window is displayed. You will also see a generated module status panel that lists the modules as they are generated and stowed by the super model.
- When all of the modules have been generated and stowed, a summary report is displayed listing the status of each module that was generated, detailing any errors that may have occurred.

➤ To generate in Batch:

- 1 Save the specification from the Natural Construct Generation main menu.
- 2 Use the NCSTBGEN utility in batch to generate, specifying the name of your super model specification, and the model name: VB-Client-Server-Super-Model.

For information about using this utility, see **Multiple Generation Utility**, page 1009, in *Natural Construct Generation User's Manual*.

What to Do If Something Goes Wrong

After generating with the super model, you should review the generation status report to reconcile any errors that may have occurred.

If a module was generated but not stowed because of a missing DDM, for example, you can regenerate the missing modules at a later time after the error has been corrected.

If there was a generation error for a specific module because of a missing dependent module, for example, you can regenerate the individual module from its model specification after you have corrected the error.

If the generation errors affect several of the individual modules, you may find it easier to regenerate them from the original super model specification after you correct the error. Re-read the original super model specification into Construct Spectrum and mark only those modules that require regeneration. Then repeat the generation step until all the modules have been successfully generated and stowed.

Tip: Ensure that the SYNERR parameter is set to “ON” in your user profile’s NATPARM. Otherwise, compilation errors in the code generated by the super model may cause cycling.

Transferring Your Application to the Client

If you have successfully generated all the modules of a package, or minimally all the modules of a browse or maintenance function, you are ready to download your client application modules to the PC and complete the process of creating a client/server application.

Using Visual Basic and the Construct Spectrum Add-In, you will set up a Construct Spectrum project, download application modules to your project, and compile the project to create a fully functional client/server application.

These steps are described in **Creating a Construct Spectrum Project**, page 123.

CREATING A CONSTRUCT SPECTRUM PROJECT

This chapter describes the process of setting up a Construct Spectrum project on your client. Follow the instructions in this chapter once you have generated your application modules on the server and are ready to download them to the client. This chapter also describes how to test, deploy, and set up security for your application.

The following topics are covered:

- **Overview**, page 124
- **Are You Ready?**, page 126
- **Creating the Project**, page 127
- **Downloading the Generated Modules**, page 131
- **What's Next?**, page 134

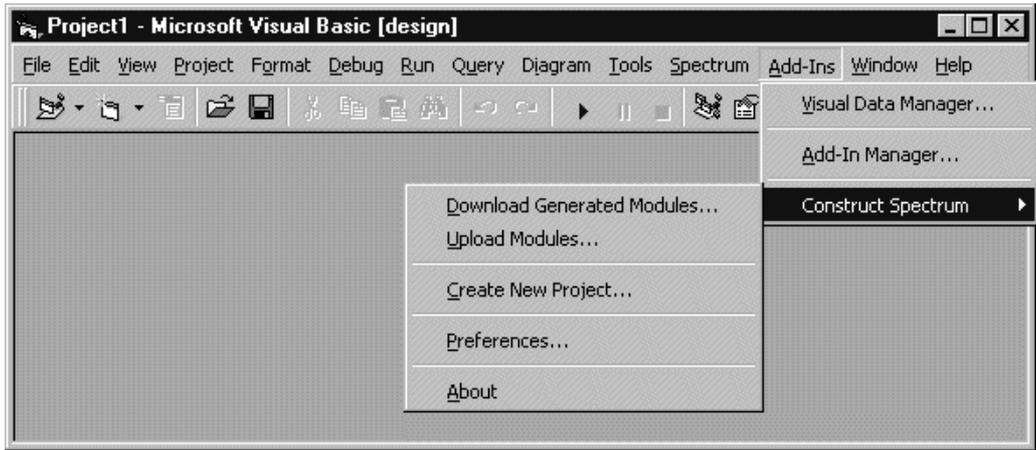
Overview

All Visual Basic client/server projects that use Construct Spectrum must include the Construct Spectrum client framework. Client framework components are reusable application components that provide a skeleton of functionality that interacts with generated and hand-coded Construct Spectrum modules to create a client/server application.

The client framework also includes forms, classes, procedures, global variables, and constants that are shared among various generated modules. This reduces the size of the generated modules and allows generated modules to interact through the shared components.

Construct Spectrum includes an Add-In that extends the Visual Basic Add-Ins menu with commands to:

- Create a Construct Spectrum project and add the client framework components to the project.
- Download generated modules from the server to the client and automatically add them to your project.
- Upload generated modules from the client to the server when you have customized the modules and need to regenerate them, preserving all of your customizations.



Construct Spectrum Add-In

Are You Ready?

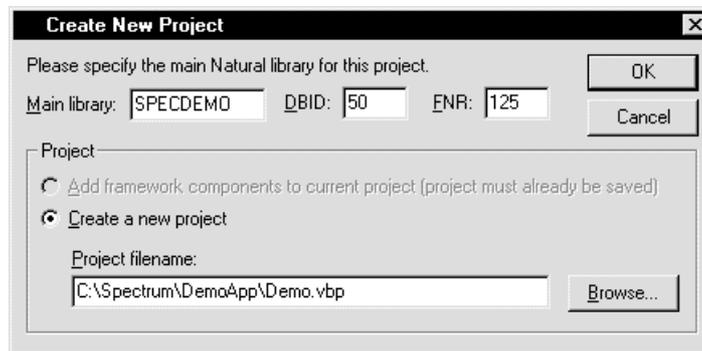
Before you use the Construct Spectrum Add-In to create a new project and download generated modules, use the following checklist to ensure that all prerequisites have been met:

- ❑ You have used the super model to generate the client and server modules of your application.
For information about using the super model, refer to **Using the Super Model to Generate Applications**, page 93.
- ❑ You know the library name, the database ID (DBID), and the file number (FNR) of the FUSER that contains the library where your generated modules reside.
- ❑ A Spectrum dispatch service is running.
For information about running a Spectrum dispatch service, refer to **Defining Construct Spectrum Services**, page 51, and **Managing Construct Spectrum Services**, page 72, in *Construct Spectrum Administrator's Guide*.

Creating the Project

The Construct Spectrum Add-In can create an entirely new project with all necessary client framework components or it can add client framework components to an existing project.

- To create a Construct Spectrum project:
 - 1 Start Visual Basic.
 - 2 Select **Create New Project** from the **Construct Spectrum** submenu. The Create New Project dialog is displayed:

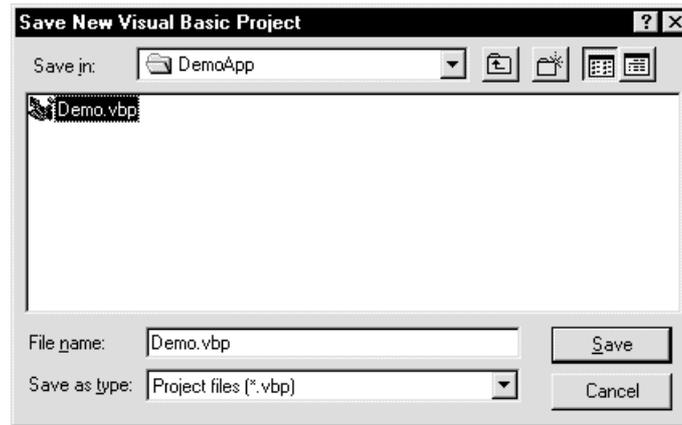


Create New Project

- 3 Type the name of the library containing your generated modules and the database ID (DBID) and file number (FNR) of the library's FUSER file.

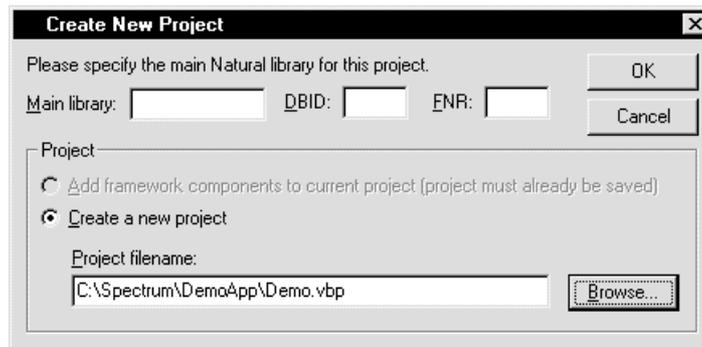
This information will be used as the default whenever you want to download or upload generated modules and will be stored in the AppSettings.bas module in your project.
- 4 Type the folder and project name in the **Project filename** text box and click **OK**.

Alternatively, you can click the Browse button to display a dialog from which you can select a folder (directory) and enter the name of your project.

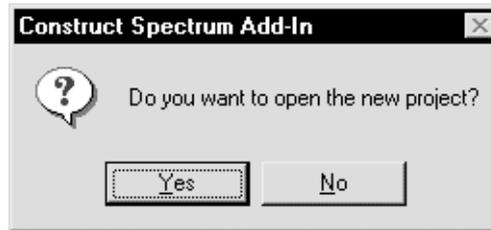


Save New Visual Basic Project

- When selecting a folder for your project using the **Save New Visual Basic Project** dialog, select **Open**.
The Create New Project dialog is displayed:



- Click **OK**.
Construct Spectrum creates the new project and prompts you to open it.



Prompt to Open New Project

- 7 Click **Yes** to open your new project or click **No** to open at a later time.

Most client framework components are not copied to your project folder. Instead, your Construct Spectrum project points to the FrameWrk5 folder in your Construct Spectrum installation directory. You can see this by choosing a client framework component such as Open.frm and choosing the Save As command on the Visual Basic File menu. These client framework components are shared among all Construct Spectrum projects created with the Construct Spectrum Add-In. Be aware that if you change one of these shared components and save it back to the FrameWrk5 directory, you could be affecting other projects.

For more information about customizing client framework components, see **Understanding and Customizing the Client Framework**, page 253.

The following client framework components are copied to your project folder because they are different for every application.

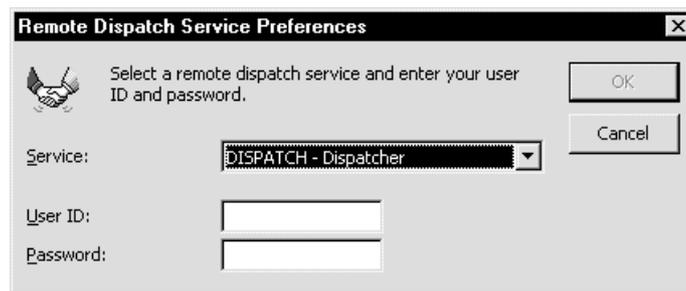
Name	Description
OFactory.bas	Contains the object factory; this module identifies all business objects within an application and instantiates objects upon request.
AppSettings.bas	Contains application-specific settings, such as the application name, library name, DBID, and FNR. You can change these settings by editing them in the module.

The Construct Spectrum Add-In also creates a new library image file for your application and places it in the project folder. The name of this file will be the library name with a “.lif” extension.

After the Construct Spectrum Add-In creates your project, you can run it and test the default functionality provided by the client framework. For more information about the client framework, see **Understanding and Customizing the Client Framework**, page 253.

Prior to Downloading

- To allow access to the mainframe:
- 1 Select **Remote Dispatch Service Preferences**:



Remote Dispatch Service Preferences Dialog

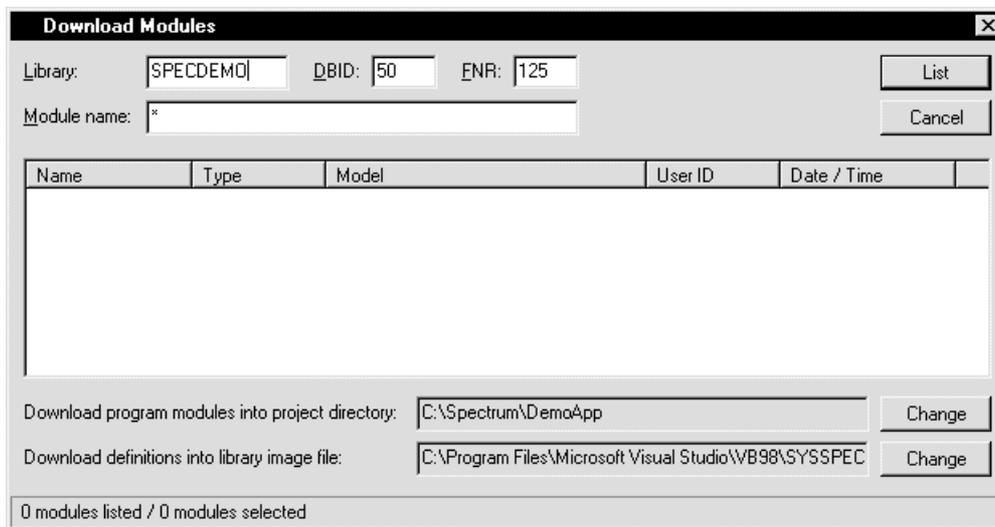
- 2 Enter your user ID and password.

For more information about this option, see **Understanding the Construct Spectrum Add-In**, page 49.

Downloading the Generated Modules

Next, download the client modules generated by the super model and add them to your project.

- To download the client modules and add them to your project:
- 1 On the Add-In menu, click **Construct Spectrum/Download Generated Modules**.
The Download Generated Modules dialog is displayed:



Download Generated Modules

Use this dialog to list the modules in a given library on the server and to select one or more modules to download.

- 2 The library name, DBID (database ID), and FNR (file number) default to the values entered for the last project created. If necessary, type the library name, DBID, and FNR that was specified for the project to which you are downloading.
- 3 In the **Module name** field, enter the package prefix followed by an asterisk.

4 Click **List**.

After a few seconds, a list of modules that match the module name pattern you entered are displayed.

Tip: If you know the name of the module you want to download, type it in **Module name**. When you click List or press Enter, the module is downloaded.

Note: When you have selected one or more modules in the list, the List button changes to Download.

Tip: To view your list in a different order, click a column header. The list is sorted according to the header item. If the list is already sorted, selecting the same header toggles the sort order between ascending and descending.

The following table lists the modules to download:

Module	Description
Parameter data areas	Parameter data area (PDA) definitions in a library image file. PDAs generated using the super model have “MSA”, “MSR”, “BKEY”, “BROW”, and “BPRI” suffixes.
Application service definitions	Application service definitions in a library image file. Modules have “App Service” type and “SUBPROGRAM-PROXY” model in the list. Subprogram proxies generated using the super model have “MSP” and “BSP” suffixes.
Visual Basic forms	Dialog definitions that are saved in the project folder with the extension “.frm” and automatically added to the project. Forms generated using the super model have a “MCDV” suffix.

Module	Description (continued)
Visual Basic classes	Modules saved in the project folder with a “.cls” extension and automatically added to the project. Classes generated using the super model have “MCPV” and “BCPV” suffixes.
Object factory	Visual Basic code module that identifies all business objects within an application and instantiates objects upon request. The name of this module is entered on the first panel of the super model. When downloaded, it is saved in the project folder with the extension “.bas”.

Tip: The lower part of the Download Generated Modules dialog shows the name of the project folder to which the modules will be downloaded and the name of the library image file where definitions will be saved. To change either of these, select the corresponding Change button.

- To download modules, select one or more from the list and click **Download** or press Enter.
Or, type the module names in the **Module name** text box and click **Download** or press Enter.

Hand-Coding the Object Factory

If you generated the object factory using the super model and downloaded it, you should be able to run your application without having to do any hand-coding. On the File menu, click Open to invoke the Open dialog; the objects and actions that you generated should be listed in the dialog.

If you did not generate an object factory, you must code it by hand. If you generated multiple object factories for your application, you must do some hand-coding to merge each object factory into one object factory module.

For information about hand-coding the object factory, see **Customizing the Object Factory**, page 307.

What's Next?

Once you have created the project and downloaded the generated components, you have the option of modifying the dialogs, testing and deploying the application, or setting up security.

Modifying the Dialogs

If this is an early iteration of your application, keep your dialog customizations to a minimum because you will lose these customizations when you regenerate the dialog. There are some modifications, however, that you need to do so that you can evaluate your application more effectively.

For more information about these modifications, see **Integrating a New Maintenance Dialog**, page 154.

Testing the Application

At this point, compile and run your application. Test the following things:

- On the File menu, click Open and test all objects and their associated actions to ensure each invokes the correct form.
- Check that each dialog displays correctly and that you have moved the controls in overflow frames onto the dialog form or onto separate tabs of a tab control.
- Test any local validations that were generated into the maintenance objects.
- Invoke and test the remote methods: Get, Next, Update, Add, and Delete.

Note: The first communication to the server typically takes a few seconds. This is because the EntireX Broker and DLLs must be loaded into memory and initialized. Subsequent calls to the server will be faster.

If you require more debugging information, see **Debugging Your Client/Server Application**, page 201, in *Construct Spectrum Programmer's Guide*.

Deploying the Application

Once your application has been tested, you can distribute it to your users. The procedure to deploy your application include:

- Creating the executable
- Collect the files to be installed
- Install the client application
- Run the application

For more information about deploying your application, see **Deploying Your Client/Server Application**, page 237, in *Construct Spectrum Programmer's Guide*.

Setting Up Security

Before allowing users to work with your application, you must implement security for their environment. You must define these users to a group. If users require different access privileges, set up one group for each type of user. You set up your application security based on these groups.

For information about setting up groups, see **Defining Groups and Users**, page 95, in *Construct Spectrum Administrator's Guide*.

Grant access to business objects by group and domain combination. You can grant a particular group/domain combination access to as many or as few business objects as necessary. Additionally, you can grant access to only specific methods within a group/domain and business object combination.

For more information about setting up security, see **Setting Construct Spectrum Security Options**, page 123, in *Construct Spectrum Administrator's Guide*.

CREATING AND CUSTOMIZING MAINTENANCE DIALOGS

This chapter provides step-by-step instructions for generating the modules required to maintain server information from a maintenance dialog on the client. It describes how to generate the necessary modules, download them to the client, integrate them into an existing Construct Spectrum project, and maintain server database information from your maintenance dialog. Also included is information on how to customize the maintenance dialog. It provides conceptual information, suggestions on the best way to approach customization problems, and step-by-step instructions for particular customization tasks.

The following topics are covered:

- **Overview of the Maintenance Dialog**, page 138
- **Are You Ready?**, page 140
- **Using Individual Models to Generate Maintenance Modules**, page 141
- **Downloading Client Modules**, page 151
- **Integrating a New Maintenance Dialog**, page 154
- **Strategies for Customizing a Maintenance Dialog**, page 155
- **Customizing on the Server**, page 159
- **Customizing on the Client**, page 175
- **Uploading Changes to the Server**, page 212

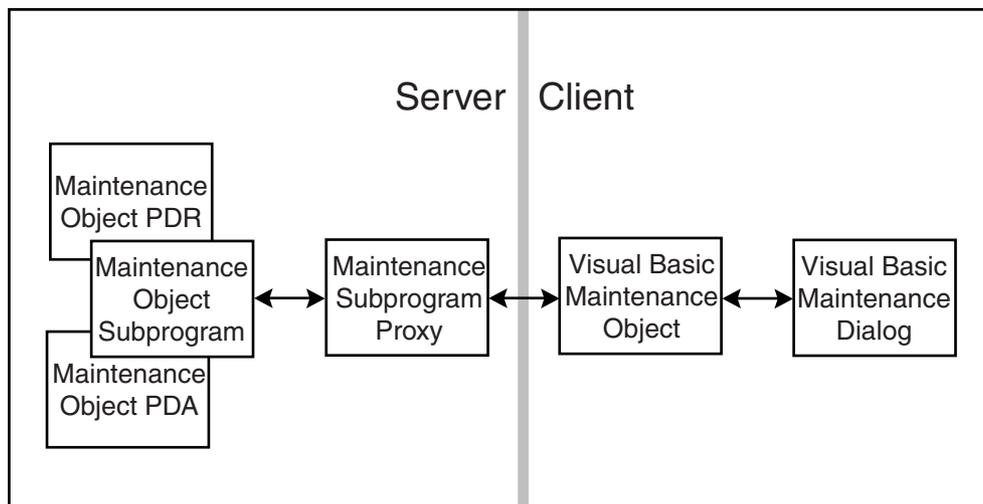
Overview of the Maintenance Dialog

Maintenance dialogs are built on the foundation provided by the existing Natural Construct object methodology. A maintenance dialog generated with Construct Spectrum can share data access modules with a character-based maintenance dialog.

The modules that must be generated to create a working Construct Spectrum maintenance dialog are:

- Object maintenance PDA
- Object maintenance PDR
- Object maintenance subprogram
- Maintenance subprogram proxy
- Visual Basic maintenance object
- Visual Basic maintenance dialog

The following example shows the relationship between these generated modules:



Relationships Between Client and Server Maintenance Components

Ways to Generate Maintenance Dialogs

Each module that a maintenance dialog requires can be generated with the VB-Client-Server-Super-Model or generated one at a time using individual models. To determine which generation approach is best for you, consider the following guidelines:

- If you are creating a new application or a new business object, use the super model.
- If you are making major changes to the Predict file definitions of one or more business objects in an existing application, use the super model.
- If you want more control over the generation results, such as customized code for user exits, use the individual models.

This section explains how to generate maintenance modules from the individual models. For information about using the super model, see **Using the Super Model to Generate Applications**, page 93.

The Process of a Maintenance Dialog

The first part of this chapter describes the tasks required to create a maintenance dialog. These include:

- Use the Construct models to generate modules
- Download the modules to the client using the Construct Spectrum Add-In
- Integrate a new maintenance dialog into your application

Once you have completed these steps, it is time to compile the application and test the new maintenance dialog.

The second part of this chapter discusses various strategies for customizing a maintenance dialog and different customization mechanisms available on both the client and server. It also explains how to upload client changes to the server.

Are You Ready?

Before generating a module for a maintenance dialog, use this checklist to ensure that the following prerequisites are met:

- ❑ The necessary Predict file(s) have been created, along with any relevant Predict definitions such as file relationships and verification rules.
- ❑ An object PDA, a restricted PDA, and an object maintenance subprogram exist for the target Predict file(s). If these modules do not exist as part of a previously generated Construct Spectrum application, create them now.

For information about creating these modules, see **Object-Maint Models**, page 479, in *Natural Construct Generation User's Manual*.

- ❑ The Entire Net-Work kernel is running on your client (if you are using Entire Net-Work) so that you can access the server used by the Spectrum dispatch service and the Spectrum security service. Your system administrator should ensure that this prerequisite is met.
- ❑ A Spectrum dispatch service and security service are set up to service requests from your client. To determine whether these services are available, ping the service using the Spectrum Service Manager. Your system administrator should ensure that these services are available on the client.

For information about managing Construct Spectrum services, see **Defining and Managing Construct Spectrum Services**, page 47, in *Construct Spectrum Administrator's Guide*.

- ❑ You are set up as a user with a password to access the Construct Spectrum environment.
- ❑ You have created a Construct Spectrum project. If you have not done so, create one now.

For information about creating a Construct Spectrum project, see **Creating a Construct Spectrum Project**, page 123.

Using Individual Models to Generate Maintenance Modules

The modules required to run a maintenance dialog share many files and parameters. If you are using individual models to generate your maintenance modules, you must generate the models in a specific order. Each model reads the source code generated by earlier models to make generation decisions.

Generate the dialog models in the following order:

- 1 Object-Maint-Subp model (object maintenance subprogram)
- 2 Subprogram-Proxy model (maintenance subprogram proxy)
- 3 VB-Maint-Object model (Visual Basic maintenance object)
- 4 VB-Maint-Dialog model (maintenance dialog)

Tip: Use the same four-character prefix to name all generated modules pertaining to a single business object. This convention makes it easier to select modules for downloading. For example, to download all client modules related to a Customer business object, type “CUST*” (where “*” is the wildcard character) to narrow the list of available items to those starting with CUST.

The models are available in the Generation subsystem on the server or you can use the model wizards in the Construct Windows interface.

Generating the Object Maintenance Subprogram and PDAs

The Object Maintenance subprogram is used to maintain a business object. This model also generates the PDA and restricted PDA for the object. Before generating a module for a maintenance dialog, ensure that an object PDA, a restricted PDA and an object maintenance subprogram exists for the target Predict file(s).

For more information on creating a PDA, see **Create the External PDA**, page 242, and **Create the Restricted PDA**, page 245, in *Natural Construct Generation User's Manual*.

Generating the Maintenance Subprogram Proxy

A subprogram proxy is required to access the generated object maintenance subprogram from the client application. The subprogram proxy calls an object maintenance subprogram, which fulfills a request on behalf of a maintenance dialog. It is also responsible for converting data between the network transfer format and the Natural data format used in the object maintenance PDA and restricted object maintenance PDA. Typically, you will not have to customize or provide any user exit code for this model — just generate and catalog it.

For information, refer to **Generating a Subprogram Proxy**, page 132, in *Construct Spectrum Programmer's Guide*.

Generating the Visual Basic Maintenance Object

The VB-Maint-Object model generates a Visual Basic maintenance object that provides maintenance dialogs with access to the business object data and methods in the Spectrum Dispatch Client.

Business Validations

A Visual Basic maintenance object is an ideal place to code simple business validations such as verification rules. The model provides the CLIENT-VALIDATION user exit for this purpose.

The VB-Maint-Object model also extracts verification rules that are attached to your Predict file and field definitions and generates validation code into a subroutine called “Validate”. The following code example illustrates the type of validation code that would be generated if the Predict verification type, Range, was attached to a field called “CUSTOMER-NUMBER”.

Example of validation code generated by the VB-Maint-Object model

```
...  
Case "CUSTOMER-NUMBER"  
    If Value < 2 or Value > 4 Then  
        Err.Raise Number:=csterrValueOutOfRange, _  
            Source:=OBJECT_PDA_NAME, _  
            Description:=csterrValueOutOfRangeMsg  
    ...
```

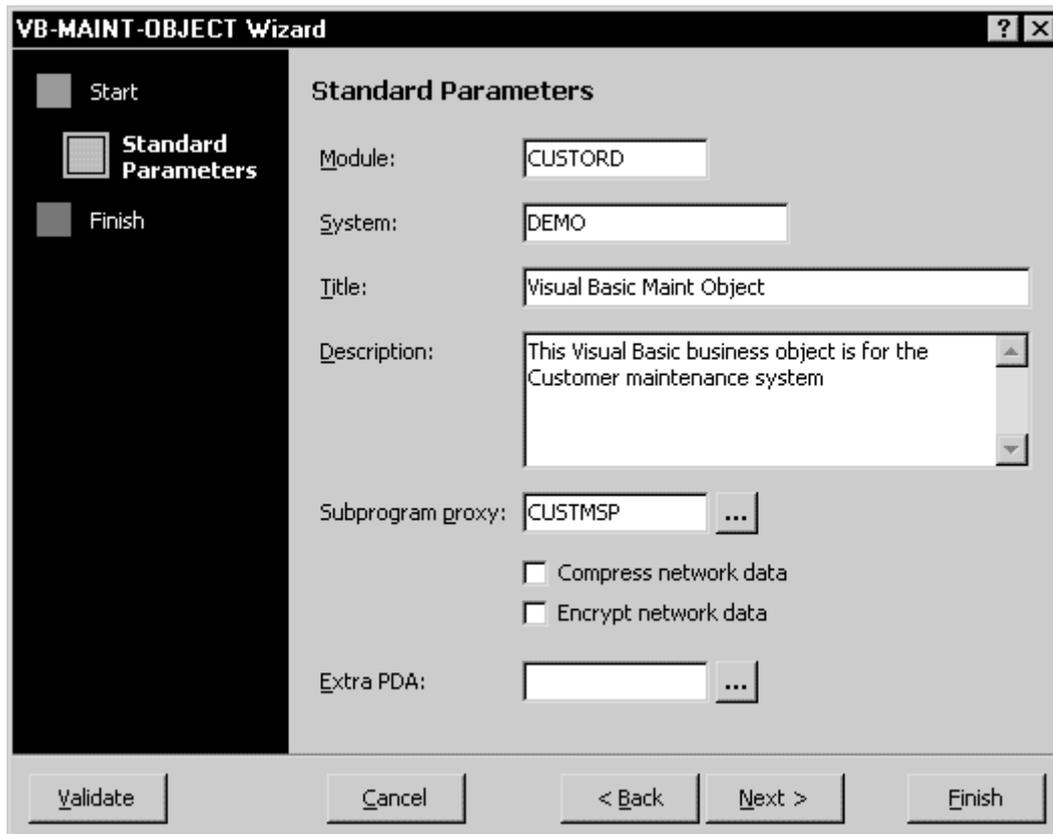
For more information about validating data, see **Validating Your Data**, page 327.

Browse Functions

The VB-Maint-Object model also generates methods that enable your maintenance dialog to have browse functions automatically linked to the primary key and all foreign keys on the dialog.

For more information about linking browse and maintenance functions, see **Understanding the Browse and Maintenance Integration**, page 343.

You can generate the Visual Basic Maintenance Object in the Generation subsystem on the server or using the VB-Maint-Object wizard in the Construct Windows interface. The following example shows the Standard Parameters step in the VB-Maint-Object wizard:



VB-MAINT-OBJECT Wizard — Standard Parameters

The Standard Parameters step is similar for all model wizards. The common parameters Module, System, Title, and Description are described in **Standard Parameters Wizard Step**, page 269, in *Natural Construct Generation User's Manual*.

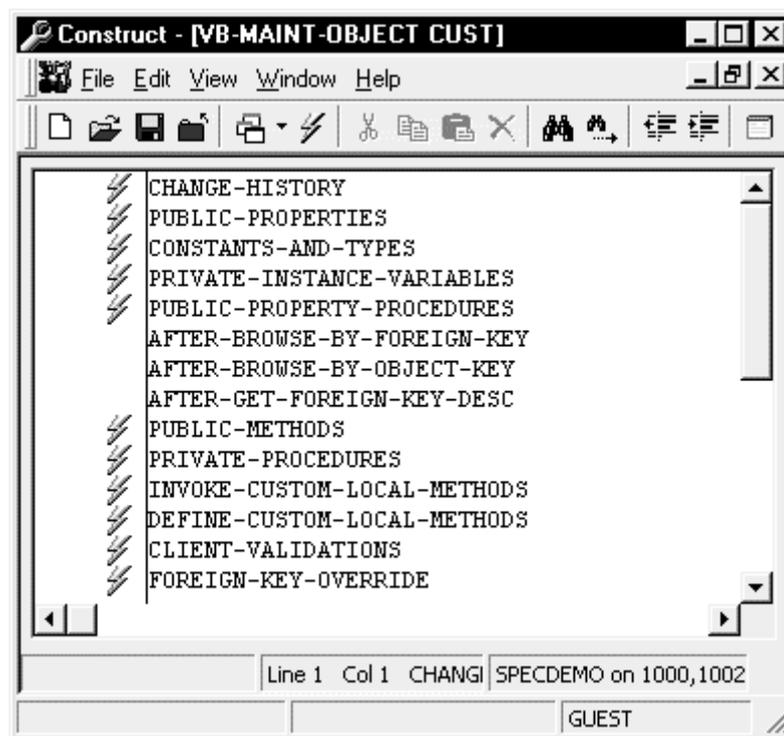
For general information, see **Using the Construct Windows Interface**, page 77, in *Natural Construct Generation User's Manual*.

The parameters on this panel are:

Parameter	Description
Subprogram proxy	Name of the subprogram proxy that communicates with the object browse subprogram for this Visual Basic browse object.
Compress network data	Indicates whether the parameters sent to the server are compressed to reduce transmission time.
Encrypt network data	Indicates whether the parameters sent to the server are encrypted. Encryption secures sensitive data.
Extra PDA	Additional parameter for your maintenance object subprogram (for example, to update foreign field descriptions on a maintenance dialog without having to make an extra call to the server). For more information about defining extra PDAs, see How Foreign Field Descriptions Are Refreshed , page 361.

*Note: The Compress data and Encrypt data flags only apply to data sent from the client to the server. To enable compression and encryption for data sent from the server to the client, set the Compress data and Encrypt data flags in the subprogram proxy, which is described in **Generating a Subprogram Proxy**, page 132, in *Construct Spectrum Programmer's Guide*.*

After supplying model parameters, you can customize the generation results by creating user exit code for the module. The following example shows the available user exits in the Code window for a Visual Basic maintenance object:



Code Window — VB-MAINT-OBJECT

The ⚡ icon indicates that sample code can be generated for the user exit. To do so, right-click the user exit and select Generate Sample from the shortcut menu. You can then modify the code as required.

For more information about using the Code window, see **Using the Construct Windows Interface**, page 77, in *Natural Construct Generation User's Manual*.

For more information about the user exists for this model, see **User Exits for the Natural Construct Models**, page 575, in *Natural Construct Generation User's Manual*.

Generating a Maintenance Dialog

The VB-Maint-Dialog model generates a maintenance dialog that provides users with a graphical user interface to data and a business object (the object maintenance subprogram) on the server. A maintenance dialog is used to maintain information for a given business object. The dialog can support any object PDAs that can be generated.

All tailoring for maintenance dialogs should be performed within the Visual Basic environment. In most cases, you will have to reposition and resize the GUI controls on the form. By default, the VB-Maint-Dialog model generates GUI controls in two columns with labels on the left and input controls on the right. The need for visual tailoring is especially evident when generating dialogs that have many fields. For more information about tailoring forms, see **Integrating a New Maintenance Dialog**, page 154.

Unlike other Construct Spectrum models, the VB-Maint-Dialog model does not support full regeneration capabilities and, therefore, supplies few user exits. You can, however, add your own user exits to preserve hand-written code and to minimize the changes required after regenerating your dialogs.

Customizations made to Visual Basic forms are not preserved during regeneration. If this is an early iteration of the application, limit any modifications to those described in the following table:

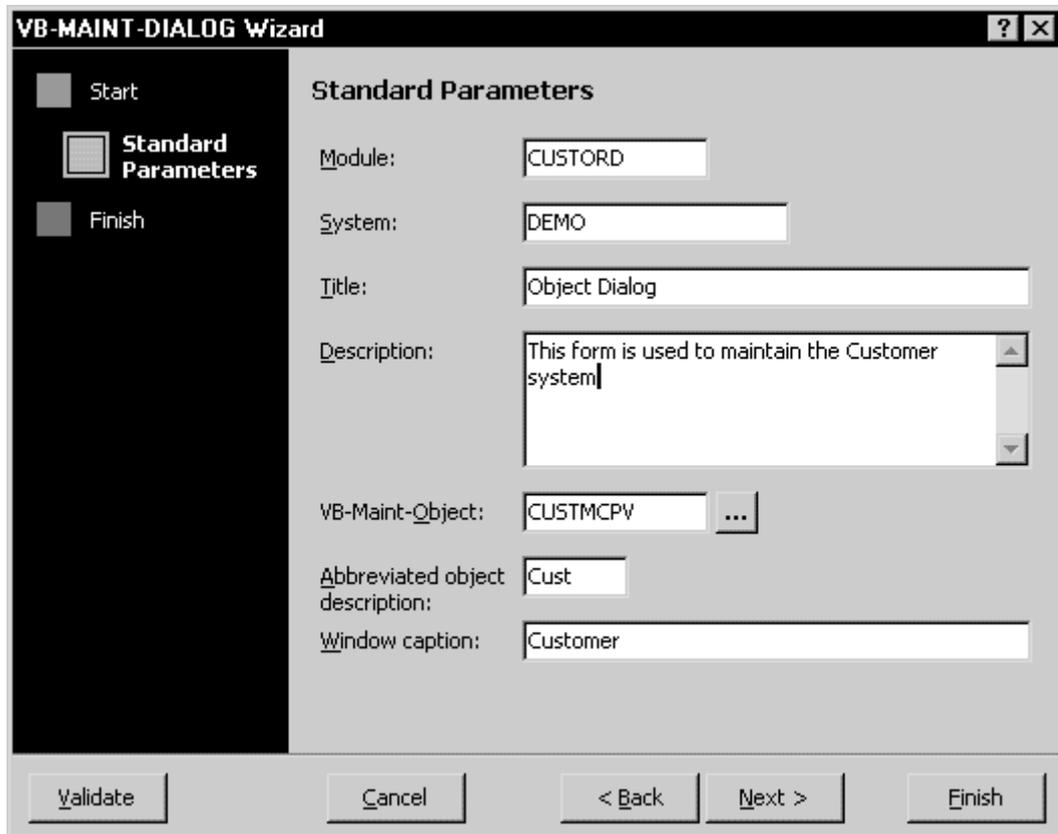
Modification	Description
Correcting overflow conditions	Overflow conditions occur when there are more fields than can be displayed on a dialog. Unless you correct the problem, these fields will be hidden.
Resizing grid controls	A grid control is a table with rows and columns that displays related information on a dialog. For example, a list of line items on a purchase order dialog. You can adjust the size of a grid to suit your GUI layout.

For more information about overflow conditions, see **Overflow Conditions**, page 174.

For more information about resizing grids, see **Resizing Grids**, page 206.

Before regenerating a maintenance dialog, see **Strategies for Customizing a Maintenance Dialog**, page 155, for information about saving customizations in your maintenance dialog.

You can use the VB-Maint-Dialog model in the Generation subsystem on the server or use the model wizard in the Construct Windows interface. The following example shows the Standard Parameters step for the VB-Maint-Dialog wizard:



The screenshot shows the 'VB-MAINT-DIALOG Wizard' window. On the left, a vertical sidebar contains three steps: 'Start', 'Standard Parameters' (which is selected and highlighted), and 'Finish'. The main area is titled 'Standard Parameters' and contains several input fields: 'Module:' with the value 'CUSTORD', 'System:' with 'DEMO', 'Title:' with 'Object Dialog', 'Description:' with a text area containing 'This form is used to maintain the Customer system', 'VB-Maint-Object:' with 'CUSTMCPW' and a browse button (...), 'Abbreviated object description:' with 'Cust', and 'Window caption:' with 'Customer'. At the bottom, there are five buttons: 'Validate', 'Cancel', '< Back', 'Next >', and 'Finish'.

VB-Maint-Dialog Wizard — Standard Parameters

The Standard Parameters step is similar for all model wizards. The common parameters (Module, System, Title, and Description) are described in **Standard Parameters Wizard Step**, page 269, in *Natural Construct Generation User's Manual*.

The parameters on this dialog are:

Parameters	Description
VB Maint Object	Name of the Visual Basic maintenance object. Click the browse button to select a module.
Abbreviated object description	Used in naming GUI controls on a form generated by the VB Maint Dialog model. For example, a GUI control for a field named CUSTOMER-NUMBER in an object named Customer might have a GUI control name of txt_CUST_CustomerNumber, where CUST represents the abbreviated object description. The default value for the abbreviated object description is the first four characters of the module name.
Window caption	Caption for the resulting maintenance dialog.

Once you generate the required modules, download them to your client.

Note: Ensure that all modules generated on the server are cataloged before downloading to the client.

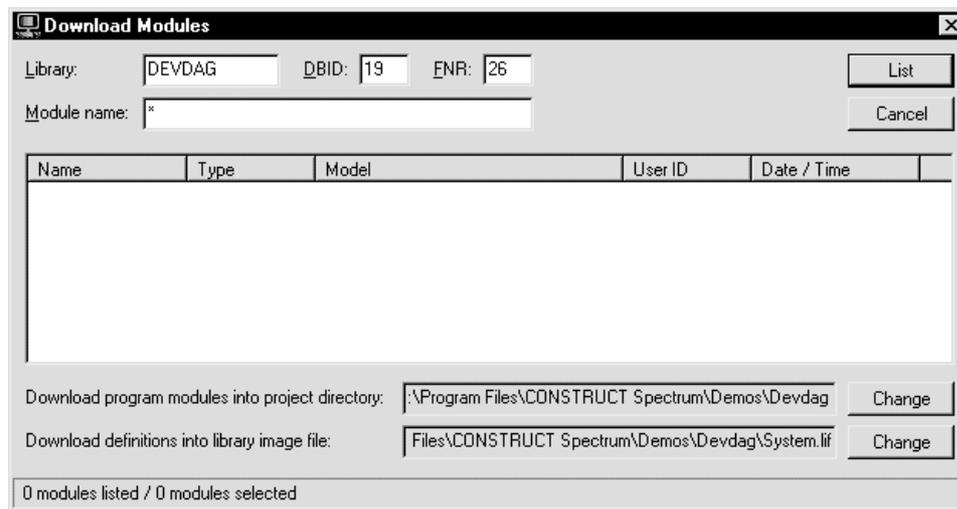
Downloading Client Modules

After generating all required maintenance modules, you must download those modules required on the client. The following table lists the modules that are required on the client and provides a brief description of their roles.

Model	Module Suffix	Visual Basic Extension	Description
Object-Maint-PDA	MSA	n/a	Encapsulates a business object. This parameter data area definition is incorporated into the library image file used by the project.
Object-Maint-PDA-R	MSR	n/a	Contains private data used by the business object. This restricted PDA definition is incorporated into the library image file used by the project.
Subprogram-Proxy	MSP	n/a	Communicates information between the Spectrum dispatch service and an object maintenance subprogram. Also updates the library image file with application service definitions that contain information about the object maintenance subprogram's methods and the data they require.
VB-Maint-Object	MCPV	.cls	Communicates with the object subprogram on the server on behalf of the maintenance dialog. Also implements validations on the client.
VB-Maint-Dialog	MCDV	.frm	Provides the graphical interface between the maintenance application and the user.

Note: The module suffixes listed in the table are suggestions only. However, when generating with the super model, modules are given these suffix names automatically.

- To download modules from the server to the client:
 - 1 Open the Construct Spectrum project that you are updating.
For information about setting up a project, see **Creating a Construct Spectrum Project**, page 123.
 - 2 On the Construct Spectrum Add-In, click **Download Generated Modules**.
The Download Modules dialog is displayed:



Download Modules

Note: Ensure that you are pointing to the correct Natural library and FUSER system file on the server.

- 3 If the default values in the **Library**, **DBID** (database ID), and **FNR** (file number) text boxes do not specify the server library from which you want to download, type the correct values in these fields.

Tip: The project folder to which the modules will be downloaded and the name of the library image file where definitions will be updated are shown in text boxes at the bottom of the dialog. To change either of these, select the corresponding Change button.

- 4 Enter a pattern (such as CUST*) in the **Module name** text box to list all modules matching that pattern.
- 5 Click **List** or press Enter.
A list of server modules is displayed. The maintenance modules you generated will be among them.
- 6 Select the maintenance modules you generated and click **Download**.
You can identify the maintenance modules based on their module suffixes, which are shown in the table at the beginning of this section.

The Visual Basic maintenance object and the maintenance dialog (.frm file) are automatically added to your Construct Spectrum project.

For more information about downloading modules to the client and about setting up a Construct Spectrum project, see **Creating a Construct Spectrum Project**, page 123.

For more information about tailoring on the client, see **Tailoring the Maintenance Dialog**, page 177.

Integrating a New Maintenance Dialog

If you are creating a new maintenance dialog and wish to add it to an existing Construct Spectrum application, hand-code the object factory to link the maintenance dialog to your application. You need to hand-code the object factory only if you are adding a new dialog to your application or you have changed the actions available for an existing business object. An example of changing the available actions for a business object is a situation where you add a maintenance action to a business object that had been available to the user only through a browse action.

Tip: To determine whether you need to hand-code the object factory, access the Open dialog and select each object and its associated action. If the selected object action does not open or if the Open dialog does not display all of the object actions, do some hand-coding to add the required object actions.

For information about hand-coding the object factory, see **Customizing the Object Factory**, page 307.

Strategies for Customizing a Maintenance Dialog

This section describes some strategies you can use to reduce the effort required to maintain your maintenance dialogs. These include:

- Doing the Predict data dictionary work up front
- Choosing the most appropriate place to add hand-written code
- Adding new user exits
- Making a copy of your changes

Doing the Predict Data Dictionary Work Up Front

Before tailoring the dialog, ensure that your data design is sound. If your data design is unstable, but you want to test the functionality of your dialog, consider postponing tailoring tasks such as creating calculated fields or rearranging the layout of your dialog until your data design is stable.

Construct Spectrum has added new points of integration with Predict that make it possible to generate robust dialogs with minimal tailoring, provided you take the time to enter the information into Predict. Following are some ways that you can enhance your generated dialog by providing Predict information:

- Enter values for Header1, 2, and 3 in the field definitions. The VB-Maint-Dialog model uses this information to generate meaningful label captions. For more information about how label captions are derived, see **Deriving Variable Names**, page 159.
- Create and attach table status verifications to fields whenever you know there is a finite set of valid values. The model uses verifications to decide what type of GUI control to generate. If a table status verification is attached to the field, the model will create either a ComboBox or a Frame and series of OptionButtons. The code that gets generated for these types of controls is different than the code generated for TextBox controls. For information about using Predict verification rules, see **Overriding GUI Controls**, page 160.
- Supply GUI and BDT keywords to help the model determine which type of GUI control to use or to fine-tune the behavior of a TextBox control. For information about how Predict keywords affect GUI generation, see **Overriding GUI Controls**, page 160.

Choosing an Appropriate Place to Add Hand-Written Code

There are many places in a Construct Spectrum-generated application to place custom code — like a Visual Basic maintenance object or in a separate Visual Basic module that you add to the application. When adding custom code to a maintenance dialog, determine if this code can be placed elsewhere and still work.

The primary reason for placing code in the dialog is to have the ability to respond directly to specific events. In such cases, you have no choice but to put code in the dialog. However, rather than writing 10 or 20 lines of event code directly in the dialog, write one line of code in the form that calls a routine in another module that can do the work for you. The following examples illustrate the difference between these two approaches:

Significant Impact on Dialog Code

```
Private Sub txt_EMPL_Salary_Change()  
    'my custom code - start  
    Dim Result As String  
    If CCur(txt_EMPL_Salary.Text) > 100000 Then  
        txt_EMPL_Salary.BackColor = vbRed  
        txt_EMPL_Salary.ForeColor = vbYellow  
        Result = InformAuthorities(EmployeeName)  
        Select Case Result  
            Case "EmployeeHasAcknowledged"  
                PublishSalaryAtPressRelease EmployeeName  
            Case "SalaryIsIncorrect"  
                Beep  
            Case "TerminateEmployee"  
                PerformAction "DELETE"  
        End Select  
    End If  
    'my custom code -end  
  
    If DetectChanges Then  
        ObjectChanged = True  
    End If  
End Sub
```

Minimal Impact on Dialog Code

```
Private Sub txt_EMPL_Salary_Change()  
    'my custom code - start  
    CheckSalary EmployeeName  
    'my custom code -end  
  
    If DetectChanges Then  
        ObjectChanged = True  
    End If  
End Sub
```

Using the second approach simplifies and minimizes the modifications that you must re-implement if the dialog is regenerated.

Adding New User Exits

Unlike other Construct and Construct Spectrum models, the VB-Maint-Dialog model comes with few predefined user exits. You can, however, add your own user exits to the dialog code. These user exits are saved when you regenerate your maintenance dialog and, therefore, reduce the effort required to maintain your dialogs on an ongoing basis.

➤ To add new user exits to the maintenance dialog:

1 Define the user exit.

Each custom user exit must be delimited with comment lines that indicate where your custom code begins and ends. Use the standard 'SAG DEFINE EXIT *abc*' and 'SAG END-EXIT' delimiters to mark the beginning and ending of your user exit. Provide a unique name for the user exit. A good convention to follow is to name the user exit after the code block in which it is found. For example, if you add custom code to the lost focus event for the txt_CUST_CustomerNumber GUI control, use the following delimiters to block your custom code:

```
'SAG DEFINE EXIT txt_CUST_CustomerNumber_LostFocus  
    txt_CUST_CustomerNumber.ForeColor = vbGreen  
'SAG END-EXIT
```

2 Upload, regenerate, and download the maintenance dialog.

Before regenerating the dialog, upload the dialog to the server to preserve your custom coding changes. After regenerating, download the maintenance dialog.

Note: You cannot preserve tailoring to the visual appearance of a maintenance dialog with user-defined user exits.

- 3 Reposition user exit code.
As part of the regeneration process, the user exits you created earlier are moved to the bottom of the maintenance dialog's source area. Move each user exit code block to the appropriate location in code. This should be an easy task if you have named the user exits after the code blocks in which they belong.

Making a Copy Before You Regenerate

If many changes have been made to your data design, or other changes on the server have had an impact on your dialog, decide whether to implement the changes by hand or to generate a new copy of your form. If you generate a new copy of the form, you must re-implement any tailoring you have done. This decision depends on which approach represents less work for you.

If you decide to generate a new copy of your dialog, save your old dialog with a different name. You can view the old dialog while tailoring the new dialog. Additionally, you can cut and paste code from one dialog to the other.

Customizing on the Server

This section describes the mechanisms available on the server for customizing your maintenance dialog.

Deriving Variable Names

When performing customizations to a maintenance dialog, it is useful to understand how variable names are derived. This will help you maintain a consistent naming convention and make it easier for you to determine what the code is doing.

Deriving GUI Control Names

GUI control names are made up of three components: a GUI Control Identifier, an Object Identifier, and a Field Identifier. Each one is separated by underscores. For example, a field called CUSTOMER-NUMBER on a Customer file might be represented by a TextBox GUI control named txt_CUST_CustomerNumber.

GUI Control Identifier

A GUI control identifier is a three-character abbreviation in the GUI control name that uniquely identifies the GUI control type. The following table lists the different types of GUI controls (along with their abbreviations) that are used in a typical Construct Spectrum project:

GUI Control	Abbreviation	GUI Control	Abbreviation
CheckBox	chk	Label	lbl
ComboBox	cbo	ListBox	lst
CommandButton	cmd	Menu	mnu
Form	frm	OptionButton	opt
Frame	fra	StatusBar	sta
Grid	grd, ddg	TextBox	txt

Object Identifier

An object identifier is a four-character abbreviation that uniquely identifies the business object represented on the dialog. The object identifier is obtained from the Abbreviated Object Description parameter of the VB-Maint-Dialog model. By default, this value contains the first four characters of the dialog form (.frm file) name. Using the Object Identifier as a component of the GUI control name is useful if you want to represent more than one business object on a single dialog.

Field Identifier

A field identifier uniquely identifies a field within a business object. The name is derived from the Predict field name — converting the letters to mixed case and removing any characters which are illegal in Visual Basic, such as hyphens. The field identifier for grid controls that are derived from intra-object relationships are obtained from the Predict relationship name.

Deriving Label Captions for GUI Controls

A label caption is a name that identifies a GUI control to the user. The label caption is usually displayed to the left of an associated input GUI control, for example, a text box. The caption for the label is obtained from one of two places. First, the model looks for header information stored in Predict's Elementary Field definition. If none is found, the label caption is derived from the field name in the same way the field Identifier is created. Label captions for grid controls that are derived from intra-object relationships are obtained from the Predict relationship name.

Overriding GUI Controls

The VB-Maint-Dialog model must choose the appropriate GUI control to represent your field as it is defined in Predict. This includes representing complex data, such as one-to-many relationships. To accomplish this, the model employs derivation logic based on information such as a field's data type, the number of occurrences, whether it is in a repeating group of fields, etc. The following steps in this section describe the derivation logic. Each topic is included in the same order in which the logic is applied by the model.

In addition to this default derivation logic, the model provides several mechanisms for you to override the default selection of a GUI control for a given field. These are described in steps 1, 2, and 3 of this section.

Note: An asterisk () appended to any GUI control name in this section indicates that the GUI control could also apply to a column of a grid, depending on the cardinality of the associated field. Therefore, TextBox* can be read as TextBox or TextBoxColumn. For more information about using GUI controls with grid columns, see **Using the Grid**, page 202.*

Step 1 — Search for GUI Keywords on Field Definitions

The model starts by looking for specific keywords that begin with GUI on the Predict field definition. The following example shows a hypothetical M-PROVINCE field being mapped to a ComboBox using the GUI_COMBOBOX keyword:

```

12:53:21          ***** P r e d i c t  3.4.1  *****          99-01-28
                    - Modify Field -

Field ID ..... M-PROVINCE          Modified: 97-01-16 at 09:32
File ID ..... NCST-CUSTOMER        by: DEVMT1
Keys .. GUI_COMBOBOX                Zoom: N

Ty L Field name          F Length  Occ  D U DB N NAT-1
* - - - - - * - - - - -
  2 M-PROVINCE          A  20.0          X4 N

Natural attributes
Header1 .... Province
Header2 ....
Header3 ....
Edit mask ..
Comments   Zoom: N

EDIT:  Owner: N  Desc: N * Veri: N          MORE  Attr.: N
    
```

Predict Modify Field Panel

The model recognizes the following keywords:

GUI Control	Description
GUI_ALPHA MULTILINE	<p>Generates a TextBox* control with the MultiLine property set to True. This gives the GUI control the feel of a mini-word processor. The control will word-wrap its contents and provide scroll bars as required.</p> <p>Use this keyword to represent a repeating alphanumeric field as a single piece of information such as a long description.</p>
GUI_CHECKBOX	<p>Generates a CheckBox* control. This keyword can be used in combination with a field of any format. If a table verification with two or more values is attached to the data field, the first value represents false and the second value represents true. If no verification is attached to the field, the model derives true and false values based on the field's format. If the field is alphanumeric, blank represents false and non-blank represents true.</p> <p>When updating the object PDA, the VB maintenance dialog uses "X" to represent true. If the field is numeric, zero represents false and non-zero represents true. When updating the object PDA, the maintenance dialog uses 1 to represent true.</p>
GUI_COMBOBOX	<p>Generates a drop-down ComboBox* control. This model looks for a table-style verification. If one has been set up, the values are used as the entries for the combo box. If a verification does not exist, the model generates one dummy entry for the combo box.</p> <p>Generate a dummy entry if the combo box is to be populated with data from an external source such as a PC on your LAN. For information about populating a combo box with external data, see Generating a ComboBox Control to Display External Values, page 164.</p>

GUI Control	Description (continued)
GUI_NULL	Prevents the generation of a GUI control definition for the field or any code pertaining to the field. Use this keyword if you defined fields that should not be displayed on the dialog.
GUI_OPTION BUTTON	Generates a frame and a series of <code>OptionButtons</code> . The model uses the table-style verification attached to the field. For this keyword to work, you must attach values to the table-style verification because each of the values maps to an option button.
GUI_PROTECTED	Treats the associated field as read-only. The user cannot modify the contents of the field. This keyword can be used in conjunction with the other keywords described in this section except when the <code>GUI_NULL</code> keyword is used. Use this keyword if the contents of the field is to be determined programmatically, as with a calculated field. For more information about calculated fields, see Creating Calculated Fields , page 175.
GUI_TEXTBOX	Generates a <code>TextBox</code> *. Text box GUI controls can have BDT (business data types) definitions attached to them. For more information about using BDTs with text box GUI controls, see Step 3 — Search for Business Data Type Keywords on Field Definitions , page 166.

Note: Option buttons are not supported in a grid control. If the `GUI_OPTIONBUTTON` keyword is attached to the field definition and the field is part of a repeating group of fields (PE) or is a stand-alone repeating field (MU), it is mapped to a `ComboBox` instead of `OptionButtons`.

Generating a ComboBox Control to Display External Values

Use the GUI_COMBOBOX keyword in Predict to force generation of a ComboBox control that displays values from an external source (for example, a LAN database).

- To set up a ComboBox control to display values from an external source:
 - 1 Set up a field definition for the field in Predict.
 - 2 Add the GUI_COMBOBOX keyword to the Predict field definition.
 - 3 On the client, write code in the Form_Load event for the dialog to populate the ComboBox with values by reading the external source when the form is loaded.

Step 2 — Search for GUI Keywords on Verification Definitions

If the model did not derive a GUI control in Step 1, it looks next for a GUI keyword on any attached table-style verifications. However, it only considers the GUI_COMBOBOX and GUI_OPTIONBUTTON keywords as valid. Other keywords are ignored.

The following example shows a hypothetical VALID-PROVINCE verification being mapped to a ComboBox using the GUI_COMBOBOX keyword:

```

13:12:21          ***** P r e d i c t  3.4.1  *****          99-01-28
                    - Modify Verification -
Verification ID . VALID-PROVINCE          Modified: 97-01-28 at 13:11
Status ..... Natural Construct          by: DEVMT
Keys .. GUI_COMBOBOX          Zoom: N

Format .....* A Alphanumeric          Modifier   Zoom: N
Type .....* T Table of values
Message nr ..... 1112
Replacement 1 ...
Replacement 2 ...
Replacement 3 ...
Message text ....

Comments      Zoom: N          Values * Zoom: N
British Columbia      BC
Alberta              ALTA
Saskatchewan         SASK
Manitoba             MAN
Ontario              ONT
Quebec               QC
New Brunswick        NB
EDIT:  Owner: N  Desc: N * Rule: N

```

Predict Modify Verification Panel

Tip: Improve the readability of a verification value by adding its concise term in the Comments field. Construct Spectrum displays the comment value in the drop-down combo box or caption name of an option button. In the previous panel, the full name of each province has been entered in the Comment field that corresponds to its database verification value. If comment values are not supplied, the database verification values are displayed.

Consider attaching a GUI keyword to a verification definition, rather than a field definition, to implement a standard GUI representation for any field using the same type of verification. This also eliminates the need to assign the keyword to each field definition. You can override the GUI keyword on the verification definition by supplying one for the field definition.

For more information about these keywords, see the description for the GUI_OPTIONBUTTON and GUI_COMBOBOX keywords in **Step 1 — Search for GUI Keywords on Field Definitions**, page 161.

For more information about verifications, see **Validating Your Data**, page 327.

Step 3 — Search for Business Data Type Keywords on Field Definitions

If the model could not derive a GUI control in Step 1 or 2, it next looks for a Business Data Type (BDT) keyword in the Predict field definition. This panel is shown in Step 1. Since you can augment the standard set of Spectrum supplied BDTs with your own BDTs, the model will accept any keyword which begins with BDT.

If the model finds a BDT-prefixed keyword on the field definition, it uses a Text-Box* GUI control to represent the field. Additionally, the model looks in the keyword comments for an actual BDT type and modifier. If a BDT exists in the comments, it is used.

Example of a BDT type with a modifier specified in the keyword comments

```
BDT=BDT_NUMERIC  
MOD="ZERO=OFF"
```

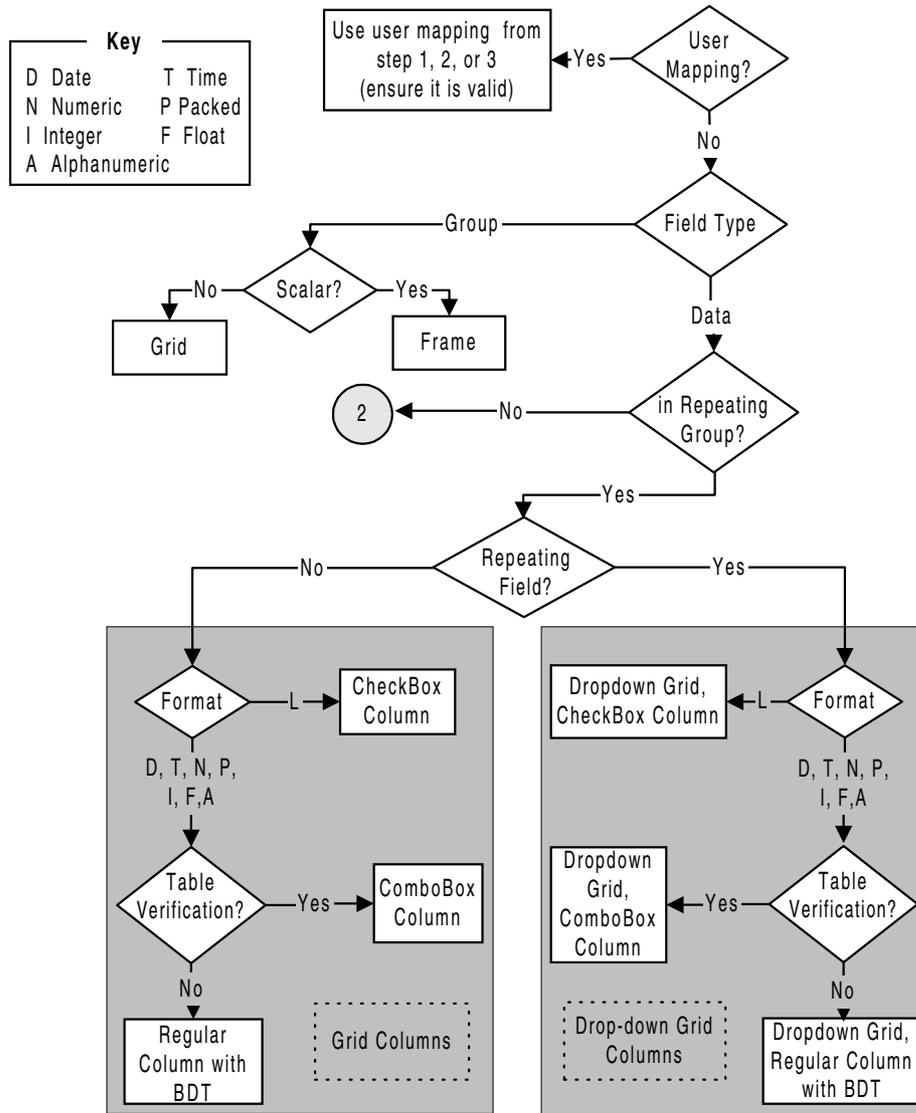
If no BDT or modifier is found, the model uses the BDT implied by the keyword itself. If no modifier was specified with the BDT, the BDT manager in the Construct Spectrum client framework defaults a modifier.

You can create your own BDT keywords which only exist on the server and map them to combinations of BDTs and modifiers on the client PC. For example, you could create two BDT keywords, BDT_NUMERIC_ZERO and BDT_NUMERIC_ROUND.

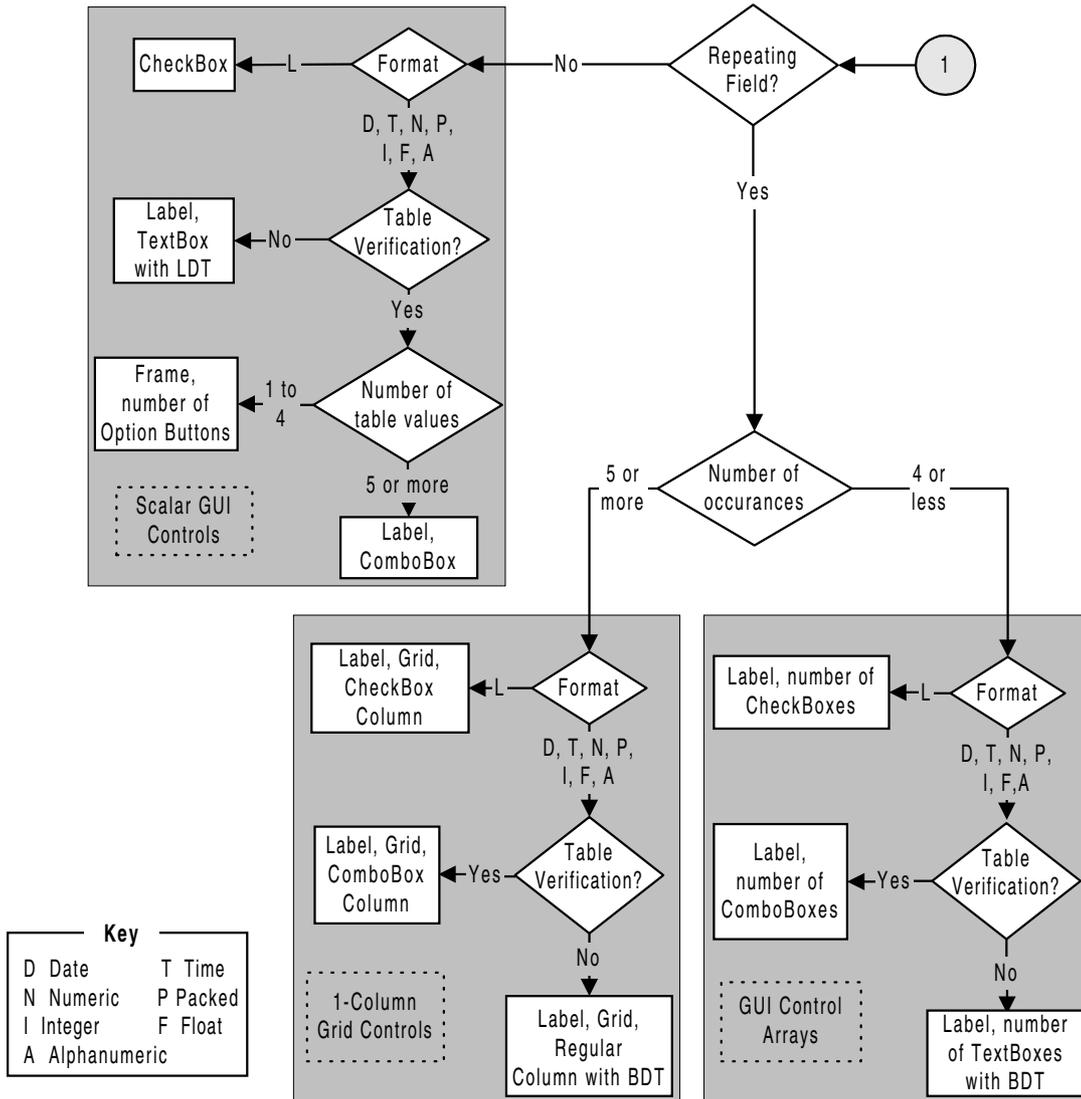
- Attach the BDT_NUMERIC_ZERO keyword to the field definition
The comments of the BDT_NUMERIC_ZERO keyword could contain BDT=BDT_NUMERIC and MOD="ZERO=ON".
- Attach the BDT_NUMERIC_ROUND keyword to the field definition
The comments of the BDT_NUMERIC_ROUND keyword could contain BDT=BDT_NUMERIC and MOD="ROUND=ON".

Step 4 — Use Default Derivation

If the model has been unable to derive a GUI control in Step 1, 2, or 3, it uses its built-in GUI derivation logic. This logic is best described pictorially in the following diagram.



Default Derivation of GUI Control - Part 1 of 2



Default Derivation of GUI Control - Part 2 of 2

The previous diagram illustrates that the choice of GUI control(s) used to represent a database field depends on several threshold variables. You can control these threshold points at a corporate level; that is, your default threshold values affect all VB Maintenance Dialogs. This is accomplished by using Construct's corporate defaulting mechanism.

The corporate defaults that affect Construct Spectrum's choice of a GUI control are described in the following sections:

- Repeating Field Threshold
 - Option Button Threshold
 - Foreign Field Threshold
- To assign a corporate default:
- Use the following code example as a guide to assigning a corporate default value. The example illustrates how a work file number and column delimiter values are defaulted.

Example of assigning corporate defaults

```

/*
/* Retrieve all model constants that are stored using the standard
/* defaulting method.
INCLUDE CCDEFLTN '''MAX-OPTION-BUTTON-COUNT'''
                'CUMDPDA.#MAX-OPTION-BUTTON-COUNT'
INCLUDE CCDEFLTN '''MAX-MU-COUNT''' 'CUMDPDA.#MAX-MU-COUNT'
INCLUDE CCDEFLTN '''MAX-DIALOG-WIDTH'''
                'CUMDPDA.#PDA-MAX-DIALOG-WIDTH'
INCLUDE CCDEFLTN '''MAX-DIALOG-HEIGHT'''
                'CUMDPDA.#PDA-MAX-DIALOG-HEIGHT'
INCLUDE CCDEFLTN '''FK-AS-COMBO-THRESH-HOLD'''
                'CUMDPDA.#PDA-FK-AS-COMBO-THRESH-HOLD'
** Note that there are 3 separate INCLUDE members: one for numeric
** defaults (CCDEFLTN), one for alphanumeric defaults (CCDEFLTA), and
** one for logical defaults (CCDEFLTTL)
** Continue normal processing and the initial values may have been
** overridden by a corporate-supplied defaulting routine.

```

To apply the changes corporation-wide, you must add the initial variable name and its initial value in the CSXDEFLT user exit routine.

Note: The internal defaulting mechanism may be affected when you use this defaulting mechanism to initialize the specification panel default keyword. Use the same keyword for both mechanisms. The specification panel default keyword overrides the internal default keyword.

Repeating Field Threshold

A repeating field that is not in a repeating group of fields is represented either by a GUI control array, such as an array of TextBoxes, or by a one-column Grid control.

The choice of GUI control depends on the MAX-MU-COUNT default value. If the number of occurrences of a repeating field is less than or equal to MAX-MU-COUNT, the field will be represented with a GUI control array.

The VB Maintenance Dialog model copies the MAX-MU-COUNT default value into the #MAX-MU-COUNT variable of the model PDA (CUMDDPA) in the model's pre-generation subprogram (CUMDPR).

Option Button Threshold

A scalar field that has a table verification attached to it is represented either by a Frame and series of OptionButtons or by a Label and ComboBox.

The choice of GUI control depends on the MAX-OPTION-BUTTON-COUNT threshold default value. If the number of table verification values is less than or equal to MAX-OPTION-BUTTON-COUNT, the field will be represented with a Frame and OptionButtons.

The VB Maintenance Dialog model copies the MAX-OPTION-BUTTON-COUNT default value into the #MAX-OPTION-BUTTON-COUNT variable of the model PDA (CUMDDPA) in the model's pre-generation subprogram (CUMDPR).

Foreign Field Threshold

If a scalar field represents a foreign field in another file, the maintenance dialog provides additional GUI controls to allow the selection of these foreign values. The maintenance dialog will either provide a button that opens a modal browse dialog or generate a ComboBox and populate it at form-load time.

The choice of GUI control depends partially on the FK-AS-COMBO-THRESH-HOLD default. If the number of foreign key values is less than or equal to FK-AS-COMBO-THRESH-HOLD, the field is represented with a ComboBox.

The VB Maintenance Dialog model copies the FK-AS-COMBO-THRESH-HOLD default value into the #PDA-FK-AS-COMBO-THRESH-HOLD variable of the model PDA (CUMDPDA) in the model's pre-generation subprogram (CUMDPR).

For more information about how foreign fields are represented with GUI controls, see **Understanding the Browse and Maintenance Integration**, page 343.

Setting Generation GUI Standards

Construct generation technology enables you to standardize your code. Construct Spectrum extends the benefits of standardization to the GUI realm. Default values for properties of GUI controls, such as Font and ForeColor, are centrally established. This means that if your company standard is to use a 10 pt. Arial font for all labels on GUI screens, you need only change one line of code.

Construct Spectrum uses a series of utility Natural subprograms to control generation of GUI dialogs. Collectively, these subprograms are known as the Visual Basic API. For each type of GUI control supported, there is a property default subprogram which is responsible for supplying default properties for that GUI control. The Visual Basic API always calls the property default subprogram for a GUI control before generating the definition for the GUI control. For example, the Visual Basic API callnats CSVBDLBL, the property default subprogram for Label GUI controls, before generating the definition for a label. This subprogram sets the default Height of a Label with the following line of code:

```
ASSIGN CSVALCTN.HEIGHT = 285
```

Following is a list of all the GUI controls supported by Construct Spectrum and the associated property default subprogram for the GUI control.

GUI Control	Subprogram	GUI Control	Subprogram
CheckBox	CSVBDCHK	ComboBox	CSVBDCBO
CommandButton	CSVBDCMD	Form	CSVBDFRM

GUI Control	Subprogram	GUI Control	Subprogram
Frame	CSVBDFRA	Grid	CSVBDGRD
Label	CSVBDLBL	ListBox	CSVBDLST
OptionButton	CSVBDOPT	StatusBar	CSVBDSTA
TextBox	CSVBDTXT	Timer	CSVBDTMR

You can change the default assignments made in any of the property default subprogram standards through the Generated Maintenance classes API. For information about changing these defaults, see **Utility Subroutines on the Client**, page 477, in *Construct Spectrum Reference Manual*.

Note: Some properties, such as Top, Left, and Caption, are dependent on the data field associated with the GUI control or the field's relative position in a Predict file. Do not attempt to provide standards for this type of control. The model controls the values for this type of property and will override any changes you specify.

Controlling the Size of a Maintenance Dialog

You can control the maximum dimensions of generated dialogs by specifying corporate default values. Generated dialogs will not exceed these dimensions. The maximum height and width values are supplied in a unit of measurement known as TWIPS. The following table shows the TWIP value equivalent of pixels for common monitor resolutions.

Resolution In Pixels	TWIPS - Small Fonts (factor of 15)	TWIPS - Large Fonts (factor of 12)
640 x 480	9600 x 7200	7680 x 5760
800 x 600	12000 x 9000	9600 x 7200
1024 x 768	15360 x 11520	12288 x 9216
1280 x 1024	19200 x 15360	15360 x 12288

Know the lowest resolution monitor your application will be used on and generate dialogs to fit that monitor. You can set default values for the maximum height and width of your dialog by using Construct's corporate defaulting mechanism. The default values are MAX-DIALOG-HEIGHT and MAX-DIALOG-WIDTH.

The VB-Maintenance-Dialog model obtains these defaults in its pre-generation subprogram (CUMDPR) and copies them into the #PDA-MAX-DIALOG-HEIGHT and #PDA-MAX-DIALOG-WIDTH variables of the model PDA (CUMDPDA).

For information about changing a corporate default value, see **Step 4 — Use Default Derivation**, page 167.

Overflow Conditions

Overflow conditions occur when a dialog cannot display all of its controls. Consider the following scenario. You are developing an application on a monitor with a resolution of 9600 x 7200 TWIPS and you generate a dialog that reaches a height of 10000 TWIPS. When you open the dialog in the Visual Basic editing environment, a third of the GUI controls extend off the bottom of the screen. This is known as an *overflow condition*. The only way to work with the hidden GUI controls is to select the control from the Properties panel and manually manipulate their Left and Top properties — not a visual solution! For information about correcting overflow conditions, see **Working with Overflow Frames**, page 179.

Customizing on the Client

This section describes the different mechanisms available on the client platform for customizing the generation results of a Visual Basic maintenance object and a maintenance dialog.

Creating Calculated Fields

Creating GUI controls whose values are based on the values of other GUI controls is a common customization task. This task involves modifications to both the maintenance dialog and the Visual Basic maintenance object.

For information about deriving values from a foreign field on a maintenance dialog, see **Supporting Multiple Descriptive Values and Derived Values**, page 360.

Does a GUI Control Exist for the Calculated Field?

The first step in creating a calculated field is to ensure that a GUI control exists on the maintenance dialog to hold the calculated value. If the field is defined in Predict, it will already exist on the dialog. Make sure that the control is not enabled. If the control is a scalar GUI control — such as a TextBox or ComboBox, set the control's Enabled property to False. If the control is a grid, modify the code in the *LoadGridNameGrid* (where *GridName* is a unique variable) routine.

Tip: In Predict, add the GUI_PROTECTED keyword to a calculated field. This keyword can be added to both input and output-only fields.

If a GUI control does not exist to hold the calculated value and it will not be stored in the database, add the GUI control by hand. For information about adding a GUI control by hand, see **Adding a New Field by Hand**, page 185.

Coding the Calculation

The calculation must be triggered whenever the value of one of the fields involved in the calculation changes. Use the LostFocus event to trigger such a calculation.

Note: The calculation should not be performed in the dialog code. Keep customized code in the dialog to a minimum. Rather, add the calculation code to the Visual Basic maintenance object. The function call might look similar to the following example.

Example of a function call in the maintenance dialog

```
txt_Empl_Pay.Text = InternalObject. _  
    Calc_Pay(CLng(txt_Empl_Rate.Text), CLng(txt_Empl_Hours.Text))
```

The function can also accept the parameters required to perform the calculation and return the result, such as in the following example.

Example of calculation code in the Visual Basic maintenance object

```
Public Function Calc_Pay(Rate As Long, Hours as Long) As Currency  
    Calc_Pay = Rate * Hours  
End Function
```

Integrating Maintenance and Browse Functions

When a foreign key field is included in a Predict defined file and you generate a maintenance dialog for the file, Construct Spectrum automatically includes browse capabilities for the foreign field. A browse linked to a maintenance dialog can be implemented as a drop-down list or as a dialog.

For more information about how maintenance and browse functions are integrated, see **Understanding the Browse and Maintenance Integration**, page 343.

Validating Data Using the Visual Basic Maintenance Object

The Visual Basic maintenance object is an ideal place to code simple business validations. The model provides the CLIENT-VALIDATION user exit for this purpose. Coding validations on the client reduces the number of data entry errors on your dialog before the data is transmitted across the network, thus enhancing the overall performance of your application. Avoid coding validations in the Visual Basic maintenance object that involve network calls; these could trigger a network call every time you change focus from one field to the next. For more information about validating your data, see **Validating Your Data**, page 327.

Tailoring the Maintenance Dialog

This section describes how to tailor your maintenance dialog. This section contains information about tailoring the dialog's appearance, adding and removing fields, and working with special types of fields.

The most common tailoring task you will encounter is altering the layout of GUI controls as they were generated on the dialog. By default, GUI controls are generated in two columns from top to bottom, with labels on the left and input controls on the right. Following is an image of a typical maintenance dialog as it would appear after generation.

Order Maintenance (123131)

Order Number: 123131

Order Amount: 8350.00

Order Date: 04/14/97

Customer Number: 10001 JOURNEYMEN FABRICATING

Warehouse Id: 111 TORONTO CENTRAL WAREHOUSE

Invoice Number: 231201

Delivery Instructions: Must be delivered between 1 and 6 PM.

Product:

	Product Id	Line Description	Quantity	Unit Cost	Total Cost
1	3245	COOPER GLOVES	40	100.00	4000.00
2	6220	SOYA FLOUR	100	43.50	4350.00
3			0	0.00	0.00

Distribution (1):

	Cost Center	Acct	Project	Dist Amount
1	TC	100	05	50.00
2	TQ	105	05	50.00
3				0.00
4				0.00

Typical Generated Maintenance Dialog

When tailoring the dialog's appearance, the changes should enhance the usability of the application. For example, group related fields so the user can easily see that they are related. The user should be able to move from field to field in a way that coincides with how they would logically perform their tasks.

There are many reasons why you would alter the appearance of your dialog, such as conforming to layout standards used by your organization. For example, there may be users whose monitor resolution is 640 by 400 pixels. Your organization wants all of your applications to run effectively on these users' machines. For more information about generating dialogs based on a specific monitor resolution, see **Controlling the Size of a Maintenance Dialog**, page 173.

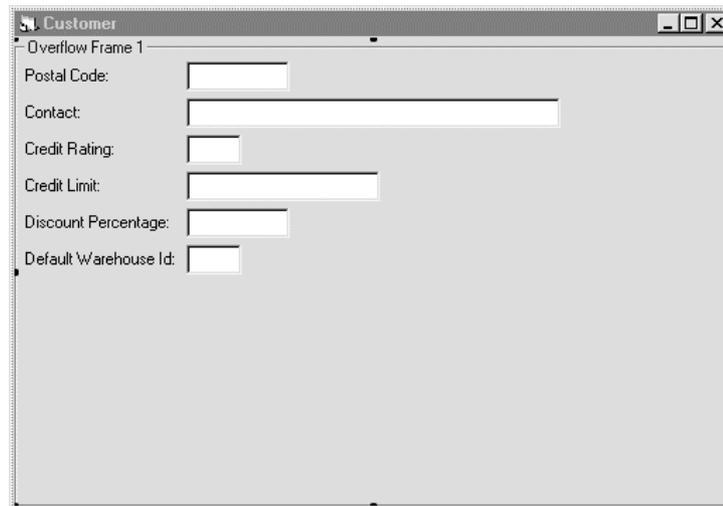
Following are several suggestions on how to lay out your maintenance dialogs so that they meet your organization's requirements. Each suggestion contains a diagram that depicts the layout. Each diagram is based on layout changes that were applied to the generated maintenance dialog shown previously.

Note: Many of the procedures described in this section require you to perform tasks specific to Visual Basic. For more information, see the documentation that comes with Visual Basic.

Working with Overflow Frames

Overflow conditions occur when a dialog cannot display all of its controls. When an overflow condition is encountered by the VB-Maint-Dialog model, it responds by generating a Frame control that is the same size as the dialog itself. The frame overlays the other GUI controls on the dialog. The model then continues generating new GUI controls in the Frame container control. If the first frame becomes filled, the model generates another frame. The process continues until all the fields in the Predict file are represented by a GUI control.

The following example shows what a dialog looks like when an overflow condition is encountered:



Dialog Overflow Conditions

When a dialog generates overflow frames, rearrange the GUI controls using one of the layouts described in this section. This will largely be a job of cutting and pasting GUI controls from the overflow frame(s) onto the dialog itself.

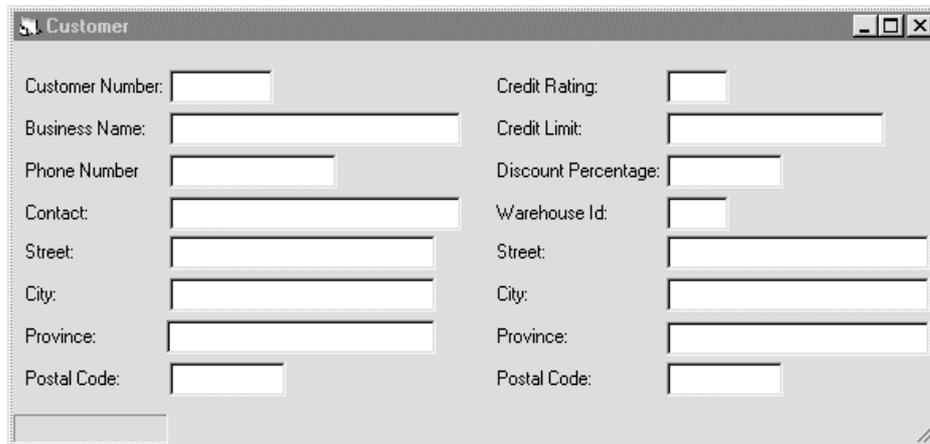
- To work with an overflow frame:
 - 1 Open the Construct Spectrum project that contains the dialog form (.frm) file you want to modify.
 - 2 On the Project Window, select the dialog form and then select View Form. The dialog is displayed.
 - 3 Make the dialog as large as you can and drag the frame to a free area of the screen. All of the controls within the frame are moved as well. If the frame is blocking your view of other controls, shrink the size of the frame.
 - 4 Rearrange the GUI controls using one of the layouts described in the following sections.

Multi-column Layout

Use a multi-column layout when your dialog contains a large number of fields. For example, if a dialog will be too long and can be wider.

- To create a multi-column layout:
- 1 Drag some of your GUI controls over to create a second column of information.
 - 2 If one or two fields are significantly wider than others and are impeding your attempts to create a second column, consider shrinking the width of these controls. Users can still type in large data values although they cannot see the entire value in the field.

The following example shows the same maintenance dialog presented at the beginning of this section, this time in a two-column layout:



The screenshot shows a window titled "Customer" with a two-column layout of input fields. The fields are arranged as follows:

Field Name	Field Name
Customer Number	Credit Rating
Business Name	Credit Limit
Phone Number	Discount Percentage
Contact	Warehouse Id
Street	Street
City	City
Province	Province
Postal Code	Postal Code

Maintenance Dialog in a Multi-Column Layout

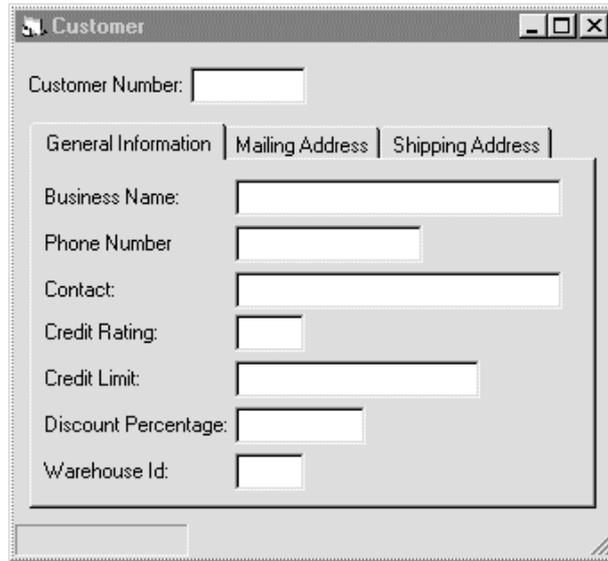
Tabbed Layout

If your dialog is larger than you would like and there is not sufficient room to create multiple columns, consider placing some or most of the GUI controls inside tab pages.

- To create a layout with tab pages:
- 1 Using the Sheridan tab which comes with Visual Basic Professional Edition, move GUI controls to a tab page by cutting them from the dialog (or overflow frame)
 - 2 Select the tab control and paste the GUI controls onto the tab.
You can now drag them to the appropriate location.

Tip: To place a group of GUI controls on the same tab page, cut and paste all the controls at the same time. The GUI controls will maintain their position relative to one another. In general, do not place key field(s) on a tab page. Key field(s) should always be visible and easily accessible.

The following example shows the same maintenance dialog presented at the beginning of this section, this time in a tabbed layout:



Maintenance Dialog with Tab Pages

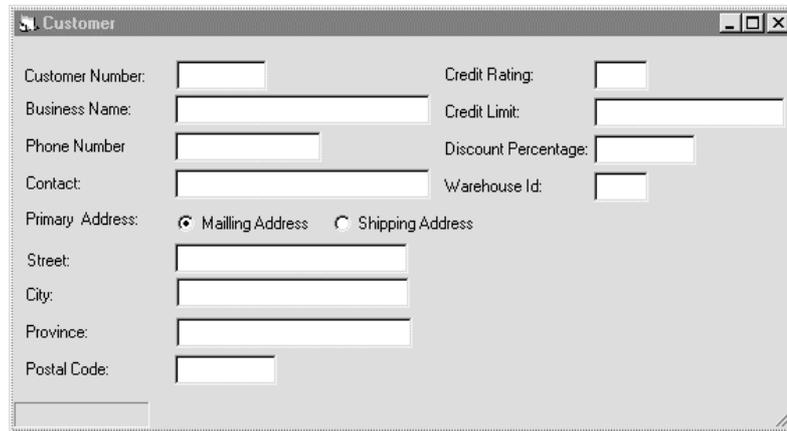
State-Dependent Layout

A state-dependent layout is the most difficult type of layout to create. Use this layout when many of the fields on the dialog are mutually exclusive (displayed and enabled only when the dialog is in a specific state).

- To create a state-dependent layout:
 - 1 Add a State field to the dialog.
This field is always visible and controls the current state.
 - 2 Assign other fields on the dialog to a specific state.
 - 3 Move the GUI controls on the screen so that fields belonging to one state overlap those in the other states.
 - 4 Write code to make the fields from one state visible and the fields from the other states invisible whenever the state field changes.

Tip: It may be easier to create two frames and place the state-dependent fields inside the frames. Make the frames visible or invisible depending on the current state.

In the following example, there is a new GUI control called Address Toggle; its label is Primary Address. This GUI control is the State field. It controls when to display Mailing Address information and when to display Shipping Address information:



The screenshot shows a window titled "Customer" with standard window controls (minimize, maximize, close). The form contains the following fields and controls:

- Customer Number:
- Business Name:
- Phone Number:
- Contact:
- Primary Address: Mailing Address Shipping Address
- Street:
- City:
- Province:
- Postal Code:
- Credit Rating:
- Credit Limit:
- Discount Percentage:
- Warehouse Id:

Maintenance Dialog with a State-Dependent Layout

Adding a New Field by Hand

If you add a new field to your Predict file definition and have already generated a dialog, it may be more efficient to manually add the new field to your dialog rather than regenerating the dialog. This is especially true if you have already tailored the generated form.

The tasks required to add a new field to a dialog by hand vary depending on the cardinality of the field (whether the field can display one, two, or more dimensions of information). One-dimensional information is displayed within a scalar GUI control. Information with two, three, or four dimensions is displayed either in a column in a grid control or in a control array such as an array of text boxes.

Adding a Scalar Field by Hand

A scalar GUI control represents one-dimensional information. Most controls on a dialog are scalar; for example, a name, an address, or an order number are typically represented with scalar fields.

- To add a scalar field:
- 1 Determine the type of GUI control to represent your new database field. GUI controls for scalar data include:
 - TextBox
 - ComboBox
 - CheckBox
 - Frame with a group of OptionButtons.
 - 2 Drag the desired type of GUI control onto the dialog from the Visual Basic toolbox.
 - 3 Add a label and GUI control input name for the control.

Tip: Choose your names based on the naming conventions used by other Construct Spectrum GUI controls.
For information about Construct Spectrum naming conventions, see **Deriving Variable Names**, page 159.

- 4 Follow the instructions provided in this section for the type of control you are adding. These procedures contain information about creating event code blocks for the new control and about adding code to some standard subroutines to implement the control.

A ComboBox control utilizes a single drop-down list from which users can select a value. The user cannot, however, type additional values in the list. The client framework includes the `ComboClass.cls`, which is useful for populating ComboBox GUI controls and Combo columns of a grid. The `ComboClass` allows you to define pairs of values: a database value and a display value.

If the new database field is a repeating field (MU field), create a control and use the same techniques described in this section. Ensure that the code blocks use an Index parameter. Control arrays are zero-based whereas array information stored in the Object PDA is one-based.

- To add a TextBox GUI control for a field:

- 1 Add a new assignment statement to the `CopyObjectToForm` subroutine. This copies the Object PDA field value to the GUI control. The following code is a sample assignment statement:

```
txt_CUST_NewField.Text =
    BDT.ConvertToDisplay(.Field("NEW-FIELD"), _
        NatFormatLength:="A6")
```

- 2 Add a new case statement to the `CheckRemoteError` subroutine. This statement enables the dialog to assign an error object to the field if the maintenance object subprogram on the server encountered a validation error for the field. The following code is a sample case statement:

```
Case "NEW-FIELD": Set ErrControl = txt_CUST_NewField
```

- 3 Add Change event code for the new GUI control. This code indicates to the dialog that the value of the field has changed. It also indicates that at least one field in the business object has changed. The following code is a sample Change event:

```
Private Sub txt_CUST_NewField_Change()
    ValueChanged = True
    ObjectChanged = True
End Sub
```

- 4 Add GotFocus event code for the new GUI control.
This code displays an error tip for the field if there is an object error attached to the field. The following code is a sample GotFocus event:

```
Public Sub txt_CUST_BusinessName_GotFocus()
    ValueChanged = False
    cstSelectContents
    CSTUtils.cstDisplayErrorTip Me
End Sub
```

- 5 Add LostFocus event code for the new GUI control.
If the user changed the value of the control, this code removes any object errors from the control and assigns the control's value to the field in the Object PDA. If an error was detected during the assignment, an object error is applied to the control. The following code is a sample LostFocus event:

```
Public Sub txt_CUST_NewField_LostFocus()
    Dim Value As String
    CSTUtils.ErrorTip.HideErrorTip
    If ValueChanged Then
        ErrorMessage = ""
        RemoveUnneededControlErrors Me, _
            txt_CUST_NewField, ValueChanged
        Value = txt_CUST_NewField.Text
        ValidAssignment Value, InternalObject, _
            "NEW-FIELD", ErrorMessage, NatFormatLength:="A6"
        txt_CUST_NewField.Text = Value
        If ErrorMessage <> "" Then
            ParseErrorString ErrorMessage, ErrorNr, ErrorSrc
            SetObjectError Me, txt_CUST_NewField, ErrorNr, _
                ErrorMessage, ErrorSrc
        End If
    End If
End Sub
```

- To add a ComboBox GUI control for a field:

- 1 Add code to the Form_Load event to load and initialize a ComboClass instance with the valid values. The following code is a sample load/initialize statement:

```
ProvList.Load cbo_CUST_Prov
ProvList.AddItem "British Columbia"
...
```

Note: If you are loading values from an external source such as a PC connected to your LAN, code the necessary logic to load these values now.

- 2 Add code to the CopyObjectToForm subroutine to update the ComboBox with values.
The update is accomplished by assigning a value from the ComboClass.cls to the ListIndex property of the ComboBox control. The following is a sample statement to update the ComboBox with values:

```
cbo_CUST_Prov.ListIndex = ProvList.GetIndex(.Field("PROV"))
```

- 3 Add code to update the business object when the selected value of the ComboBox is changed, as occurs when a Click event is triggered.
The following is a sample statement executed on the client to update the business object with a new database value:

```
Value = _
    ProvList.DBValue(cbo_CUST_Prov.ItemData(cbo_CUST_Prov.ListIndex))
ValidAssignment Value, InternalObject, "PROV", ErrorMessage, _
    NatFormatLength:="A20"
```

For more information about using the ComboClass, see **Maintenance Classes**, page 249, in *Construct Spectrum Reference Manual*.

- To add a CheckBox field:

Note: The sample code for this procedure assumes that the new database field is Alphanumeric.

- 1 Add a new assignment statement to the CopyObjectToForm subroutine.
This copies the object PDA's field value to the GUI control. The following code is a sample assignment statement:

```
chk_CUST_NewField.Value = IIf(.Field("NEW-FIELD") <> "", _
    vbChecked, vbUnchecked)
```

- 2 Add a new case statement to the CheckRemoteError subroutine.
This statement enables the dialog to assign an error object to the field if the object maintenance subprogram on the server encountered a validation error for the field. The following code is a sample case statement:

```
Case "NEW-FIELD": Set ErrControl = chk_CUST_NewField
```

- 3 Add Click event code for the new GUI control.
The functions performed in the Click event are to indicate that the field value has changed, remove any object errors from the control, assign the new value to the Object PDA (client's version), set an object error for the control if an error was encountered during the assignment, and finally display an error tip if an error is attached to the control. The following code is a sample Click event:

```
Private Sub chk_CUST_NewField_Click()  
    Dim ErrorMsg As String  
  
Public Sub chk_CUST_NewField_GotFocus()  
    ValueChanged = False  
    cstSelectContents  
    CSTUtils.cstDisplayErrorTip Me  
End Sub  
  
    End If  
    NewFieldNdx = Index  
    CSTUtils.cstDisplayErrorTip Me  
End If  
End Sub
```

- 4 Add GotFocus event code for the new GUI control.
This code displays an error tip for the field if there is an object error attached to the field. The following code is a sample GotFocus event:

```
Public Sub chk_CUST_NewField_GotFocus(Index As Integer)  
    ValueChanged = False  
    cstSelectContents  
    CSTUtils.cstDisplayErrorTip Me  
End Sub
```

Adding a Regular Grid Column for a Field

Grid controls are used to represent two, three, or four-dimensional information. If the field you are adding is part of a grid, you must perform modifications to the column indexing values of some of the grid variables. For information about manipulating grid controls, see **Using the Grid**, page 202.

Each column within a grid is associated with a database field. The grid code must know the relative position of a column to identify its associated database field. Therefore, when adding a grid column, you must adjust the column indices in the dialog code as described in the following steps.

- To add a Regular Grid Column for a field:
- 1 In the Global Declarations section, increase the MAX_GridName_COLS constant by one.

Sample code before

```
Public IncomeGrid As New TrueGridClass
Const MAX_NCSTORDERHASLINES_COLS = 8
Const MAX_NCSTORDERHASLINES_ROWS = 30
```

Sample code after

```
Public IncomeGrid As New TrueGridClass
Const MAX_INCOME_COLS = 9
Const MAX_INCOME_ROWS = 30
```

- 2 In the CheckRemoteError sub section, increase the ErrColumn value in the case statement for every field in the same grid with a higher column number than the new column.

Sample code before

```
Case "SALARY"
    Set ErrControl = IncomeGrid
    ErrColumn = 4
Case "BONUS"
    Set ErrControl = IncomeGrid
    ErrColumn = 5
```

Sample code after

```
Case "SALARY"
    Set ErrControl = IncomeGrid
    ErrColumn = 4
Case "NEW-FIELD"
    Set ErrControl = IncomeGrid
    ErrColumn = 5
Case "BONUS"
    Set ErrControl = IncomeGrid
    ErrColumn = 6
```

- 3 In the `grd_ObjectName_GridName_UpdateObject` sub section (where `ObjectName` and `GridName` are unique variables), increase the case value for every field whose associated column comes after the new column. If the new field is reflected in the database, add a case statement for it.

Sample code before

```
Case 4
  FieldName = "SALARY(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
Case 5
  FieldName = "BONUS(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
```

Sample code after

```
Case 4
  FieldName = "SALARY(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
Case 5
  FieldName = "NEW-FIELD(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "A10"
Case 5
  FieldName = "BONUS(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
```

- 4 In the LoadGridNameGrid sub section, add a call to the ColumnAdd method. If the new column is not to be modified, include the Modifiable:=False parameter.

Sample code before

```
.ColumnAdd "Product/Suffix", BDT_ALPHA, "A2"  
.ColumnAdd "Line/Description", BDT_ALPHA, "A40"
```

Sample code after

```
.ColumnAdd "Product/Suffix", BDT_ALPHA, "A2"  
.ColumnAdd "New/Field", BDT_ALPHA, "A5", Modifiable:=False  
.ColumnAdd "Line/Description", BDT_ALPHA, "A40"
```

- 5 In the CopyGridNameToForm subroutine, for each assignment statement within the for loop(s), increase the second index of the array variable on the left side of the assignment if the column number is higher than the new column. Depending on the cardinality of the grid's data, this will either take the form of GridNameGrid.GridData(i, 1) or GridNameArray(GridNameRow)(i, 1). If the new field is defined on the database, add an assignment statement for the field.

Sample code before

```
IncomeGrid.GridData(i, 4) = _  
    BDT.ConvertToDisplay(.GetField("SALARY", i), _  
        NatFormatLength:="P9.2")  
IncomeGrid.GridData(i, 5) = _  
    BDT.ConvertToDisplay(.GetField("BONUS", i), _  
        NatFormatLength:="P9.2")
```

Sample code after

```
IncomeGrid.GridData(i, 4) = _  
    BDT.ConvertToDisplay(.GetField("SALARY", i), _  
        NatFormatLength:="P9.2")  
IncomeGrid.GridData(i, 5) = _  
    BDT.ConvertToDisplay(.GetField("NEW-FIELD", i), _  
        NatFormatLength:="A10")  
IncomeGrid.GridData(i, 6) = _  
    BDT.ConvertToDisplay(.GetField("BONUS", i), _  
        NatFormatLength:="P9.2")
```

- 6 In the `grd_ObjectName_GridName_KeyDown` section, increase the `ColumnNumber` by one if there is an `If` or `ElseIf` statement such as `If CurrCol = ColumnNumber` where `ColumnNumber` is greater than or equal to the number of the new column.

Sample code before

```
ElseIf CurrCol = 5 Then  
    If KeyCode = ..
```

Sample code after

```
ElseIf CurrCol = 6 Then  
    If KeyCode = ..
```

- To add a ComboBox Grid Column for a field:
- 1 In the Form_Load event, add code to populate the ComboClass object for the selection list associated with the new field.

Sample code

```
NewFieldList.Load NewFieldColumn  
NewFieldList.AddItem "CDN", "Canadian Dollar"  
NewFieldList.AddItem "USA", "American Dollar"  
NewFieldList.AddItem "GER", "German Mark"  
NewFieldList.AddItem "FRA", "French Franc"
```

- 2 In the Global Declarations section, declare a variable as type Column. This variable is used in the Form_Load event and the Load_GridName_Grid sub. Also increase the MAX_GridName_COLS constant by one.

Sample code before

```
Public IncomeGrid As New TrueGridClass  
Const MAX_INCOME_COLS = 8  
Const MAX_INCOME_ROWS = 30
```

Sample code after

```
Public IncomeGrid As New TrueGridClass  
Const MAX_INCOME_COLS = 9  
Const MAX_INCOME_ROWS = 30  
Private NewFieldColumn As Column
```

- 3 In the CheckRemoteError sub section, increase the ErrColumn value in the case statement for every field in the same grid which has a higher column number than the new column.

Sample code before

```

Case "SALARY"
  Set ErrControl = IncomeGrid
  ErrColumn = 4
Case "BONUS"
  Set ErrControl = IncomeGrid
  ErrColumn = 5

```

Sample code after

```

Case "SALARY"
  Set ErrControl = IncomeGrid
  ErrColumn = 4
Case "NEW-FIELD"
  Set ErrControl = IncomeGrid
  ErrColumn = 5
Case "BONUS"
  Set ErrControl = IncomeGrid
  ErrColumn = 6

```

- 4 In the `grd_ObjectName_GridName_UpdateObject` sub section (where `ObjectName` and `GridName` are unique variables), increase the case value for every field whose associated column comes after the new column. If the new field is reflected in the database, add a case statement for it.

Sample code before

```

Case 4
  FieldName = "SALARY(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
Case 5
  FieldName = "BONUS(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"

```

Sample code after

```

Case 4
  FieldName = "SALARY(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
Case 5
  FieldName = "NEW-FIELD(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "A10"
Case 6
  FieldName = "BONUS(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"

```

- 5 In the LoadGridNameGrid sub section, add a call to the ColumnAdd method. If the new column is not to be modified, include the Modifiable:=False parameter. In this example, the Presentation argument is set to dbgSortedComboBox. It is this setting which makes the column behave like a ComboBox.

Sample code before

```

.ColumnAdd "Salary", BDT.GetBDT("P9.2"), "P9.2"
.ColumnAdd "Bonus", BDT.GetBDT("P9.2"), "P9.2"

```

Sample code after

```

.ColumnAdd "Salary", BDT.GetBDT("P9.2"), "P9.2"
.ColumnAdd "New/Field", BDT.ALPHA, "A10", _
    Presentation:=dbgSortedComboBox
.ColumnAdd "Bonus", BDT.GetBDT("P9.2"), "P9.2"

```

- 6 In the CopyGridNameToForm sub section, for each assignment statement within the for loop(s), increase the second index of the array variable on the left side of the assignment if the column number is higher than the new column. Depending on the cardinality of the grid's data, this will either take the form of GridNameGrid.GridData(i, 1) or GridNameArray(GridNameRow)(i, 1). If the new field is defined on the database, add an assignment statement for the field.

Sample code before

```
IncomeGrid.GridData(i, 4) = _
    BDT.ConvertToDisplay(.GetField("SALARY", i), _
    NatFormatLength:="P9.2")
IncomeGrid.GridData(i, 5) = _
    BDT.ConvertToDisplay(.GetField("BONUS", i), _
    NatFormatLength:="P9.2")
```

Sample code after

```
IncomeGrid.GridData(i, 4) = _
    BDT.ConvertToDisplay(.GetField("SALARY", i), _
    NatFormatLength:="P9.2")
IncomeGrid.GridData(i, 5) = _
    BDT.ConvertToDisplay(.GetField("NEW-FIELD", i), _
    NatFormatLength:="A10")
IncomeGrid.GridData(i, 6) = _
    BDT.ConvertToDisplay(.GetField("BONUS", i), _
    NatFormatLength:="P9.2")
```

- 7 In the `grd_ObjectName_GridName_KeyDown` section, increase the `ColumnNumber` by one if there is an `If` or `ElseIf` statement such as `If CurrCol = ColumnNumber` where `ColumnNumber` is greater than or equal to the number of the new column.

Sample code before

```
ElseIf CurrCol = 5 Then
    If KeyCode = ..
```

Sample code after

```
ElseIf CurrCol = 6 Then
    If KeyCode = ..
```

- To add a CheckBox Grid Column for a field:
- 1 In the Global Declarations section, declare constants to represent true and false database values. This variable is used in the `grd_GridName_UpdateObject` and `CopyGridNameToForm` subs. Also increase the `MAX_GridName_COLS` constant by one.

Sample code before

```
Public IncomeGrid As New TrueGridClass
Const MAX_INCOME_COLS = 8
Const MAX_INCOME_ROWS = 30
```

Sample code after

```
Public IncomeGrid As New TrueGridClass
Const MAX_INCOME_COLS = 9
Const MAX_INCOME_ROWS = 30
Const NEWFIELD_FALSE_CONST = "AAA"
Const NEWFIELD_TRUE_CONST = "BBB"
```

- 2 In the `CheckRemoteError` sub section, increase the `ErrColumn` value in the case statement for every field in the same grid which has a higher column number than the new column.

Sample code before

```
Case "SALARY"
    Set ErrControl = IncomeGrid
    ErrColumn = 4
Case "BONUS"
    Set ErrControl = IncomeGrid
    ErrColumn = 5
```

Sample code after

```
Case "SALARY"  
    Set ErrControl = IncomeGrid  
    ErrColumn = 4  
Case "NEW-FIELD"  
    Set ErrControl = IncomeGrid  
    ErrColumn = 5  
Case "BONUS"  
    Set ErrControl = IncomeGrid  
    ErrColumn = 6
```

- 3 In the `grd_ObjectName_GridName_UpdateObject` sub section (where `ObjectName` and `GridName` are unique variables), increase the case value for every field whose associated column comes after the new column. If the new field is reflected in the database, add a case statement for it.

Sample code before

```
Case 4  
    FieldName = "SALARY(" & IncomeRow & ")"  
    Value = grd_EMPL_Income.Columns(ColIndex).Value  
    NatFormatLength = "P9.2"  
Case 5  
    FieldName = "BONUS(" & IncomeRow & ")"  
    Value = grd_EMPL_Income.Columns(ColIndex).Value  
    NatFormatLength = "P9.2"
```

Sample code after

```

Case 4
  FieldName = "SALARY(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"
Case 5
  FieldName = "NEW-FIELD(" & IncomeRow & ")"
  Value = IIf(grd_EMPL_Income.Columns(ColIndex).Value = _
              TRUE_STRING, NEWFIELD_TRUE_CONST, _
              NEWFIELD_FALSE_CONST)
  NatFormatLength = "A10"
Case 6
  FieldName = "BONUS(" & IncomeRow & ")"
  Value = grd_EMPL_Income.Columns(ColIndex).Value
  NatFormatLength = "P9.2"

```

- 4 In the LoadGridNameGrid sub section, add a call to the ColumnAdd method. In this example, the BDT argument is set to BDT_BOOLEAN, regardless of the format of the underlying database field. It is this setting which makes the column behave like a CheckBox.

Sample code before

```

.ColumnAdd "Salary", BDT.GetBDT("P9.2"), "P9.2"
.ColumnAdd "Bonus", BDT.GetBDT("P9.2"), "P9.2"

```

Sample code after

```

.ColumnAdd "Salary", BDT.GetBDT("P9.2"), "P9.2"
.ColumnAdd "New/Field", BDT_BOOLEAN, "A10"
.ColumnAdd "Bonus", BDT.GetBDT("P9.2"), "P9.2"

```

- 5 In the CopyGridNameToForm sub section, for each assignment statement within the for loop(s), increase the second index of the array variable on the left side of the assignment if the column number is higher than the new column. Depending on the cardinality of the grid's data, this will either take the form of GridNameGrid.GridData(i, 1) or GridNameArray(GridNameRow)(i, 1). If the new field is defined on the database, add an assignment statement for the field.

Sample code before

```
IncomeGrid.GridData(i, 4) = _
    BDT.ConvertToDisplay(.GetField("SALARY", i), _
    NatFormatLength:="P9.2")
IncomeGrid.GridData(i, 5) = _
    BDT.ConvertToDisplay(.GetField("BONUS", i), _
    NatFormatLength:="P9.2")
```

Sample code after

```
IncomeGrid.GridData(i, 4) = _
    BDT.ConvertToDisplay(.GetField("SALARY", i), _
    NatFormatLength:="P9.2")
If NEWFIELD_FALSE_CONST = 0 Then
    IncomeGrid.GridData(i, 5) = _
        IIf(.GetField("NEW-FIELD", i) <> "", _
        TRUE_STRING, FALSE_STRING)
Else
    IncomeGrid.GridData(i, 5) = _
        IIf(.GetField("NEW-FIELD", i) = NEWFIELD_TRUE_CONST, _
        TRUE_STRING, FALSE_STRING)
End If
IncomeGrid.GridData(i, 6) = _
    BDT.ConvertToDisplay(.GetField("BONUS", i), _
    NatFormatLength:="P9.2")
```

- 6 In the `grd_ObjectName_GridName_KeyDown` section, increase the `ColumnNumber` by one if there is an `If` or `ElseIf` statement such as `If CurrCol = ColumnNumber` where `ColumnNumber` is greater than or equal to the number of the new column.

Sample code before

```
ElseIf CurrCol = 5 Then
    If KeyCode = ..
```

Sample code after

```
ElseIf CurrCol = 6 Then
    If KeyCode = ..
```

Removing a Field by Hand

The steps required to remove a field are the reverse of those for adding a field. To remove a scalar field by hand, see **Adding a Scalar Field by Hand**, page 185, and reverse the procedure. To remove a grid column field by hand, see **Adding a Regular Grid Column for a Field**, page 189, and reverse the procedure.

Using the Grid

Construct Spectrum supports business object data with up to four dimensions. Business objects with two or more dimensions are referred to as complex business objects. The VB-Maint-Dialog model uses the True DBGrid control to present complex objects on dialogs. To the user, the grid is displayed as a table with each row displaying a unique record, for example, a customer order line. Each column in the grid displays a specific type of information such as a name, a quantity, a price, and so on. The Construct Spectrum client framework comes with the TrueGridClass.cls — a class that encapsulates the True DBGrid control and shields the developer from many of the intricacies of using the grid.

You can write your own code to use Construct Spectrum's TrueGridClass.cls, you can write code to directly manipulate the True DBGrid control, or you can customize the TrueGridClass.cls to meet your specific needs.

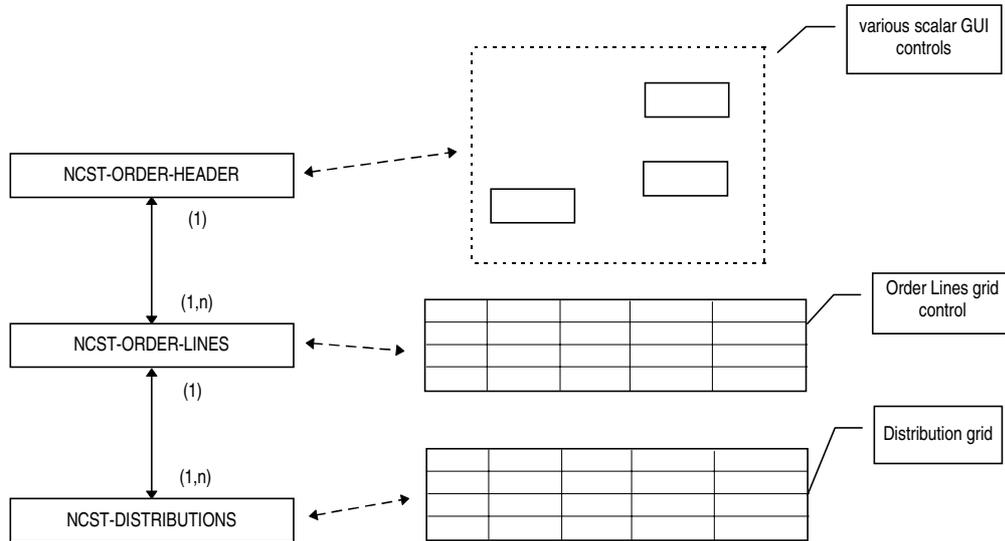
For information about the methods and property interfaces of the TrueGridClass.cls, see **Maintenance Classes**, page 249, in *Construct Spectrum Reference Manual*.

For information about working with the True DBGrid directly, refer to the TrueGrid folder located in Spectrum SDK.

Nested Grids

A single grid can only display data which has the same cardinality — that is, the same number of dimensions. Therefore, if a business object contains both two and three-dimensional information, two grids are required to display all the data.

The demo application (described in Chapter 2) contains an Order Entry example that is a complex business object. The Order Entry example has two grids: one showing order line details and one showing distribution details for each order line. Using the Order Entry as an example, consider the following diagram which shows the relationships between the Order object files and the GUI controls.



Relationship Between a Complex Business Object and GUI Controls In a Grid

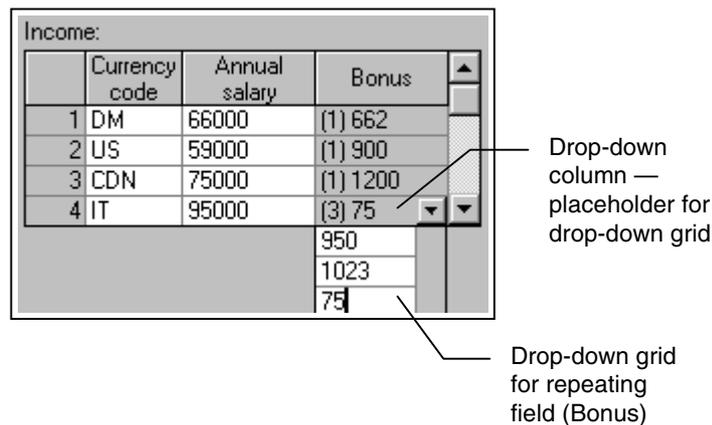
In this example, there is a one-to-many relationship between the Order Lines grid and the Distributions grid. The Distributions grid is said to be nested within the Order Lines grid. Because it is nested, it only displays the rows that are related to the row that is currently selected in the Order Lines grid.

Nested Drop-Down Grids

A drop-down grid is a special type of nested grid that can be used to display nested information. Drop-down grids are used when there is a single repeating field (an MU field) within a block of grid information. In the following data definition example, the Address field maps to a drop-down grid.

```
01 EMPLOYEE-INFO (1:10)
  02 NAME (A10)
  02 ADDRESS (A20/1:3)
  02 SALARY (P10.2)
```

Drop-down grids appear to drop-down out of a parent grid. The parent grid has a placeholder column from which to invoke a drop-down grid. This column is referred to as a drop-down column. Drop-down columns are distinguished from other grid columns because each cell contains a down button from which drop-down data is accessed and because drop-down data is prefixed with an occurrence number:



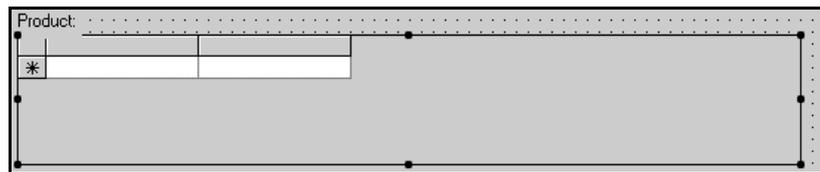
Drop-Down Grid

Nested drop-down grids differ from regular grids in two major ways. First, the GUI control name is prefixed with `ddg` rather than `grd`. Second, the size and position of the nested drop-down grid is controlled by the code at runtime. Therefore, do not tailor the size and position of the drop-down grid.

Note: Nested drop-down grids share the same container as their parent grid, that is, the grid from which the nested drop-down grid is accessed.

Displaying Grids

When the VB-Maint-Dialog model generates a grid control on the dialog, it does not set the grid's properties and, therefore, the grid does not appear properly formatted within the Visual Basic design environment. The following example shows what the grid looks like in the Visual Basic design environment:



Unformatted Grid

Instead, the model generates a subroutine, called LoadGridNameGrid, which is called from the Form_Load event at run time. One load subroutine is generated for each grid on the dialog. Each load subroutine is responsible for formatting a grid before it is displayed to the user. The load subroutine makes a call to the TrueGrid-Class.cls Load method to initialize the grid. It then calls the class ColumnAdd method for each field column to be added to the grid. When the Load subroutine is finished executing, the grid is displayed as follows:

	Product Id	Line Description	Quantity	Unit
1	187361	CAT NUGGETS	10	
2				
3				

Formatted Grid

For more information about the load and add methods, see **Maintenance Classes**, page 249, in *Construct Spectrum Reference Manual*.

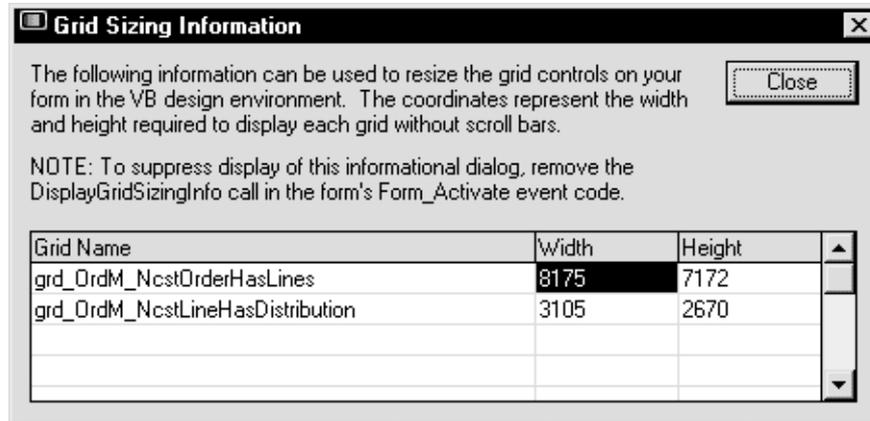
Resizing Grids

The load subroutine described in **Adding a Regular Grid Column for a Field**, page 189, makes one final method call (`SetWidth`) to the `TrueGridClass.cls` to resize the width of the grid based on the length and format of the fields represented in the grid. The `TrueGridClass.cls` makes the grid as wide as required to display all the columns of information, unless it exceeds the right border of the dialog. In this case, a horizontal scroll bar is displayed on the grid, allowing you to scroll the grid to see hidden fields.

Because the `TrueGridClass.cls` automatically resizes the grid, this can cause problems when working on the layout of other GUI controls surrounding the grid. For example, if you want to place GUI controls to the right of the grid, it is difficult to determine at design time whether the grid will overlap the controls at runtime.

You can deal with this situation in two ways: resize the grid using the Grid Sizing dialog or resize the grid manually. The first option involves working with the automatic grid resizing feature. The second option involves disabling this feature and sizing your grid manually. Use the second option when you require more control over the width of your grid and do not require all grid columns to be visible at once.

- To resize your grid using the Grid Sizing Information dialog:
- 1 Run the application.
As the dialog loads, the Grid Sizing Information dialog is displayed:



Grid Sizing Information Dialog

This modal dialog indicates how big to make grids on your form at design time so all grid information is visible and scroll bars are not necessary. Note this information and stop the running application.

- 2 Re-enter the Visual Basic design mode and resize the grid(s) based on the information you gathered from the Grid Sizing Information dialog.
Now you can determine where you can safely place other GUI controls that are in close proximity to the grid.
- 3 Suppress the display of the Grid Sizing Information dialog when you no longer need this information.
To suppress the dialog, comment out the following event code in the Form_Activate event:

```
If Not RepressGridSizingDisplay Then  
    DisplayGridSizingInfo  
    RepressGridSizingDisplay = True  
End If
```

- To resize your grid manually:
- 1 Disable automatic grid resizing by commenting out the `SetWidth` call in the load subroutine.
Commenting out this call will not affect the calculated width of each column but will keep the grid from resizing itself to make all columns visible.
 - 2 Resize the height and width of the grid manually in the Visual Basic design environment.

Tip: At runtime, if there are more columns than can be displayed in the specified width, a horizontal scroll bar is displayed at the bottom of the grid. Users can click the scroll bar to see the remaining columns.

- 3 Comment out the code that displays the Grid Sizing Information dialog (as described in the previous procedure).

Adding Sound to Error Notifications

This section explains how to add sound support to your error notification information. When a field is in error, a Construct Spectrum application can notify the user in several ways. First, the background color of the field can be set to a different color such as red. Second, when the user tabs into the field, the application can display an error tip which looks similar to a Windows tooltip. Construct Spectrum also gives you the option of including sound information with an error.

A Construct Spectrum application can play an error sound file that you provide when the user tabs into a field which is in error or when the user clicks on the sound icon in an error tip. These options can be set by the user.

For more information about setting error notification preferences, see **Using the Demo Application**, page 68.

Construct Spectrum uses the .wav file format for error sound files. You can use Window's Sound Recorder application to record .wav files for your application errors.

Note: If no error sound file exists for a specific GUI control and error, no sound icon is displayed in the error tip — even if the user has selected the sound icon as an error notification preference.

Understanding How a Sound File is Associated With an Error

When an error sound is to be played, a Construct Spectrum application uses a pre-defined convention to associate a .wav file with a specific error. The components required to create this association are outlined in the following table:

Error Component	Source of Error Component	Description
Sound File Path	ERROR_SOUND_PATH constant	Location of the .wav files. (declared in CStObjectConstants.bas). If the constant is empty, the application defaults to the value of App.Path.
Language Indicator	Res.Language	Language indicator. By default, Construct Spectrum applications use the language indicators used by Natural (for example, 1=English, 2=German, 3=French). For a list of language indicators, see System Variables in <i>Natural Reference Manual</i> .
Error Source	ObjectError.MsgType	Error source. Construct Spectrum applications recognize four distinct error sources: <ul style="list-style-type: none"> • Business data type (BDT) errors • Spectrum Dispatch Client (SDC) errors • Local business validation errors (originating in a Visual Basic maintenance object) • Server errors (originating in an object subprogram) Valid error source values are represented by constants stored in CStObjectConstants.bas. These constants are: <ul style="list-style-type: none"> • ERROR_SOURCE_SDC • ERROR_SOURCE_BDT • ERROR_SOURCE_VALIDATE • ERROR_SOURCE_SERVER

Error Component	Source of Error Component	Description (continued)
Error Number	ObjectError. ErrorNr	Error within the specified error source.
Sound File Delimiter	SOUND_FILE_ DELIMITER constant	Character used to delimit the components of an error sound file.

These components are assembled as follows:

Sound File Path + \ + Language Indicator + Sound File Delimiter + Error Source
+ Sound File Delimiter + Error Number + .wav

The following example shows how the application attempts to associate a .wav file
with an error:

Example input

```
ERROR_SOUND_PATH = blank
Res.Language = 1
ObjectError.MsgType = ERROR_SOURCE_SDC (1)
ObjectError.ErrorNr = 522
SOUND_FILE_DELIMITER = "-"
```

Example output

```
C:\Program Files\Construct Spectrum\MyApp\1-1-522.wav
```

Tip: Errors that originate in the SDC, BDT, or local validation layers are raised using Visual Basic's Err object. The error number used when raising the error is derived by adding a Visual Basic constant (vbObjectError) to a unique application-specific number. Look at the constants defined in CSTConst.bas for examples. These errors are all handled in the ValidAssignment subroutine in the BDTSupport.bas module. To make the error number more readable (adding vbObjectError produces a large, negative number), the ValidAssignment subroutine subtracts vbObjectError from the error number. Therefore, the original, unique, application-specific number is used to associate a .wav file with an error.

Multilingual Support for Maintenance Dialogs

Construct Spectrum provides support for multilingual applications. To set up a multilingual application, create language specific resource files for the application.

The generated maintenance dialog and Visual Basic maintenance object have code that looks for resources in the application directory in a resource file called App. For each supported language, create App.* resource files (where * is the language code). The generated dialogs will then use the resource files.

For more information about setting up multilingual applications, see **Internationalizing Your Application**, page 367.

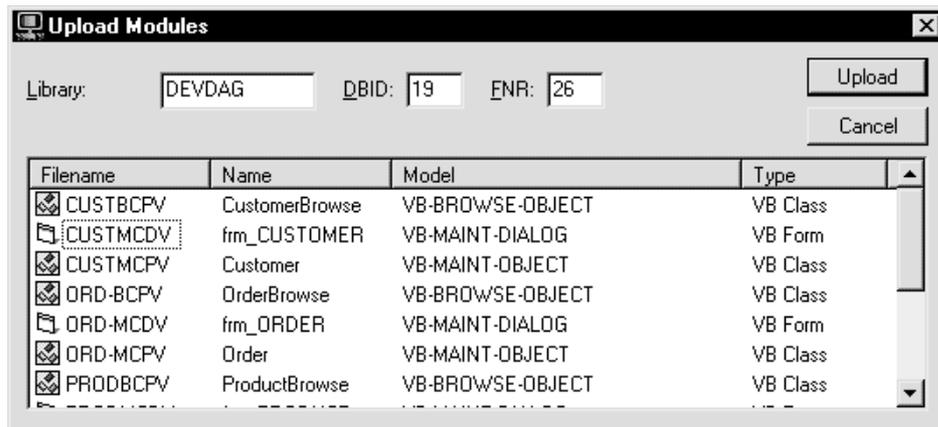
Uploading Changes to the Server

Sometimes changes occur on the server, such as changes to the Predict file and field definitions used by your maintenance dialog. It is often easier to regenerate the affected modules than to implement the changes by hand. This includes modules that were generated for the client — specifically, Visual Basic maintenance objects.

If you have tailored a Visual Basic maintenance object or a maintenance dialog on the client (for example, by adding user exit code), upload the client version of the Visual Basic maintenance object or maintenance dialog to the server to preserve the user exit code during regeneration. Once regeneration is complete, you can download the regenerated module(s).

Tip: Before regenerating a maintenance dialog, see **Strategies for Customizing a Maintenance Dialog**, page 155, for information about saving customizations in your maintenance dialog.

- To upload changes to the server:
- 1 Open the Construct Spectrum project that contains the changes you are uploading.
 - 2 On the Construct Spectrum Add-In, select **Upload Generated Modules**. The Upload Modules dialog is displayed:



Upload Modules

- 3 The library name, DBID (database ID), and FNR (file number) default to the values entered for the last open project. If necessary, type the library name, DBID, and FNR for the server library to which you are uploading.
- 4 Click **Upload**.
The selected modules are uploaded to the server.

CREATING AND CUSTOMIZING BROWSE DIALOGS

This chapter provides step-by-step instructions for generating the modules required to provide browse services from the client. It describes how to generate the necessary modules, download the client modules to your PC, integrate the new browse modules into an existing Construct Spectrum project, and display server database information from a browse dialog. Also included is information about modifying the components so that you can customize the features and functions of the resulting browse dialog.

The following topics are covered:

- **Overview of the Browse Dialog**, page 216
- **Creating a Browse Dialog**, page 220
- **Customizing On the Client**, page 232
- **Understanding Browse Command Handlers**, page 238

Overview of the Browse Dialog

A browse dialog provides users with lists of data. Typically, this data is shown within a browse dialog and represents rows of information from a remote database table. Browse dialogs can also be set up to display data that is obtained locally — from a PC server connected to your network, for example.

About Browse Dialogs

The underlying structure of a browse dialog is different from that of a maintenance dialog. Unlike maintenance dialogs, which use a unique Visual Basic form for each maintenance object in your application, all generated browse dialogs use the same underlying browse form that is supplied with the Construct Spectrum client framework. This generic form communicates with other client framework components and with the browse modules you generate to configure itself at runtime for a particular object browse subprogram and to retrieve data. The browse dialog that is displayed to the user is the result of this process.

Although you cannot modify a browse dialog directly, you can influence its behavior based on:

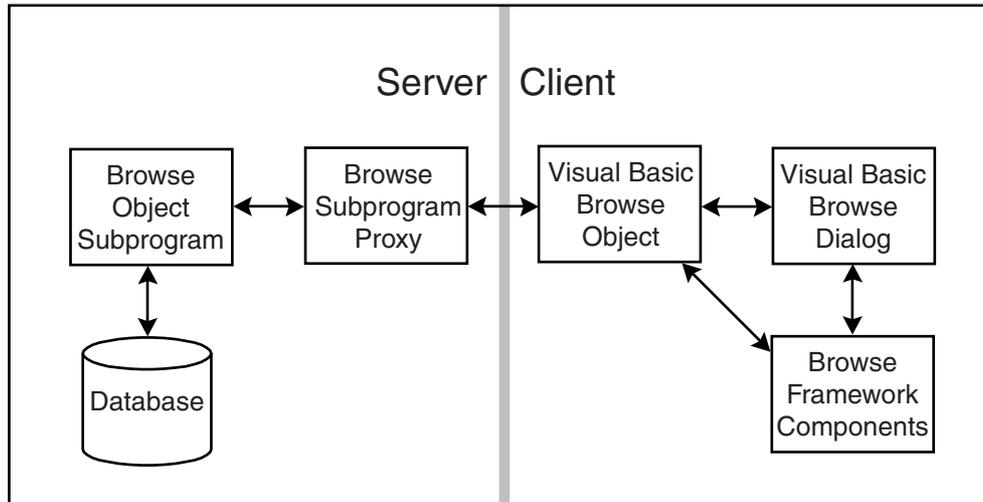
- How the data file(s) used in the browse are set up in Predict
- Options you choose when you generate browse modules
- Customized code you write to work with your generated browse modules and their related client framework components

The Browse Process

The browse dialog that a user works with is configured dynamically at runtime. Unlike maintenance dialogs, which have a unique form that corresponds to each dialog, there is no unique form that corresponds to each browse dialog. Rather, a browse dialog is configured at runtime based on the interaction of the following:

- Object browse subprogram
- Object browse subprogram proxy
- Visual Basic browse object
- Client framework components

The following diagram illustrates these components:



Components Included in the Browse Process

The features and functions of a particular browse dialog depend on how these components are configured. You can modify these components to influence the features and functions of a browse dialog.

Browse Object Subprogram

The browse object subprogram reads database records on the server and returns them to the client. Each browse subprogram can support multiple keys to allow the user to select the most appropriate access path to retrieve the desired records.

Generate a browse object subprogram using the Object-Browse-Subp model. You can specify overrides to many of the default values selected by the model before generating or regenerating. For example, you can specify the keys available for accessing records displayed in your browse dialog.

The characteristics of your browse object subprogram depend on the relationships between the related database files and fields. You can perform a number of modifications to the metadata that describes these relationships using Predict. For more information, see **Understanding Browse Command Handlers**, page 238.

Browse Object Subprogram Proxy

A subprogram proxy is required to access the browse object subprogram from the client application. The subprogram proxy calls an object subprogram that fulfills a data request on behalf of a browse request. It is also responsible for converting data between the network transfer format and the Natural variable used by the parameters of the browse object subprogram.

You can make a number of changes to the subprogram proxy that affect the functioning of your browse dialog. Most of these changes are related to how browse information is transmitted between the client and server. For information about customizing the subprogram proxy, see **Using the Subprogram Proxy Model**, page 129, in *Construct Spectrum Programmer's Guide*.

Visual Basic Browse Object

The Visual Basic browse object delivers information about the columns and keys supported by the browse subprogram to the client framework components.

The Visual Basic browse object is generated by the VB-Browse-Object model for a specific database file. It uses the BrowseBase class to interface with other parts of the client framework and with the application. The Visual Basic browse object instantiates and initializes a BrowseBase object. The initialization performed by the Visual Basic browse object sets up definitions for:

- logical search keys
- formatting information for data columns
- optionally, inserts data into the data cache for static lists

It also sets up a data cache area on the client to save the results of multiple requests to minimize network congestion and speed up the re-display of previously fetched data. The data cache is an object in its own right.

Data Cache

The data cache is populated by the BrowseBase object Fetch method when a user specifies a starting value and presses the Get button. This triggers a remote CallNat that reads records from a database and returns them to the client. As records are received, they are added to the data cache. From the data cache, they are transferred to a ListView control on the browse dialog where the user sees the data. If the user requests the next (contiguous) set of records, they are retrieved from the server and appended to the data cache and ListView. This process continues until the user repositions the view to a new location in the file by selecting a new starting value or changing the key value. Whenever the user repositions the view, the data cache and ListView are cleared and a new list of rows is presented.

The data cache mechanism is significant for the following reasons:

- It enables the user to scroll backward through previously viewed data without having to reread this data from the server.
- Because the data cache represents a copy of the data, it may not always reflect the current state of data on the server. For example, if cached records are updated or deleted, the user must issue a Refresh command to obtain the new values.
- It is possible to read server data into the data cache and retrieve it programmatically, without having to invoke a browse dialog. For more information, see **Browse Classes**, page 57, in *Construct Spectrum Reference Manual*.
- The data cache can be saved in memory when a browse dialog is closed and restored when the browse dialog is requested again. This alleviates the need to continually retrieve the same browse data from the server.

Framework Components

Several client framework components work together to provide browsing services at runtime. These components are encapsulated in a single class, the BrowseManager class. This class provides an interface to perform common browsing activities, for example, to get a specific row of information, get all rows of information, or display a modal or MDI browse dialog.

Internally, the BrowseManager uses several framework components, the most important of which is the browse dialog. There are two versions of the dialog: a modal (GenericBrowse.frm) and an MDI (GenericMDIBrowse.frm) dialog. Each dialog is dynamically configured at runtime to display specific browse data. This process is described in **Understanding Browse Command Handlers**, page 238.

Creating a Browse Dialog

The following tasks are required to create a browse dialog. Once you have completed these steps, you are ready to compile the application in Visual Basic and test the new browse dialog.

- Review and optionally modify Predict set up
- Use the Construct models to generate modules
- Download the modules to the client using the Construct Spectrum Add-In
- Update the Construct Spectrum project

These tasks are described in detail in the following sections.

Setting up Predict for the Browse Dialog

Prior to generating the modules of your browse dialog, certain attributes can be defined within Predict to extend the functionality of what is generated. You can modify any of these attributes in Predict and regenerate your browse modules to implement your changes. For information about regenerating browse modules, refer to *Construct Spectrum Programmer's Guide*.

Business Data Types

Browses make use of business data types (BDTs) to format the data that is shown within the ListView control of the browse dialog. If you want special formatting of the browse data, add business data types to the fields within Predict prior to generating the browse components. For information about working with business data types, see **Using Business Data Types**, page 153, in *Construct Spectrum Programmer's Guide*.

Descriptive Fields

When a browse is initiated from a field on a maintenance dialog, it is referred to as a foreign key browse. For example, the Construct Spectrum demo application has a foreign key browse set up for the Warehouse field located on the Order maintenance dialog. When a foreign key browse is initiated, only the foreign key values (warehouse numbers in this case) are displayed unless you designate other fields in the foreign file as descriptive in Predict.

In the demo application, the WAREHOUSE-NAME field is designated as descriptive. When you browse on the Warehouse field from the Order maintenance dialog, warehouse numbers and their corresponding names are displayed so that users can easily select the appropriate warehouse. For more information about linking browse and maintenance functions, see **Understanding the Browse and Maintenance Integration**, page 343.

Using the Construct Models to Generate Browse Modules

Each module that a browse dialog requires can be generated with the VB-Client-Server-Super-Model, or you can generate them one at a time using their models. Use the following guidelines to determine which generation approach is appropriate for you.

- If you are creating a new application or a new object, use the super model.
- If you changed the file structure of a previously generated application, use the super model.
- If you want finer control over the generation results, such as hand-coding user exits, use the individual models.

This section explains how to generate a browse module from the individual models. For information about using the super model, see **Using the Super Model to Generate Applications**, page 93.

Generating browse modules involves the following steps, which must be performed in this order:

- 1 Use the Object-Browse-Subp model to generate the object browse subprogram and supporting parameter data areas (PDAs) on the server.
- 2 Use the Subprogram Proxy model to generate a proxy that enables the client to access the browse subprogram.
- 3 Use the VB-Browse-Object model to generate a Visual Basic browse class that supports the generated browse subprogram.
- 4 Extend the object factory for the application to include references to the browse business object.
- 5 If the browse dialog is to support record selection and action buttons, you will also need to create a command handler and link this to the object factory.

These steps are described in more detail in the following sections and must be performed in the order specified.

Tip: Use the same four-character prefix to name all generated modules belonging to a single object. This convention makes it easier to select modules for downloading. For example, to download all client modules related to a Customer object, type “CUST*” (where “*” is the wildcard character) to narrow the list of available items to those starting with CUST.

Generating the Browse Subprogram and PDAs

A browse subprogram reads database records on the server and returns them to the client. Each browse subprogram requires three application-specific parameter data areas that contain information that is passed to, or received from, the subprogram. Each browse subprogram can support multiple keys to allow the user to select the most appropriate access path to retrieve the desired records.

The Object-Browse-Subp model is used to generate the object browse subprogram and its three supporting parameter data areas: *BPRI, *BROW, and *BKEY, where * represents a prefix that you specify.

For a detailed description of this model, see **Object-Browse Models**, page 417, in *Natural Construct Generation User's Manual*.

Generating the Subprogram Proxy

A subprogram proxy is required to access the generated browse subprogram (or any other subprogram) on the server from the client application. The subprogram proxy is responsible for converting data between the network transfer format and the Natural parameter data format used by the browse subprogram.

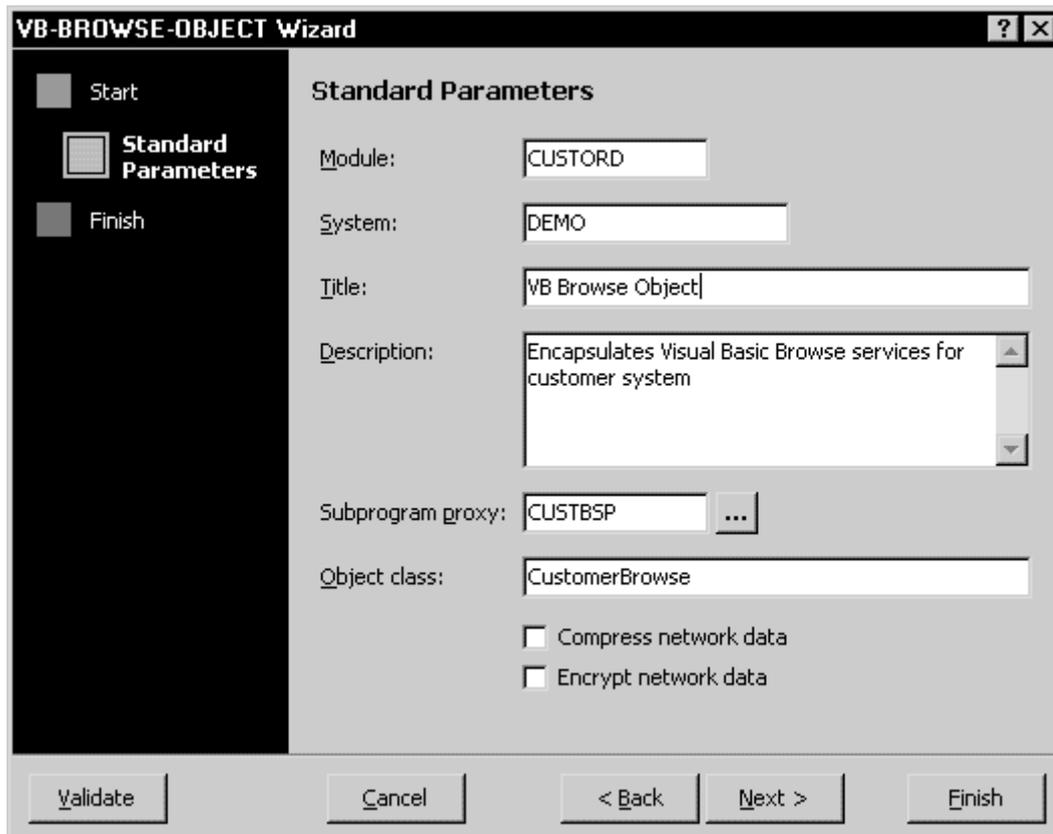
For information about generating a subprogram proxy, see **Using the Subprogram Proxy Model**, page 129, in *Construct Spectrum Programmer's Guide*.

Generating the Visual Basic Browse Object

Each object browse subprogram that will be accessed by users needs a supporting class generated using the VB-Object-Browse model. This class delivers information about the columns and keys supported by the browse subprogram to the client framework components, which then populates a browse dialog with the requested browse information.

You can access the VB-Object-Browse model in the Generation subsystem on the server or use the model wizard in the Construct Windows interface.

The following example shows the Standard Parameters step in the VB-OBJECT-BROWSE wizard:



VB-OBJECT-BROWSE Wizard — Standard Parameters

The Standard Parameters step is similar for all model wizards. The common parameters (Module, System, Title, and Description) are described in **Standard Parameters Wizard Step**, page 269, in *Natural Construct Generation User's Manual*.

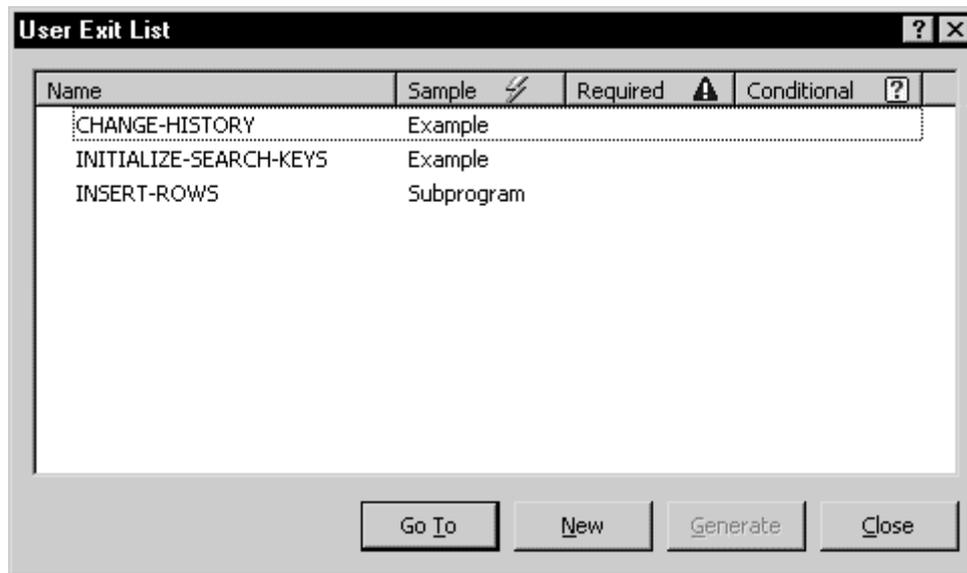
For general information about using the Construct Windows interface, see **Using the Construct Windows Interface**, page 77, in *Natural Construct Generation User's Manual*.

The parameters on this dialog are:

Parameter	Description
Subprogram proxy	Name of the subprogram proxy that communicates with the object browse subprogram for this Visual Basic browse object.
Object class	Name for the generated browse class to be used within Visual Basic.
Compress network data	Indicates whether the parameters sent to the server are compressed to reduce transmission time. Compression is typically not required for a Visual Basic browse object because parameters sent to the server tend to be small. Enabling compression in this situation may actually increase demands on system resources because the overhead associated with invoking compression routines is not offset by the reduced volume of data being transferred.
Encrypt network data	Indicates whether the parameters sent to the server are encrypted. Encryption is used to secure sensitive data. Typically, this check box is not selected because browse data requests sent to the server usually do not contain sensitive information.

Note: The *Compression* and *Encryption* options apply only to data sent from the client to the server. To enable compression and encryption for data sent from the server to the client, select the *Compression* and *Encryption* options for the *Subprogram-Proxy* model, which is described in **Using the Subprogram Proxy Model**, page 129, in *Construct Spectrum Programmer's Guide*.

After supplying model parameters, you can customize the generation results by creating user exit code for the module. The following example shows the User Exit List for the VB-Browse-Object model:



User Exit List

For more information about using the Code window, see **Using the Construct Windows Interface**, page 77, in *Natural Construct Generation User's Manual*.

For more information about the user exists for this model, see **User Exits for the Natural Construct Models**, page 575, in *Natural Construct Generation User's Manual*.

Defining Alternate Browse Data Sources

The VB-Browse-Object model is used to retrieve server database records by making requests to a generated object browse subprogram. There may be times when you want to allow browsing of data that is not defined in a server database file. Instead, you may have data that is defined within files or hard-coded on a client. In such cases, you can present this data to the user with an interface that is similar to the browse interface they are familiar with.

To generate this type of browse dialog you would not use the Object-Browse-Subp, Subprogram-Proxy, and VB-Browse-Object models as described above. Instead, you would use the VB-Browse-Local-Data-Object model.

You can access this model in the Generation subsystem on the server or use the VB-Browse-Local-Data-Object wizard in the Construct Windows interface. The following example shows the Standard Parameters step for the model wizard:

The screenshot shows a wizard window titled "VB-BROWSE-LOCAL-DATA-OBJECT Wizard". On the left is a navigation pane with three steps: "Start", "Standard Parameters" (which is selected and highlighted), and "Finish". The main area is titled "Standard Parameters" and contains the following fields:

- Module:** CUSTORD
- System:** DEMO
- Title:** Browse Province Data
- Description:** This Browse object supports browsing of hard-coded province data
- PREDICT view:** PROVINCE-TABLE (with a browse button "...")
- Object class:** ProvinceBrowse

At the bottom of the wizard are five buttons: "Validate", "Cancel", "< Back", "Next >", and "Finish".

VB-Browse-Local-Data-Object Wizard — Standard Parameters

The Standard Parameters step is similar for all model wizards. The common parameters (Module, System, Title, and Description) are described in **Standard Parameters Wizard Step**, page 269, in *Natural Construct Generation User's Manual*.

For general information about using the Construct Windows interface, see **Using the Construct Windows Interface**, page 77, in *Natural Construct Generation User's Manual*.

The parameters on this dialog are:

Parameter	Description
Predict view	Name of a Predict view (optional). The VB-Browse-Local-Data-Object model allows you to define your file within Predict as a means to document the required field names, field lengths, and column headings. Be aware, however, that no physical file is required to support this model. If you do not want to create a definition of your browse fields within a Predict file, you must define your browse fields in the ADD-COLUMNS user exit as in the following example.
Object class	Name for the generated browse class to be used within Visual Basic.

Example of adding browse field definitions in the ADD-COLUMNS user exit

```
DEFINE EXIT ADD-COLUMNS
'
' AddColumn Name, Heading, Business Data Type, Format, Show by Default
AddColumn "STATE-CODE", "State Code", "", "A2", True
AddColumn "STATE", "State Name", "", "A40", True
AddColumn "TAX", "Sales Tax", BDT_PERCENT, "N2.2", False
END-EXIT
```

In the previous example, the browse dialog shows the State Code and State Name by default; however, the user could modify the options to also display the Sales Tax column. A BDT has been associated with the Sales Tax column to provide special formatting.

Additionally, you need to add code to the INSERT-ROWS user exit. This user exit defines data that is to be shown in the browse by calling the AddData method as in the following example.

Example of defining browse data from the INSERT-ROWS user exit

```
DEFINE EXIT ADD-COLUMNS
'
' AddData Unique ID, State code, State, Sales tax
AddData "1", "ALBA", "Alabama", 8.0
AddData "2", "AK", "Alaska", 5.5
etc.
END-EXIT
```

In addition to the values to be displayed on the browse screen, the first parameter of the AddData method must contain a unique value that is used as an internal record identifier.

Downloading the Client Modules

After generating all required browse modules on the server, you must download the modules required on the client. The following table outlines which modules are required on the client and includes brief descriptions of their roles:

Model	Module Suffix	Visual Basic extension	Description
Object-Browse-Subp	BKEY BPRI BROW	n/a	Updates the library image file with parameter definitions.

Model	Module Suffix	Visual Basic extension	Description (continued)
Subprogram-Proxy	BSP	n/a	Updates the library image file with application service definitions that describe the object subprogram browse method and the data it requires.
VB-Browse-Object	BCPV	.cls	Delivers information about the columns and key fields supported by the browse subprogram to the client framework components.

Note: The module suffix names listed in the previous table are suggested names only. However, when you generate with the super model, modules are given these suffix names automatically.

- To download modules to the client:
- 1 Open the Construct Spectrum project that you are updating.
For information, see **Creating a Construct Spectrum Project**, page 123.
 - 2 On the Construct Spectrum submenu, click **Download Generated Modules**.
 - 3 Ensure you are pointing to the correct library and FUSER on the server.
 - 4 List the modules from the library you want to download by using wildcard notation (*) in the **File Download** text box and then click **List**.
A list of modules on the server is displayed. The browse modules you generated should be among them.
 - 5 Select the browse modules you generated and click **Download**.
You can identify browse modules based on their module suffixes, which are shown in the table at the beginning of this section. The Visual Basic browse object is automatically added to your Construct Spectrum project.

For more information on downloading modules to the client and setting up a Construct Spectrum project, refer to **Creating a Construct Spectrum Project**, page 123.

Updating the Project

There may be times where you want to update the project using the extend object factory. The following discusses when you would need to hand-code the object factory, and how to determine if you need to.

Extend Object Factory

You must hand-code the object factory only if you are adding a new browse dialog to your application or you have changed the actions available for an existing business object. An example of changing the available actions for a business object is when you add a browse action to a business object that had been available to the user only through a maintenance action.

Tip: To determine whether you need to hand-code the object factory, access the Open dialog and select each object and its associated action. If the selected object action does not display, do some hand-coding to add the required object actions.

For information about hand-coding the object factory, see **Customizing the Object Factory**, page 307.

Customizing On the Client

Although you cannot modify browse dialogs directly, there are customizations you can make on the client to modify or enhance the behavior of a browse dialog.

Adding Command Handlers

If the browse dialog is to support action buttons that perform specialized processing on the selected records, define and create command handlers for these buttons.

For more information about adding command handlers for your browse dialog, see **Understanding Browse Command Handlers**, page 238.

Customizing the Generic Browse Dialog

The generic browse dialog is the dialog from which all browse dialogs are configured at runtime. This dialog can be customized through the Browse Dialog API. For more information, see **Browse Classes**, page 57, in *Construct Spectrum Reference Manual*.

Understanding the BrowseManager Class

Every Construct Spectrum application contains a Visual Basic class called the BrowseManager. This class encapsulates the handling of browse services in a single class. Application components use instances of this class as described in the following sections.

Display the Browse Dialog

The BrowseManager creates a browse dialog, links it to a specific browse data source, and formats the dialog to display the data. The dialog can be a modal or an MDI child dialog. Additionally, the dialog can be formatted to begin browsing with a specific key field and key field value.

Support a Browse Command Handler

The `BrowseManager` can link a custom browse command handler to a browse dialog. Browse command handlers add features to your browse form such as:

- Command buttons
- Toolbar buttons enabled on the MDI frame
- Actions for double-click or the Enter key
- Menus that are activated by the right mouse button

Return a Specific Row of Data

The `BrowseManager` returns a specific browse row of data from a data source, based on a key name and key value. An example of a data source is a Natural database residing on your server.

Return All Rows of Data

The `BrowseManager` returns all data rows in a specified table from a data source.

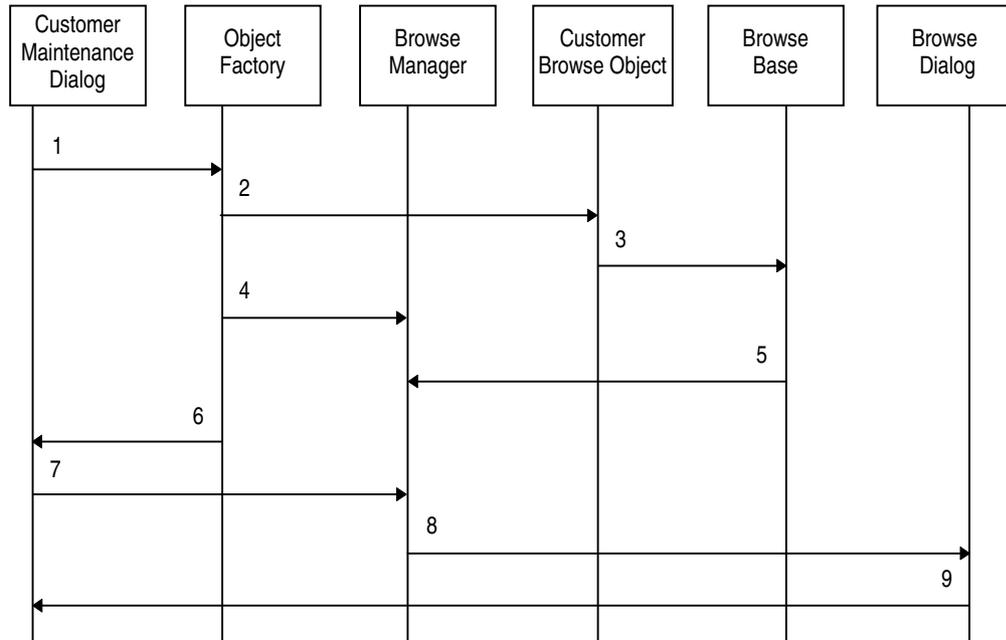
Using the BrowseManager

Applications use the global function, `GetBrowser(tablename)`, to create instances of the `BrowseManager` class for a specific database file. `GetBrowser()`, which is located in the object factory, creates, initializes, and returns a reference to a `BrowseManager` object. The `tablename` parameter is a logical name that identifies which Visual Basic browse object to use when it initializes the `BrowseManager`. For more information, see **Using the Object Factory**, page 305.

Some of the application components that use the `BrowseManager` class are:

- Object factory
- Visual Basic browse object
- Maintenance dialogs
- Custom browse command handlers

The following diagram shows how a Customer maintenance dialog can use the `BrowseManager` class:



Interaction Required to Display a Browse Dialog

Each numbered step in the diagram is explained below:

- 1 The user requests a browse from the Customer maintenance dialog. In this example, the user requests to browse a list of customers on the CUSTOMER file. The maintenance dialog calls the GetBrowser function in the object factory with the parameter "CUSTOMER".
- 2 The object factory creates a CustomerBrowse Visual Basic browse object. This object contains information unique to the Customer browse such as:
 - Column names and captions
 - Column formats and business data types (BDTs) used to format data for display
 - Key names and captions
- 3 Settings from the CustomerBrowse Visual Basic browse object are used to configure a BrowseBase object.
- 4 The object factory instantiates a BrowseManager.

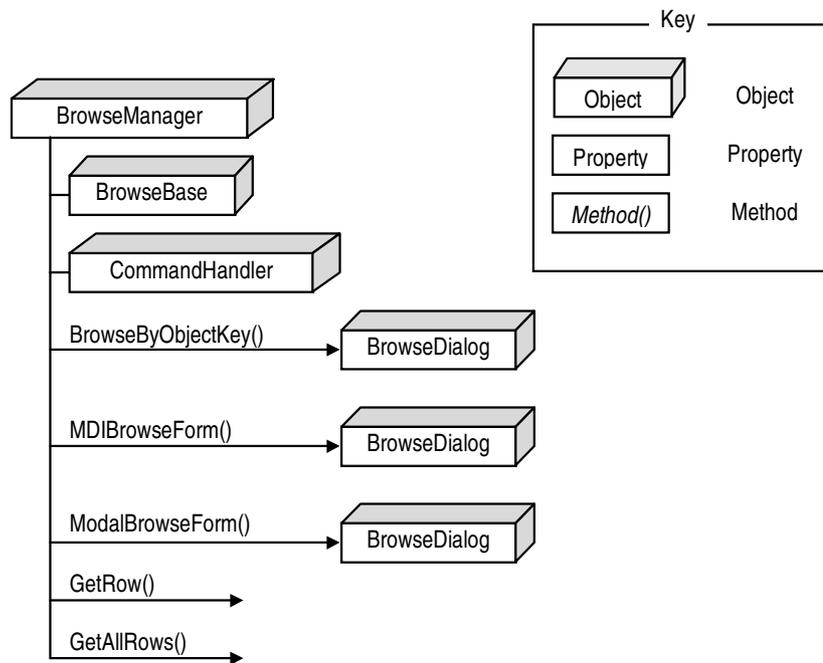
- 5 The BrowseManager object is initialized by setting its BrowseBase property to point to the BrowseBase object created in step 3.
- 6 A reference to the initialized BrowseManager is returned to the Customer maintenance dialog. At this point, the BrowseManager is configured to support the services of the Customer browse.
- 7 The user's initial request to browse a list of customers can be fulfilled. The list of customers is displayed in a modal dialog. To do this, the following command is sent from the Customer maintenance dialog:

```
BrMgr.ModalBrowseForm("CUSTOMER")
```

- 8 The BrowseManager configures and displays a modal browse dialog listing the customers from the CUSTOMER file.
- 9 Any actions requested from the browse dialog are handled by the BrowseManager. For example, if the user selects a customer record and then selects the OK button, the browse dialog is closed and the selected record is returned to the Customer maintenance dialog.

Tip: You can customize the BrowseManager class to support new properties and methods. However, do not modify the interfaces of the current methods supported by the BrowseManager.

The following diagram depicts the structure of the BrowseManager:



Internal Structure of the BrowseManager Class

The BrowseManager class bundles browsing functionality into several methods. These methods are only enabled when the BrowseBase property has been set to an initialized BrowseBase object. A command handler object is an optional property that can be used to enhance the functionality of browse forms created by the BrowseManager class.

BrowseManager Methods

This table lists the methods or services offered by the BrowseManager:

Service	Description
BrowseByObjectKey	Creates a modal browse dialog. The dialog's search key value(s) are set to the values in a parameter reference to a NaturalDataArea object, where the NaturalDataArea is the key structure used by maintenance dialogs. If a row is selected, maps the key values in the row to the NaturalDataArea parameter and returns True.
MDIBrowseForm	Creates a child MDI (multiple-document interface) browse dialog based on the GenericMDIBrowse.frm client framework component. Optionally, links a command handler to the dialog. Returns a reference to the dialog.
ModalBrowseForm	Creates a modal browse dialog based on the GenericBrowse.frm client framework component. Optionally, sets the form's search key to a key specified in a parameter. If a row is selected, returns a reference to the BrowseDataCache object.
GetRow	Clears the data cache in the BrowseBase object unless it is a static browse (fixed number of rows). Sets the BrowseBase object search key to the key specified in a parameter. If a row is successfully retrieved and stored, returns a reference to the BrowseDataCache object.
GetAllRows	Clears the data cache in the BrowseBase object. If all rows are successfully retrieved from the data source, returns a reference to the BrowseDataCache object.

For more information about BrowseManager methods, see **Creating and Customizing Browse Dialogs**, page 215, in *Construct Spectrum Reference Manual*.

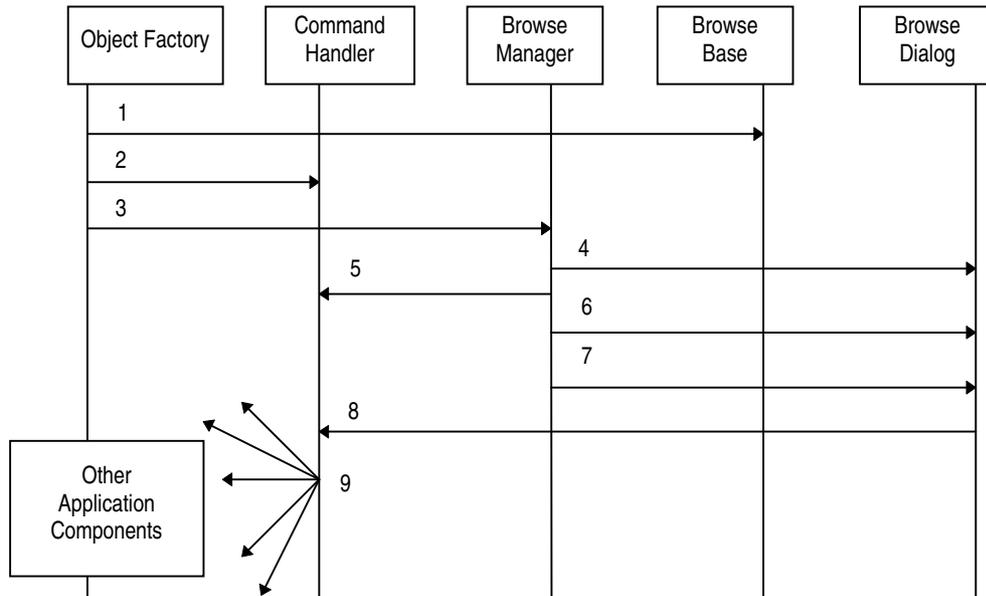
Understanding Browse Command Handlers

Browse Command handlers are custom objects you create to handle commands originating from browse dialogs. They can be used to add command buttons to a browse dialog, enable toolbar buttons on the MDI frame, set default actions for double-click and the Enter key, and to display menus activated by the right mouse button.

All browse command handlers must implement certain public methods and properties. These are supplied in a sample browse command handler class template that you can copy and use as a starting point to create your own browse command handlers.

Tip: Use the browse command handler class template, `BrowseCmdHandler.cls` located in the Construct Spectrum client Framework directory as the starting point for creating your own browse command handler.

The following diagram illustrates how a browse command handler object interacts with other objects in your application:



Browse Command Handler Overview

Each numbered step in the diagram is explained below:

- 1 The object factory creates a BrowseBase object which is initialized with a specific Visual Basic browse object. Interaction between the BrowseBase and browse objects is described in **Using the BrowseManager**, page 233.
- 2 The object factory creates the browse command handler.
- 3 The object factory creates a BrowseManager object and links it to the command handler and the BrowseBase object.
- 4 BrowseManager creates the browse dialog.
- 5 BrowseManager initializes the command handler with a reference to the browse dialog and the BrowseBase object.
- 6 BrowseManager adds a command button, menu item or both for each supported command handler command.
- 7 BrowseManager sets the default command if the command handler supports one. This command is invoked by double-clicking or pressing Enter on a selected row.

- 8 When a user initiates a command on the browse dialog that is handled by a command handler, the command handler is notified.
- 9 The command is executed.

Other features you can implement with a command handler include:

- A browse dialog.
Users can drill down into more detailed information using a browse dialog.
- A link to a maintenance dialog.
Users can invoke a maintenance dialog that is populated with a row selected from a browse dialog. To view an example of this, see the Order browse dialog set up in the demo application. From the Order browse dialog, users can select a row and then select the Update button to open the Order maintenance dialog.
- A delete function.
Users can delete a database record from a browse dialog. The Order browse dialog in the demo application also includes an example of this function. To delete a record in the demo application, the user selects a row and then the Delete button. The record corresponding to that row is deleted. To accomplish this, the Order maintenance dialog object is invoked behind the scenes and used to delete the record.

Creating Browse Command Handlers

The steps to create a browse command handler and link it to your application are described below. Once you create the command handler, you must supply the code to customize the command handler. This is described in **Coding the Custom Browse Command Handler**, page 241.

- To create a browse command handler and link it to your application:
- 1 Create a Visual Basic class that implements the browse command handler. Copy the sample BrowseCmdHandler.cls template in the client Framework directory to use as a starting point.
 - 2 Make the application aware of the browse command handler by copying and modifying the following code in the GetBrowser() function. The GetBrowser() function creates the BrowseManager object for the particular browse dialog created at run time and is part of the object factory.

```

Public Function GetBrowser(TableName As String) As BrowseManager

    Dim BrMgr As New BrowseManager

    ' Return a browser object for the requested table.
    Select Case TableName
    Case "NCST-ORDER-HEADER"
        Set BrMgr.BrowseObject = New OrderBrowse
        BrMgr.Caption = "Query Orders"
    ' Copy and Modify this block of code to hook in a browse command handler
    -- >>
    ' Setting this property will attach the OrderAsBrowseTarget object
    ' to the BrowseManager to handle any commands originating from
    ' the browse.
        Set BrMgr.CommandHandler = New OrderAsBrowseTarget
    ' -----
    <<

```

Now that you have created a custom command handler and linked it to your application, see the next section.

Coding the Custom Browse Command Handler

A command handler is an object that implements two special public methods: `UICommandState()` and `UICommandTarget()`. These two methods are the hooks into the client framework components that allow commands to be triggered, intercepted, and handled throughout your application.

These methods are described in more detail in **Defining, Sending, and Handling Commands**, page 273.

When a command handler is linked to a browse dialog, the dialog notifies the application framework that it needs to handle commands linked to the Command IDs in the command handler. For example, the framework would be notified whenever the Print toolbar button or menu command is clicked on the Browse dialog.

If the command IDs of the browse command handler match any of those on the MDI toolbar or menu, those commands are hooked by the browse dialog. When a user clicks on the hooked toolbar button or selects the hooked menu item, the command in the browse command handler is triggered.

Note: Commands that are to be hooked into the MDI toolbar or menu must already exist on or be added to the MDI frame.

Enabling Commands on the Browse Toolbar and Menu

An important decision to make when coding the `UICommandState()` and `UICommandTarget()` methods is whether or not you want the handled commands to be enabled by the toolbar buttons and menu on the MDI frame. To enable these commands on the MDI frame, assign the proper command IDs to each command in your command handler. The correct command ID is determined by matching it with the corresponding command ID assigned to the command you want to hook in the MDI frame.

The following code sample shows how you would enable commands on the toolbar and menu by assigning command IDs.

Sample code from the `CommandHandler` template that assigns command IDs:

```
Private Sub Class_Initialize()  
  
    ' Initialize The commands supported by this command handler.  
    CommandHandlers(1).ID = CMD_ACTIONS_UPDATE  
    CommandHandlers(1).Caption = "Update..."  
    CommandHandlers(2).ID = CMD_ACTIONS_DELETE  
    CommandHandlers(2).Caption = "Delete"  
  
End Sub
```

Tip: If you want to internationalize your application, avoid hard-coding text strings like `Caption = "Update"`. For more information, see **Internationalizing Your Application**, page 367.

Coding the `UICommandTarget()` Method

This method contains a `Select` statement, with a `Case` statement for every command ID that is handled by the command handler. You can add any code in these `Case` statements to implement the handling of a specific command.

The following example is an excerpt from a command handler designed to update a data row:

Sample CommandHandler code to update a row (record)

```
Select Case Cmd.ID
Case CMD_ACTIONS_UPDATE

    ' For each selected row in the cached data ...
    For SelRow = 1 To m_BrowseBase.Cache.SelectedCount

        ' Create and initialize a new Order Maint Object.
        Set maintObj = New Order
        Set maintObj.Dispatcher = CreateDispatcher()

        ' Initialize the Key in the Order Maint Object from
        ' selected row from the Visual Basic browse object's cached data.
        maintObj.Field("ORDER-NUMBER") = _
            m_BrowseBase.Cache.GetValue("ORDER-NUMBER", _
                SelRow, _
                BR_SELECTED_DATA, _
                BR_RAW_DATA)

        ' Move the KeyData from the KeyPDA to the ObjectPDA.
        maintObj.MoveByNameKey MOVE_DATA_TO_KEY

        ' Create a new Order Maint Form.
        Set frm = New frm_Order

        ' Link the Order object to the Order Form.
        Set frm.InternalObject = maintObj

        ' Display this form.
        frm.Show
    Next
```

Marking Updated Rows Using the UpdateListViewIcons Method

If your command handling changes affect the data displayed on the browse dialog when the user executes the command, decide how to reflect the updated data in the browse form. You can use the State property of a BrowseDataRow to mark the row as being updated. This property is used by the browse dialog when its Form_Activate event is triggered.

Alternatively, you can programmatically refresh the browse dialog's ListView with small icons by calling the `UpdateListViewIcons` method in the browse form. If a State ID has been assigned to a row, the browse dialog checks to see if this is the ID of a small icon in a global image list, found on the browse form. If the State ID matches the ID of one of the small icons in the image list, the icon is placed beside the row on the browse dialog.

Example code for marking updated rows with small icons

```
' Mark a row in the browse object as being "Updated" with a small
' icon.
  m_BrowseBase.Cache.Rows.SelectedItem(Index).State = _
  BR_MARK_ROW_UPDATED

' Refresh the browse dialog's listview to display small icons beside
' rows that have been updated.
m_BrowseForm.UpdateListViewIcons
```

MOVING EXISTING APPLICATIONS TO CONSTRUCT SPECTRUM

This chapter describes how to move existing Natural Construct-generated server-based applications to a client/server architecture using the Construct Spectrum models. To move existing Natural applications to a client/server architecture without using the models, see **Creating Spectrum Applications Without the Client Framework**, page 297, in *Construct Spectrum Programmer's Guide*.

The following topics are covered:

- **Overview**, page 246
- **Moving Natural Construct Object Applications**, page 247
- **Moving Non-Object Natural Construct Applications**, page 248

Overview

There are two scenarios that you could encounter when moving your Natural Construct-generated applications to Construct Spectrum:

- moving applications created with the Natural Construct object models (Object-Maint-Subp and Object-Browse-Subp)
- moving applications created without the Natural Construct object models

The object models were available in Natural Construct version 3.1.1 and later. These models enable you to generate encapsulated applications. Applications created with Construct Spectrum take advantage of this object approach.

Moving Natural Construct Object Applications

If you have existing Natural Construct applications developed with the object models (Object-Maint-Subp and Object-Browse-Subp), much of the work involved in creating a client/server application has already been completed.

To create a Construct Spectrum client/server application from existing Natural Construct Object applications, complete the following steps:

- 1 Set up your server environment.
For information about setting up your server environment, see **Are You Ready?**, page 126.
- 2 Perform optional Predict set up.
For more information, see **Setting Up Predict Definitions**, page 47, in *Construct Spectrum Programmer's Guide*.
- 3 Regenerate your Object-Maint-Subp modules and generate the remaining client/server modules.
For more information, see **Using the Super Model to Generate Applications**, page 93.
- 4 Set up and run your Construct Spectrum project.
For more information, see **Creating a Construct Spectrum Project**, page 123.

Moving Non-Object Natural Construct Applications

Natural Construct applications generated prior to Natural Construct release 3.1.1 or those generated with the Maint and Browse models must be modified to conform to the object-based structure required by Construct Spectrum.

- To create a Construct Spectrum client/server application from non-object Natural Construct applications, complete the following steps:
 - 1 Set up your server environment
 - 2 Evaluate your application data
 - 3 Perform optional Predict set up
 - 4 Generate the client/server modules
 - 5 Update your subprograms with existing business logic
 - 6 Set up and run your Construct Spectrum project

These steps are explained in detail in the rest of this chapter.

Step 1 — Set Up Your Server Environment

Before moving your application, ensure that your server is set up so that you can create and use client/server applications with Construct Spectrum.

To set up your server, perform the steps outlined in **Are You Ready?**, page 126.

Step 2 — Evaluate Your Application Data

Determine whether the files and fields that define your application data conform to an object-based relational database structure. If they do not, modify them to conform to this structure to take advantage of the Object-Maint models. For example, you must determine which database files should logically be grouped into business objects and establish relationships between related files and fields.

For information about organizing your database files in an object-based and relational manner, see **Design Methodology**, page 231, and **Use of Predict in Natural Construct**, page 941, in *Natural Construct Generation User's Manual*.

Step 3 — Perform Optional Predict Set Up

Some Predict set up tasks relate specifically to Construct Spectrum. For example, you can attach special keywords to a field to define its corresponding GUI control on the client dialog. These tasks are optional because Construct Spectrum applies default logic to determine how each field will be implemented on the client.

For information about these tasks, see **Setting Up Predict Definitions**, page 47, in *Construct Spectrum Programmer's Guide*.

Tip: Postpone these optional tasks until you have created and tested at least a first iteration of your client/server application and are ready to fine-tune it.

Step 4 — Generate the Client/Server Modules

To get an iteration of your client/server application up and running quickly, use the super model to generate modules for your client/server application. Generate modules for each business object, such as a Customer object and an Order object.

Generate the modules by selecting the models listed in the following table. The first four models generate the modules required for maintenance services, such as updating or adding Customer records. The remaining three models generate the modules required for browse services, such as looking up and selecting a customer record for an action.

Model	Module	Source Type
Object-Maint-Subp	Object maintenance subprogram and required PDAs	Natural subprogram
Subprogram-Proxy	Object maintenance subprogram proxy	Natural subprogram
VB-Maint-Object	Visual Basic maintenance object	Visual Basic class
VB-Maint-Dialog	Visual Basic maintenance dialog	Visual Basic form
Object-Browse-Subp	Object browse subprogram and required data areas	Natural subprogram
Subprogram-Proxy	Object browse subprogram proxy	Natural subprogram
VB-Browse-Object	Visual Basic browse object	Visual Basic class

Tip: Although you can generate all of the models listed in the previous table separately, use the super model to create a first iteration of your application as quickly as possible. Additionally, the models must be generated in the order shown in the previous table. The super model automatically generates these models in the correct order.

For information about using the super model, see **Using the Super Model to Generate Applications**, page 93.

Step 5 — Update Your Object Subprograms with Existing Business Rules

You must update your newly generated object maintenance modules with any business rules from your previous applications — those applications that were created without the Object Maintenance model. You must compare the business rules, which are contained in the user exits, in your previous application and decide how they should be incorporated into the user exits in your new application.

As you complete the procedure described below, see **Object-Maint Models**, page 479, in *Natural Construct Generation User's Manual*. This chapter contains information about generating an object maintenance subprogram and working with its user exits.

- To update your object maintenance subprogram with business rules:
 - 1 Regenerate the maintenance subprogram module with the Object-Maint-Sub model.
 - 2 Update the user exits with your business rules and compile the subprogram.

Considerations for Implementing Business Rules

When you have a working client/server application and are ready to refine your application, pay special attention to the procedures devoted to refining the implementation of your business rules. For information about implementing business rules, see **Validating Your Data**, page 327.

Because your client/server application was initially a non-object application, you probably have all of your business rules coded in the Maint model user exits. Consider placing as many of these rules as possible in other locations, such as:

- The Predict verification rules linked to your field definitions
- The Visual Basic maintenance object user exits
- The object maintenance subprogram user exits

For example, some verification rules can be implemented or duplicated on the client through the Visual Basic maintenance object. Business data types can also be used to validate data. These techniques improve the performance of your application because validations occur on the client, therefore, avoiding a call to the server.

Tip: Validations set up on the client should also be implemented on the server if users can access your application from a non-GUI environment such as a character-based display terminal. This ensures that validations are consistent no matter where the application is accessed from.

Step 6 — Set Up and Run Your Construct Spectrum Project

Once your application client and server modules have been generated on the server, set up a Construct Spectrum project on the client using the Construct Spectrum Add-In. Then download the client modules to your project, run the project, test it, and modify it as required.

These steps are described in **Creating a Construct Spectrum Project**, page 123.

UNDERSTANDING AND CUSTOMIZING THE CLIENT FRAMEWORK

This chapter describes how to customize the client framework supplied with Construct Spectrum while developing your Construct Spectrum application. It describes what each framework component is, where you use it, a conceptual overview of how it works, and procedures for customizing the component.

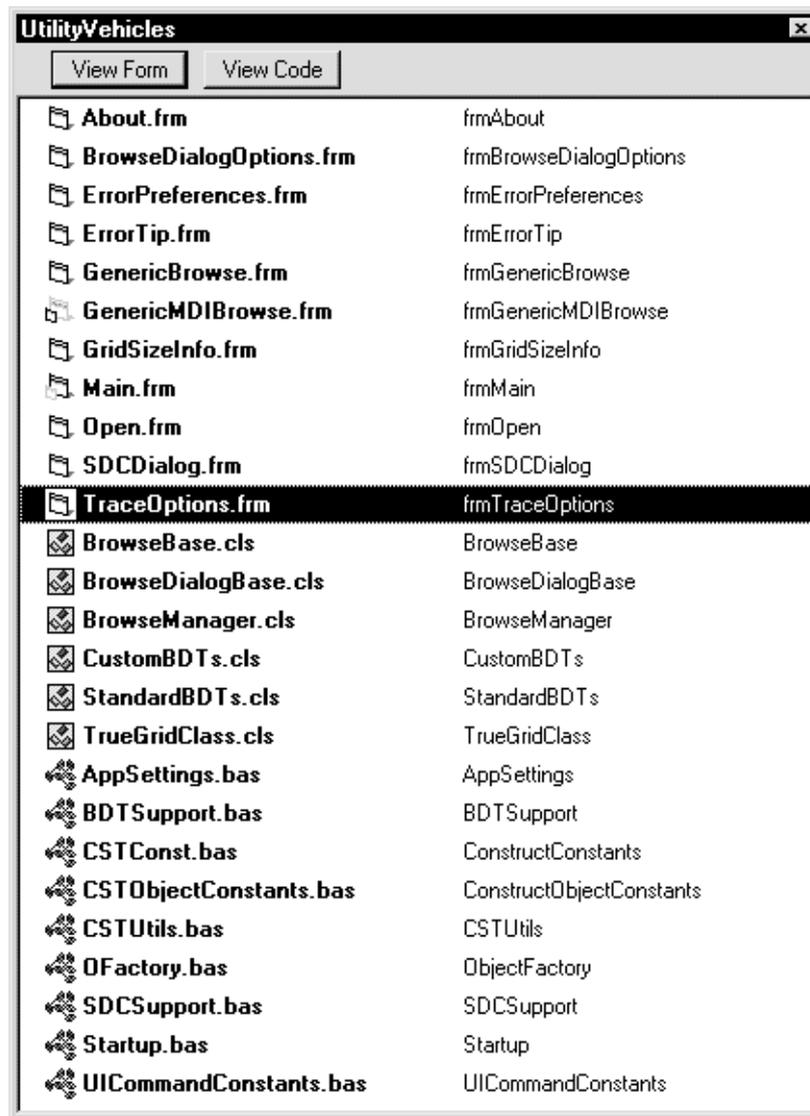
The following topics are covered:

- **Introduction to the Client Framework**, page 254
- **About Box**, page 257
- **Application Preferences**, page 260
- **Application Settings**, page 262
- **Browse Support**, page 265
- **Internationalization Support**, page 267
- **Maintenance Classes**, page 268
- **Menu and Toolbar Support**, page 270
- **Multiple-Document Interface (MDI) Frame Form**, page 301
- **Object Factory**, page 302
- **Spectrum Dispatch Client Support**, page 319
- **Utility Procedures**, page 323

Introduction to the Client Framework

Construct Spectrum automatically adds the client framework components to a standard Visual Basic project when you select Create New Project from the Construct Spectrum Add-In menu. The client framework is made up of many files that display in your application's project window. Each type of client framework component consists of one or more Visual Basic forms, modules, or classes. The following illustration shows an example of the client framework components for a Spectrum project.

For more information about creating a new project, see **Creating a Construct Spectrum Project**, page 123.



Client Framework Components of a Construct Spectrum Project

These files are grouped into logical client framework components. The components are described in this chapter and referred to throughout the Construct Spectrum documentation.

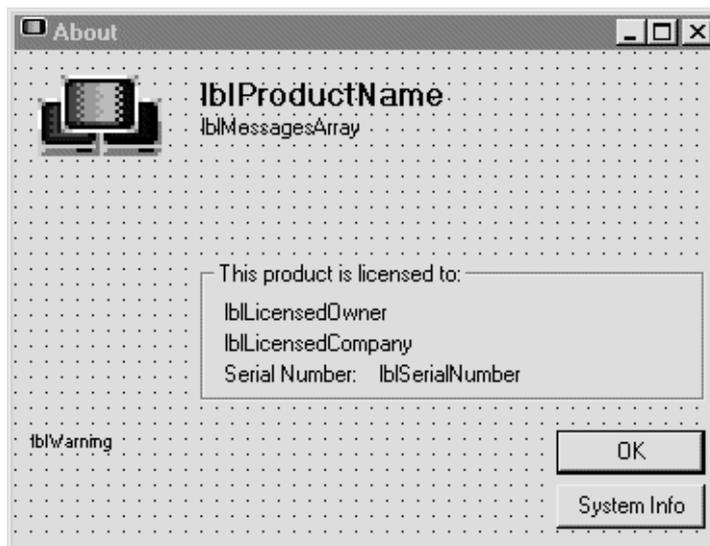
Additional client framework components are provided in an OLE automation server (CSTVBFW.dll) as classes. You can browse these OLE classes by selecting Object Browser from the View menu in Visual Basic.

Each component is described in more detail in the following sections.

About Box

The client framework includes a standard About box form. This form contains an icon, application title, application version information, licensed user and company name, serial number, copyright notices, and a System Info button to invoke the standard Windows system information applet.

The user invokes the About box by selecting the About command on the Help menu:



Default About Box Supplied with the Construct Spectrum Client Framework

You can customize the About box as desired. For example, you can include your application's icon, product name, company name, trademark, or copyright notices.

Component	Description
About.frm	Contains the About box form.

The `lblMessagesArray`, `lblLicensedOwner`, `lblLicensedCompany`, `lblSerialNumber`, and `lblWarning` values are place holders for custom messages you code in the Load event for `About.frm`.

Customizing the About Box

The About box provided with the client framework is available for you to customize for each of your applications.

Tip: To customize the About box, edit the default About box and use the **Save As** command on the **File** menu to save the tailored About box to your project directory.

Several features can be customized as described in the following table:

To Change	Follow this Procedure
Application name and window title	<ol style="list-style-type: none"> 1 Open the <code>AppSettings.bas</code> file. 2 Change the <code>gAppSettings.ApplicationName</code> variable to change the application name that is displayed at the top of the dialog.
Copyright notice	<ol style="list-style-type: none"> 1 Open the <code>Form_Load</code> event file. 2 Change the <code>lblMessages</code> variable by adding one or more lines of text to change the copyright notice. <p><i>Note: The dialog automatically grows vertically to accommodate any amount of text assigned to this variable.</i></p>
Icon	<ol style="list-style-type: none"> 1 Open the Form editor. 2 Load a different bitmap into the <code>Picture</code> property of the <code>imgApplicationBitmap</code> control to change the icon that is displayed in the upper left corner of the About box.

To Change	Follow this Procedure (continued)
Licensed owner, Company, and Serial Number	<ol style="list-style-type: none">1 Open the Form_Load event file.2 Change the text assigned to the lblLicencedOwner, lblLicensedCompany, and lblSerialNumber label controls. <p><i>Note: The client framework does not provide any specific functionality for licensing your applications. These label controls are informational only.</i></p>
Version text	<ol style="list-style-type: none">1 Open the Form_Load event file.2 Change the lblMessages variable by adding one or more lines of text to change the version text. <p><i>Note: The dialog automatically grows vertically to accommodate any amount of text assigned to this variable.</i></p>

Application Preferences

The application preferences client framework components are made up of a group of classes that allow you to define the settings of each of your applications. Use these classes to add, read, and update user or application preferences.

Applications frequently require the ability to maintain persistent *settings* over multiple executions of an application. For example, you might want your application to save window positions when a user shuts down the application. When the user restarts the application, the windows would appear in the same place on the desktop. You may also want your application to save internal configuration information such as directory names or timeout values.

The application preferences provide a high-end interface for defining the metastructure of persistent settings and for reading and writing setting values. Your preferences separate settings into two logical categories: user-specific settings and application settings. Each user ID that logs on to Windows has its own copy of the user-specific settings. Application settings are constant for all users.

The metastructure for settings can also be hierarchical, similar to a directory tree on a disk. Each node on the settings tree structure can contain any number of settings or sub-nodes (analogous to files and sub-directories, respectively). A sub-node itself can contain settings and sub-nodes. This makes it easy to group settings in the most appropriate structure.

The application preferences use the Windows registry to store the metastructure and the values of all the settings. The Windows registry is encapsulated in the implementation of the application preferences and is not exposed through the public interfaces of the settings' classes. This insulates the application from the specific requirements of reading and writing to a specific storage medium.

The following table describes the application preferences client framework components supplied with CSTVBFW.dll.

Component	Description
Setting	Creates and manipulates an individual setting
SettingList	Creates and manipulates a SettingList, which is an aggregation of SettingLists and Settings objects
SettingLists	Contains a collection of SettingLists
Settings	Contains a collection of Settings

For more information about customizing the application preferences, see **Understanding Application Preferences**, page 27, in *Construct Spectrum Reference Manual*.

Application Settings

The application settings client framework components allow you to specify your application's window title and other values that control how the application starts. These values are used by other client framework components, including the About box, the Spectrum Dispatch Client, and the Construct Spectrum Add-In.

Component	Description
AppSettings.bas	Contains the application-specific settings, such as the application name, main library, and whether to force the user to logon at application startup.
Startup.bas	Contains the Sub Main procedure and other global variables. Every Construct Spectrum application has one Sub Main procedure which is the first procedure that gets executed when your application starts running.

For more information, see:

- **About Box**, page 257
- **Spectrum Dispatch Client Support**, page 319
- **The Development Process**, page 33.

Customizing the Application Settings

The InitAppSettings procedure in the AppSettings.bas file contains settings that determine the name of the application, how the application starts up (whether the Logon form is displayed), and where application preferences are stored.

You can change the InitAppSettings by editing this procedure.

Example of a customized InitAppSettings procedure

```
Public Sub InitAppSettings()  
  
    With gAppSettings  
        .ApplicationName = "Construct Demo Application"  
  
        .ForceLogonAtStartup = False  
        .RememberUserID = True  
  
        .RegistryKey = "Software\SAGA SOFTWARE\CST"  
  
        ' Add-In Defaults  
        .DefaultLibrary = "CSTDemo"  
        .DBID = 17  
        .FNR = 38  
    End With  
  
End Sub
```

These settings are described in the following table:

Setting Name	Description
ApplicationName	Name displayed in the MDI frame form's title bar and the About box.
ForceLogonAtStartup	If True, the Logon dialog is displayed when the application is started. This option is useful when more than one person uses the same PC and you want to ensure that each person logs on using their own user ID.
RememberUserID	If True, the client framework saves the most recent user ID in the Windows Registry and recalls it when displaying the Logon dialog.
RegistryKey	Root node in the Windows Registry where application preferences are saved. These preferences are saved in HKEY_CURRENT_USER under this key.

Setting Name	Description (continued)
DefaultLibrary	Name of the main Natural library with which this application is associated. Construct Spectrum uses this setting to derive the name of the primary library image file containing the Natural data area and application service definitions used by the application.
DBID, FNR	Default database ID (DBID) and file number (FNR) for the download and upload functions for the Construct Spectrum Add-In.

Browse Support

The browse support client framework components are used to implement the browse dialog, a generic browse form used to display all browses.

The following table describes the browse support components supplied with Construct Spectrum. All of these components are stored in the CSTVBFW.dll, with the exception of classes (.cls), dialogs (.frm), and standard module files (.bas) which are included as part of the client framework in your application's project window.

Component	Description
ApplicationControl	Contains the references to the browse dialog's button, including its tag, index, command handler, caption, and button.
ApplicationControls	Contains a collection of browse dialog's application control objects.
BrowseBase.cls	Contains all of the code common to generated browse objects and is a client component accessible in source code format in the project window.
BrowseDataCache	Stores browse data.
BrowseDataColumn	Contains definitions of a table column.
BrowseDataColumns	Contains a collection of columns.
BrowseDataRow	Contains definitions and values of a table row.
BrowseDataRows	Contains a collection of rows.
BrowseDialogBase.cls	Contains all of the code common to both the MDI child and standalone versions of the browse dialog and is a client component accessible in source code format in the project window.
BrowseDialogOptions.frm	Allows users to customize the appearance of the browse dialog. It is a client component accessible in source code format in the project window.

Component	Description (continued)
BrowseManager.cls	Simplifies using the browse dialog for common functions such as selection of a foreign key value.
ColumnDisplay	Contains definition data for a displayed column, including ColumnName, ColumnCaption, ColumnWidth, and Visible.
ColumnsDisplay	Contains a collection of browse dialog's ColumnDisplay objects.
FieldKey	Defines a field used as a component in a logical key.
FieldKeys	Contains a collection of field keys.
GenericBrowse.frm	Contains the standalone version of the browse dialog and is a client component accessible in source code format in the project window.
GenericMDIBrowse.frm	Contains the MDI child version of the browse dialog and is a client component accessible in source code format in the project window.
KeyMatch	Defines a search key's associated text box attributes, including FieldName, ColumnIndex, ControlIndex, Visible, Enabled, Locked, Validated, and Fixed.
KeyMatches	Contains a collection of KeyMatch objects.
LogicalCombo	Defines an internal combo box object.
LogicalKey	Defines a key used to browse a database table.
LogicalKeys	Contains a collection of logical keys.

For more information, see **Overview of the Browse Dialog**, page 216, in this manual and **Browse Classes**, page 57, in *Construct Spectrum Reference Manual*.

Internationalization Support

The internationalization support client framework components make it easy to create applications that will be deployed in more than one language. These internationalization components enable you to develop internationalized applications.

The following table describes the internationalization support client framework components supplied with CSTVBFW.dll:

Component	Description
Resource	Reads resources from resource files.
ResourceGroup	Returns a list of resources in a resource group.

These client framework components provide you with the ability to store text and graphics appearing throughout the application separate from the compiled executable. This allows you to change them without accessing the source code of the application. Forms are designed to contain as little code as possible to provide this feature.

Tip: You do not need to build internationalization components into your design when creating small applications or applications that will only ever be used in one locale. These internationalization components are optional.

For more information about internationalization support, see **Internationalizing Using the Client Framework**, page 369.

Maintenance Classes

These client framework components allow you to manipulate items in combo boxes using the key and description, as well as use the grid to change the look of your generated maintenance dialogs.

The following table describes the generated maintenance dialog client framework components supplied with Construct Spectrum:

Component	Description
ComboClass	Contains a key list and a descriptive list that map to a combo box. It includes methods which allow you to access their information, including the Add and Load methods.
GridSizeInfo.frm	Helps the application developer size the grid columns to the best width. This form is displayed from a generated maintenance dialog's Activate event.
TrueGridClass.cls	Simplifies the use of the TrueDBGrid control in unbound mode.

For more information, see **Strategies for Customizing a Maintenance Dialog**, page 155, in this manual and **Maintenance Classes**, page 249, in *Construct Spectrum Reference Manual*.

Grid Support

To display array data and data from secondary and tertiary files, generated maintenance dialogs use the Apex TrueDBGrid custom control. The grid client framework components centralize some of the code required by TrueDBGrid so you do not have to repeat code in each generated maintenance dialog.

The client framework provides a TrueDBGrid helper class containing most of the mundane code required to use this control in unbound mode, significantly reducing the amount of code you must provide with the form.

Note: Using a TrueDBGrid control in unbound mode usually requires many lines of event code to handle displaying and editing data, inserting and deleting rows, and setting cell-level attributes such as color.

At design time, you only need to instantiate this class for each TrueDBGrid control on the form and delegate the important events (such as UnboundReadData, UnboundWriteData, and FetchCellStyle) to the equivalent methods in the class. At runtime, you can load the helper class instance with data that will be displayed in the cells of the grid.

For more information, see **Strategies for Customizing a Maintenance Dialog**, page 155, in this manual and **Maintenance Classes**, page 249, in *Construct Spectrum Reference Manual*.

Menu and Toolbar Support

The menu and toolbar client framework components allow you to dynamically change their states between enabled and disabled, and checked and unchecked. The menu and toolbar command classes provide a robust mechanism for locating and calling the code that will execute when the user selects a menu command (such as File | Open) or clicks a toolbar button.

In a multiple-document interface (MDI) application, there is only one menu bar on the MDI frame window with typically one or more toolbars. In the Construct Spectrum client framework, the MDI frame window “owns” the menu bar and toolbars. It contains the code that is executed when the user selects a menu command or clicks a toolbar button. However, what the executing code does often depends on what type of MDI child window is active. Often you will find it more appropriate to have the MDI child window itself contain the code that does the actual processing of the command. This allows the MDI frame window to be generic and contain only processing that is independent of the active MDI child window.

This client framework component allows you to design the menu and toolbar structure of the entire application on the MDI frame form, and then program each MDI child window to “hook into” the menu commands and toolbar buttons it wants to process itself. This improves functionality for the user and reduces your maintenance.

The menu and toolbar command-handling framework components implement a mechanism that centralizes the code required to determine if a menu command needs to be enabled or disabled, and checked or unchecked.

The following table describes the menu and toolbar client framework components supplied with Construct Spectrum:

Component	Description
UICommands	Class that implements menu and toolbar command handling. UICommands is stored in CSTVBFW.dll.
UICommandConstants. bas	File that defines the command IDs used to uniquely identify an end-user function in the application.
UICmd	Class containing information about a single command. UICmd is stored in CSTVBFW.dll.

Understanding Menu and Toolbar Command Handling

This section provides a conceptual overview of the command handlers that you need to understand before beginning to customize your application's menu and toolbar. The following section describes the steps to take to customize your menu and toolbar using the client framework.

The client framework classes that allow menu controls and toolbar buttons to be programmed to send application-specific commands such as FileOpen, EditPaste, or GridInsertRow are described in this section. These commands are intercepted by command handlers, which can be any form or object in the application. The command handler can also automatically update the enabled or disabled state and checked or unchecked state of menu commands and toolbar buttons.

The MDI frame, the browse dialog, and the generated maintenance dialogs all use this command handling to process menu clicks and toolbar button clicks in a single, unified fashion.

This section:

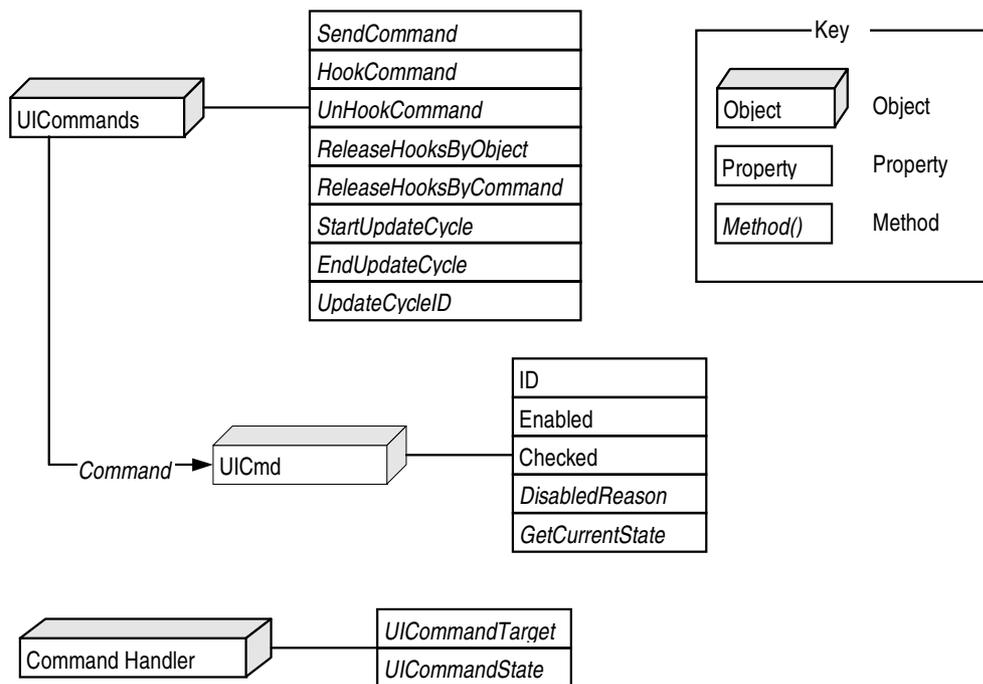
- Provides a summary of the classes
- Explains how to define, send, and handle commands
- Describes how to update user-interface controls
- Explains additional methods for command handling

For more information about menu and toolbar support, see **Menu and Toolbar Support**, page 270.

For more information about using a command handler to customize your browse dialogs, see **Understanding Browse Command Handlers**, page 238.

Class Summary

This section illustrates the classes that implement the command handler.



Classes in the Command Handler

The following sections describe many of the classes and their associated methods used to tailor the menu controls and toolbar buttons using the command handler.

Defining, Sending, and Handling Commands

This section describes how these application-specific commands are defined, how menus and toolbars are programmed to send the commands, and how they are intercepted by command handlers.

The steps required to define, send, and handle menu and toolbar commands are summarized in the following table:

Steps	Description
Step 1	Declare a global instance of the <code>UICommands</code> class. This class is part of the <code>CSTVBFW.dll</code> OLE automation server called Construct Spectrum framework classes.
Step 2	Define the commands.
Step 3	Code menu and toolbar events to send the commands.
Step 4	Code the command handlers.
Step 5	Link the commands to the command handlers.

Each of these steps is described in the following sections.

Step 1 — Declare a Global Instance of the `UICommands` Class

Declare a global variable of the `UICommands` class. This class is the primary interface to this command-handling client framework component. This variable will be used by various client framework components of the application.

Example of declaring a global variable

```
Public gUICmds As New UICommands
```

Note: The `UICommandConstants.bas` client framework component declares this variable.

Step 2 — Define the Commands

Define the application-specific commands your menu items and toolbar buttons will be sending. You will define these commands by defining named constants:

```
Public Const CMD_FILE_NEW As String = "FileNew"  
Public Const CMD_FILE_OPEN As String = "FileOpen"  
Public Const CMD_FILE_SAVE As String = "FileSave"  
...
```

These constants are called command IDs. Their values are entirely up to you; your code will never refer to the values directly, only the constant names. Define one command ID for each unique menu and toolbar command.

Step 3 — Code Menu and Toolbar Events to Send the Commands

Before you begin this step, ensure that your application has a menu or a toolbar structure from which you intend to send commands.

- To code menu events to send commands:
 - Write Click events for the menu controls.
- To code toolbar events to send commands:
 - Write ButtonClick events for the toolbar controls.

Example of coding the menu and toolbar events for three commands

```
Private Sub mnuFileNew_Click()  
    gUICmds.SendCommand CMD_FILE_NEW  
End Sub  
  
Private Sub mnuFileOpen_Click()  
    gUICmds.SendCommand CMD_FILE_OPEN  
End Sub  
  
Private Sub mnuFileSave_Click()  
    gUICmds.SendCommand CMD_FILE_SAVE  
End Sub  
  
...  
  
' For toolbar buttons, use the Tag property to store the  
' command ID you want the button to send.  
  
Private Sub Form_Load()  
    With tbrMain  
        .Buttons("NEW").Tag = CMD_FILE_NEW  
        .Buttons("OPEN").Tag = CMD_FILE_OPEN  
        .Buttons("SAVE").Tag = CMD_FILE_SAVE  
        ...  
    End With  
End Sub  
  
Private Sub tbrMain_ButtonClick(ByVal Button As Button)  
    If Button.Tag <> "" Then  
        gUICmds.SendCommand Button.Tag  
    End If  
End Sub
```

As you can see from this example, you can easily send the same command from both a menu control and a toolbar button. The event code uses the `SendCommand` method of the `UICmds` class to send a specific command ID from each control. In the next step, define the command handlers that receive these commands.

Step 4 — Code the Command Handlers

Provide the code that will be executed for each command. This code will reside in a command handler object, which can be a Visual Basic form, a Visual Basic class, or an OLE object. The only requirement for this object is that it must have a public method called `UICommandTarget` with the following declaration:

```
Public Sub UICommandTarget (Cmd As UICmd, ByRef ForwardToNext As Boolean)
```

When a menu control or toolbar button's click event calls `SendCommand`, the `UICommands` class eventually calls the `UICommandTarget` method. Into this method is passed a `UICmd` object which contains information about the command received.

`UICommandTarget` usually has a `Select Case Cmd.ID` statement so it can handle more than one command and perform specific processing for each command.

Example of coding a command handler

```
Select Case Cmd.ID
Case CMD_FILE_NEW
    ' Processing for the File|New command.
    ' ...
Case CMD_FILE_OPEN
    ' Processing for the File|Open command.
    ' ...
Case CMD_FILE_SAVE
    ' Processing for the File|Save command.
    ' ...
...
End Select
```

Step 5 — Link the Commands to the Command Handlers

Next, tell the `UICmds` class what the command handler is for each command ID. This action is called *hooking a command*. When you hook a command, specify the command handler object and a list of command IDs.

Example of linking the command handler to the command ID

```
With gUICmds
    .HookCommand frmMDIFrame, CMD_FILE_NEW, CMD_FILE_OPEN, _
        CMD_FILE_SAVE ...
```

End With

The `HookCommand` method of the `UICmds` class links the command handler with one or more command IDs. This method can be called any time during the execution of the program. Once a command has been hooked by a command handler, the link is established between the GUI control that sends the command and the code that receives and processes the command.

More than one object can hook a given command ID. The `UICmds` class stores a list of command handlers for each command ID (the command handler list). The last command handler to be hooked to a command ID is called first. If this command handler decides not to perform the processing for the command, it can set the `ForwardToNext` output parameter to `True` before returning, to tell the `UICmds` class to send the command to the next command handler (the one that hooked the command second last). This sequence continues until `ForwardToNext` is set to `False` (the default) or all command handlers in the list have been called.

If a command handler is hooked to a command ID and `HookCommand` is called again for the same command handler and command ID, the command handler will be moved to the front of the list, instead of being in the list twice.

The `SendCommand` method in the `UICommands` is actually implemented as shown in the following pseudocode.

Pseudocode demonstrating how the `SendCommand` method works

```
Sub SendCommand(CmdID As Variant)
    Look up the command handler list for the given CmdID
    For each object, cmdtarget, in the list
        Set ForwardToNext to False
        Call cmdtarget.UICommandTarget(Cmd, ForwardToNext)
        If ForwardToNext is False
            Exit the loop
        End If
    End For
End Sub
```

When an object no longer wants to hook a command ID, you can call the `UnHookCommand` method of `UICommands` to break the link between the command ID and the command handler. `UICommands` will remove the object from the command handler list.

Example of unlinking the command ID and the command handler

```
With gUICmds
    .UnHookCommand frmMDIFrame, CMD_FILE_NEW, CMD_FILE_OPEN, _
        CMD_FILE_SAVE ...
End With
```

Updating User Interface Controls

When a user opens a menu, the menu commands that are not currently valid are visibly disabled. The object that processes a command (the command handler object) also decides whether or not the command is valid. The `UICommands` class implements a mechanism whereby it asks the command handler object whether a given command is valid or not. Modify the event code to customize the actions that are performed when the user selects a menu item or clicks a toolbar button.

The steps required to update user interface controls are summarized in the following table:

Steps	Description
Step 1	Code events to update the menu controls.
Step 2	Code the logic in the command handler that determines the state of a command.
Step 3	Code events to update the toolbar buttons.

Each of these steps is described in the following sections.

Step 1 — Code Events to Update the Menu Controls

Write event code that enables or disables the menu items just before the menu is displayed to the user. Take advantage of the Click event of a menu control, such as the File menu or Edit menu, as a place to include your event code. Visual Basic calls this event just before displaying the menu to the user.

Example of updating menu controls before the menu is displayed to the user

```

Private Sub mnuFile_Click()
    SetMenuState mnuFileNew, CMD_FILE_NEW
    SetMenuState mnuFileOpen, CMD_FILE_OPEN
    SetMenuState mnuFileSave, CMD_FILE_SAVE
    ...
End Sub

Private Sub SetMenuState(mnu As Menu, CmdID As Variant)
    With gUICmds.Command(CmdID)
        .GetCurrentState
        mnu.Enabled = .Enabled
        mnu.Checked = .Checked
    End With
End Sub

```

where:

Command	returns a UICmd object that contains information for a given CmdID. This is the same UICmd object that was passed to UICmdTarget.
GetCurrentState	causes UICmds to call the command handler object again, but this time the command handler will not process the command, but will return whether or not the command is valid and whether or not it should be checked. These settings can then be read from the Enabled and Checked properties when GetCurrentState returns.

If you disable a menu control in its parent's Click event (the parent is the submenu that contains the menu control), Visual Basic disables the menu control when the menu is displayed. You can do the same thing with the Checked property for the menu control.

Step 2 — Code the Logic that Determines the State of a Command

The logic that determines whether a command is enabled or disabled and checked or unchecked resides in the command handler in a public method called `UICommandState`. It must have the following declaration:

```
Public Sub UICommandState(Cmd As UICmd, ByRef ForwardToNext As Boolean)
```

The `UICommandState` method is called by `UICommands` whenever `GetCurrentState` is called as shown in the following example:

Example of using the `UICommandState` method

```
With Cmd
  Select Case .ID
    Case CMD_FILE_NEW
      ' Code that determines if this command is valid.
      .Enabled = some condition
    Case CMD_FILE_OPEN
      ' Code that determines if this command is valid.
      .Enabled = some condition
    Case CMD_FILE_SAVE
      ' Code that determines if this command is valid.
      If some condition Then
        .Enabled = True
      Else
        .Enabled = False
        .DisabledReason = "the document has not changed since" & _
          "it was saved"
      End If
    End Select
  End With
```

In the previous example, the `Select Case Cmd.ID` statement enables the method to handle more than one command and provide specific processing for each command. The `Enabled` property can be set to `True` or `False`. If `Enabled` is set to `False`, you can also set the `DisabledReason` property to provide a message to the user explaining why the command is not available. You also have the option of setting the `Checked` property to `True` or `False`.

Similar to `UICommandTarget`, if the command handler object is not required to determine the state of the command, it can set the `ForwardToNext` parameter to `True` before returning, instructing `UICommands` to invoke the next object in the command handler list.

If a command has at least one object in its command handler list, the object will be `Enabled` and `Unchecked`. Therefore, you only need to provide handling in `UICommandState` when you want to disable or check a command. If a command's command handler list is empty, `GetCurrentState` will return `Disabled` and `Unchecked`.

Step 3 — Code Events to Update the Toolbar Buttons

Add code to enable or disable toolbar buttons. There are several different ways to present disabled toolbar buttons to the user:

- Display a message. When the user clicks the button, either a window with a message is displayed or the message is displayed on the status bar.
- Show a toolbar button with a disabled bitmap so that the user can immediately see the button is disabled. The client framework uses this approach.

Displaying a Disabled Bitmap

If you decide to display a disabled bitmap, you must continually update the button image. To update the button image, use a `Timer` control on the form and include the event code as indicated in the following example:

Example of adding a Timer control to update the Button image

```
Private Sub tmrToolbarUpdate_Timer()  
    Dim i As Integer  
    Dim btn As Button  
  
    For Each btn In tbrMain.Buttons  
        If btn.Tag <> "" Then  
            With gUICmds.Command(btn.Tag)  
                .GetCurrentState  
                btn.Enabled = .Enabled  
                If btn.Style = tbrCheck Then  
                    btn.Value = IIf(.Checked, tbrPressed,  
                                   tbrUnpressed)  
                End If  
            End With  
        End If  
    Next  
End Sub
```

The previous example updates all the toolbar buttons that have command IDs assigned to them by the Tag property. Set the timer interval so that this event executes frequently. An interval of 250 ms ensures that the toolbar button bitmaps do not lag too far behind the application's state. Timer events are only triggered when the application becomes idle. This is advantageous because it does not take away processing time from the running application to update the toolbar buttons, but it is disadvantageous because it continues to update the toolbar button bitmaps when your application is idle.

Displaying a Message

If you decide not to update the toolbar button bitmaps continually, leave the buttons enabled and instead display a message when the user clicks on a disabled button.

Example of displaying a message after the Click event on a disabled menu item

```
Private Sub tbrMain_ButtonClick(ByVal Button As Button)
    Dim msg as string
    If Button.Tag <> "" Then
        With gUICmds.Command(Button.Tag)
            .GetCurrentState
            If .Enabled Then
                gUICmds.SendCommand Button.Tag
            Else
                msg = "This command is not available"
                If .DisabledReason <> "" Then
                    msg = msg & " because " & .DisabledReason
                End If
                DisplayStatusBarMessage msg
            End If
        End With
    End If
End Sub
```

When the user clicks on a toolbar button, the code determines whether or not the button is valid. If it is valid, it executes the event code. If it is not valid, it displays a message on the status bar, explaining to the user why the button is disabled.

Update Cycles

When the `GetCurrentState` method is called repeatedly for each menu item on a menu or for each button on a toolbar, the application's state does not change between the first call and the last call to `GetCurrentState` because the only thread of execution is busy. If, however, one of the `UICmdState` methods yields the CPU with a `DoEvents` or calls a Windows function that yields, for example, with a blocked DDE request, the Construct Spectrum application could be re-entered allowing its state to change.

Assuming that the `UICmdState` methods do not act in this way, it is possible to optimize the code that executes within these methods by using the concept of an update cycle.

During an update cycle, it is known that the application's state will not change. Therefore, at the beginning of an update cycle, you can look up all of the information about the application's state that the `UICommandState` methods will need. Instead of looking this up for each command ID that needs the information, you can look it up once, store the information in Static variables, and use it several times.

For example, the validity of the Edit menu commands Undo, Cut, Copy, Paste, Delete, and Select All, all depend on the control that currently has focus. You could write the following code to determine the state of each of these commands at once.

Example of code that determines the state of multiple commands simultaneously

```
bcanundo = False
bcancut = False
bcancopy = False
bcanpaste = False
bcandelelete = False
bcansselectall = False

Set ctl = Screen.ActiveControl
If Not (ctl Is Nothing) Then
    Select Case TypeName(ctl)
        Case "TextBox", "MaskedTextBox"
            bcanundo = (SendMessage(ctl.hwnd, EM_CANUNDO, 0, ByVal 0&) <> 0)
            bcancut = (ctl.SelLength > 0)
            bcancopy = bcancut
            bcanpaste = Clipboard.GetFormat(vbCFTText)
            bcandelelete = bcancut
            bcansselectall = (ctl.Text <> "")
        End Select
    End If
```

You may only want to run this code at the beginning of each update cycle because during the update cycle the application's state will not change. To set the update cycle, bracket the calls to `GetCurrentState` with a call to `StartUpdateCycle` and a call to `EndUpdateCycle` as shown in the following example:

Example of setting the update cycle

```
Private Sub mnuEdit_Click()  
    gUICmds.StartUpdateCycle  
  
    SetMenuState mnuEditUndo, CMD_EDIT_UNDO  
    '---  
    SetMenuState mnuEditCut, CMD_EDIT_CUT  
    SetMenuState mnuEditCopy, CMD_EDIT_COPY  
    SetMenuState mnuEditPaste, CMD_EDIT_PASTE  
    SetMenuState mnuEditDelete, CMD_EDIT_DELETE  
    '---  
    SetMenuState mnuEditSelectAll, CMD_EDIT_SELECT_ALL  
  
    gUICmds.EndUpdateCycle  
End Sub
```

StartUpdateCycle assigns an *update cycle ID* (a 32-bit integer), which will be constant until the call to EndUpdateCycle. The code that determines the state of the edit commands will now only be executed when the update cycle ID changes, as shown in the following example:

Example of changing the update cycle ID

```
Dim ctl As Control
Static llastupdateedit As Long
Static bcanundo As Boolean
Static bcancut As Boolean
Static bcancopy As Boolean
Static bcanpaste As Boolean
Static bcandelete As Boolean
Static bcansselectall As Boolean

With Cmd
  Select Case .ID
    ...
    Case CMD_EDIT_UNDO, _
      CMD_EDIT_CUT, _
      CMD_EDIT_COPY, _
      CMD_EDIT_PASTE, _
      CMD_EDIT_DELETE, _
      CMD_EDIT_SELECT_ALL
      If llastupdateedit <> gUICmds.UpdateCycleID then
        llastupdateedit = gUICmds.UpdateCycleID

        bcanundo = False
        bcancut = False
        bcancopy = False
        bcanpaste = False
        bcandelete = False
        bcansselectall = False

        Set ctl = Screen.ActiveControl
        If Not (ctl Is Nothing) Then
          Select Case TypeName(ctl)
            Case "TextBox", "MaskedTextBox"
              bcanundo = (SendMessage(ctl.hwnd, EM_CANUNDO, _
                0, ByVal 0&) <> 0)

              bcancut = (ctl.SelLength > 0)
              bcancopy = bcancut
              bcanpaste = Clipboard.GetFormat(vbCFText)
              bcandelete = bcancut
              bcansselectall = (ctl.Text <> "")
            End Select
          End If
        End If
      Select Case .ID
        Case CMD_EDIT_UNDO:          .Enabled = bcanundo
```

```
Case CMD_EDIT_CUT:           .Enabled = bcancut
Case CMD_EDIT_COPY:         .Enabled = bcancopy
Case CMD_EDIT_PASTE:        .Enabled = bcanpaste
Case CMD_EDIT_DELETE:       .Enabled = bcandeleate
Case CMD_EDIT_SELECT_ALL:   .Enabled = bcansselectall
End Select
End Select
End With
```

Additional Methods For Command Handling

This section describes other methods of `UICommands` you can use with your application.

Unhooking Commands

To remove an object from all command handler lists, regardless of command ID, you must release all references to it. The `UICommands` class provides the `ReleaseHooksByObject` method as illustrated in the following syntax example.

Syntax of the ReleaseHooksByObject Method

```
Sub ReleaseHooksByObject (HookObject As Object)
```

Pseudocode Showing How the ReleaseHooksByObject Method Works

```
For all commands
  If HookObject is in this command's command handler list
    Remove it from the list
  End If
End For
```

To empty the command handler list for a given command, use the `ReleaseHooksByCommand` method provided with the `UICommands` class:

```
Sub ReleaseHooksByCommand(CmdID As Variant)
```

where:

<i>CmdID As Variant</i>	is replaced with one of the previously-defined command IDs (for example, <code>CMD_EDIT_UNDO</code>).
-------------------------	--

Customizing the Menu and Toolbar in the Client Framework

This section describes how to tailor the menu items and the toolbar buttons. You will learn how to model your changes on the code in the client framework's multiple-document interface (MDI) frame form.

For more information about tailoring the menu items, see **Changing the Menu Structure**, page 289.

For more information about tailoring the buttons on the toolbar, see **Changing the Toolbar Structure**, page 298.

For information about how to change the states of the menu items and the toolbar between enabled and disabled, and checked or unchecked, see **Understanding Menu and Toolbar Command Handling**, page 271.

Changing the Menu Structure

The multiple-document interface (MDI) frame form in the client framework has a predefined menu structure. You may change this menu structure by following the pattern used in the MDI frame form.

For more information about the MDI frame form, the command IDs, command handlers, and update cycles, see **Understanding Menu and Toolbar Command Handling**, page 271.

The pattern supplied with the MDI frame form is implemented with the following code requirements:

- Each menu item control sends a command ID through event code. The event code for all menu item controls is identical except for the command ID constant.

Example of event code for three commands on the File menu and two commands on the Edit menu

```
Private Sub mnuFileOpen_Click()
    gUICmds.SendCommand CMD_FILE_OPEN
End Sub

Private Sub mnuFileClose_Click()
    gUICmds.SendCommand CMD_FILE_CLOSE
End Sub

Private Sub mnuFileExit_Click()
    gUICmds.SendCommand CMD_FILE_EXIT
End Sub

Private Sub mnuEditCut_Click()
    gUICmds.SendCommand CMD_EDIT_CUT
End Sub

Private Sub mnuEditCopy_Click()
    gUICmds.SendCommand CMD_EDIT_COPY
End Sub
```

- The command IDs are all defined in a global module called `UICmdConstants.bas`.

Example of UICmdConstants.bas where all command IDs are defined

```
Public Const CMD_FILE_OPEN = "FileOpen"
Public Const CMD_FILE_CLOSE = "FileClose"
Public Const CMD_FILE_EXIT = "FileExit"

Public Const CMD_EDIT_CUT = "EditCut"
Public Const CMD_EDIT_COPY = "EditCopy"
```

- When the MDI frame form is loaded, the command IDs are hooked into the command classes.

Example of hooking the commands IDs into the command classes

```
Private Sub MDIForm_Load ()
    With gUICmds
        .HookCommand Me, CMD_FILE_OPEN, _
            CMD_FILE_CLOSE, _
            CMD_FILE_EXIT

        .HookCommand Me, CMD_EDIT_CUT, _
            CMD_EDIT_COPY
    End With
End Sub
```

Note: Although you can hook all command IDs with one call to the *HookCommand* method, the previous example illustrates how to group the command IDs by category — *File commands and Edit commands*.

- Each menu item has a parent menu control. This control's Click event is triggered when the user chooses the menu. Use the Click event to enable or disable and check or uncheck each menu item.

Example of using the Click event to control menu items

```
Private Sub mnuFile_Click()
    gUICmds.StartUpdateCycle
    SetMenuState mnuFileOpen, CMD_FILE_OPEN
    SetMenuState mnuFileClose, CMD_FILE_CLOSE
    SetMenuState mnuFileExit, CMD_FILE_EXIT
    gUICmds.EndUpdateCycle
End Sub

Private Sub mnuEdit_Click()
    gUICmds.StartUpdateCycle
    SetMenuState mnuEditCut, CMD_EDIT_CUT
    SetMenuState mnuEditCopy, CMD_EDIT_COPY
    gUICmds.EndUpdateCycle
End Sub
```

The previous example calls `SetMenuState` for each item on the menu. These calls are bracketed by `StartUpdateCycle` and `EndUpdateCycle`.

- Finally, you must code the `UICommandTarget` and `UICommandState` procedures in each form that will be receiving these command IDs. You can model your procedures on the procedures used by the MDI frame form and Visual Basic maintenance objects generated with Natural Construct.

Example of UICCommandTarget and UICCommandState procedures used by the MDI frame form

```
Public Sub UICCommandTarget(Cmd As UICmd, ForwardToNext As Boolean)
    Select Case Cmd.ID

        Case CMD_FILE_OPEN
            frmOpen.Show vbModal
        Case CMD_FILE_CLOSE
            Unload Screen.ActiveForm
        Case CMD_FILE_EXIT
            Unload Me

        Case CMD_EDIT_CUT
            With Screen.ActiveControl
                Clipboard.SetText .SelText
                .SelText = ""
            End With
        Case CMD_EDIT_COPY
            Clipboard.SetText Screen.ActiveControl.SelText

    End Select
End Sub

Public Sub UICCommandState(Cmd As UICmd, ForwardToNext As Boolean)
    Dim frm As Form
    Dim ctl As Control

    Static llastupdateedit As Long
    Static bcancut As Boolean
    Static bcancopy As Boolean

    With Cmd
        Select Case .ID
            Case CMD_FILE_CLOSE
                .Enabled = False
                .DisabledReason = "there are no child windows open"
                Set frm = Screen.ActiveForm
                If Not (frm Is Nothing) Then
                    .Enabled = IsMDIChild(frm)
                End If

            Case CMD_EDIT_CUT, _
                CMD_EDIT_COPY
                If llastupdateedit <> gUICmds.UpdateCycleID Then
                    llastupdateedit = gUICmds.UpdateCycleID
                End If
            End Select
    End With
End Sub
```

```

        bcancut = False
        bcancopy = False

        Set ctl = Screen.ActiveControl
        If Not (ctl Is Nothing) Then
            Select Case TypeName(ctl)
            Case "TextBox", "MaskedTextBox"
                bcancut = (ctl.SelLength > 0)
                bcancopy = bcancut
            End Select
        End If
    End If
    Select Case .ID
    Case CMD_EDIT_CUT: .Enabled = bcancut
    Case CMD_EDIT_COPY: .Enabled = bcancopy
    End Select
End Select
End With
End Sub

```

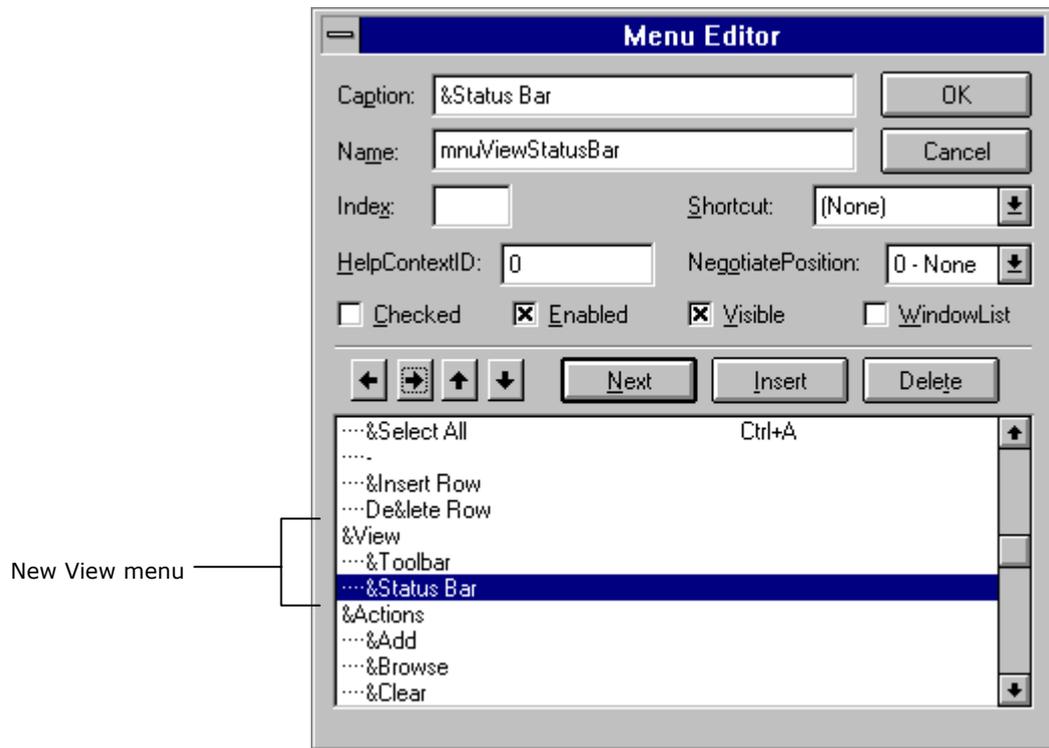
Example of Changing the Menu Bar and Its Menu Items

The following example adds a new menu called View to the menu bar and includes commands that allow you to toggle the toolbar and status bar on and off.

- To add a View menu to the menu bar with the menu items Toolbar and Status Bar:
- 1 Use Visual Basic's menu editor to add the following menu controls to the MDI frame form.

Menu Caption	Menu Control Name
View	mnuView
Toolbar	mnuViewToolbar
Status Bar	mnuViewStatusBar

Note: The rest of this example assumes the previous menu structure has been added.



Example of the New Menu View Added to the Menu Bar

2 Define command IDs in UICommandConstants.bas:

```
Public Const CMD_VIEW_TOOLBAR = "ViewToolbar"
Public Const CMD_VIEW_STATUSBAR = "ViewStatusBar"
```

The names of these constants and their values can be anything you choose, but try to follow the conventions established in the code.

3 Code the event handlers for the menu controls:

```
Private Sub mnuViewToolBar_Click()
    gUICmds.SendCommand CMD_VIEW_TOOLBAR
End Sub

Private Sub mnuViewStatusBar_Click()
    gUICmds.SendCommand CMD_VIEW_STATUSBAR
End Sub
```

4 Hook the command IDs into the command classes:

```
Private Sub MDIForm_Load ()
    With gUICmds
        ...
        .HookCommand Me, CMD_VIEW_TOOLBAR, _
            CMD_VIEW_STATUSBAR
        ...
    End With
End Sub
```

5 Add code to the Click event of the menu control on the menu bar to update the state of the menu controls:

```
Private Sub mnuView_Click()
    gUICmds.StartUpdateCycle

    SetMenuState mnuViewToolBar, CMD_VIEW_TOOLBAR
    SetMenuState mnuViewStatusBar, CMD_VIEW_STATUSBAR

    gUICmds.EndUpdateCycle
End Sub
```

6 Lastly, add code to the UICommandTarget and UICommandState procedures in the MDI frame form to handle these two new command IDs:

```
Public Sub UICommandTarget(Cmd As UICmd, ForwardToNext As Boolean)
    Select Case Cmd.ID
        ...
        Case CMD_VIEW_TOOLBAR
            tbrMain.Visible = Not tbrMain.Visible
        Case CMD_VIEW_STATUSBAR
            sbrMain.Visible = Not sbrMain.Visible
        ...
    End Select
End Sub

Public Sub UICommandState(Cmd As UICmd, ForwardToNext As Boolean)
    With Cmd
        Select Case .ID
            ...
            Case CMD_VIEW_TOOLBAR
                .Checked = tbrMain.Visible
            Case CMD_VIEW_STATUSBAR
                .Checked = sbrMain.Visible
            ...
        End Select
    End With
End Sub
```

By using the command handler, you do not need to set the menu controls' `Checked` properties when you toggle the visibility of the toolbar or status bar. Instead, read the current visibility state in the `UICommandState` method. If another piece of code changes the visibility state, that other code is not required to toggle the menu's `Checked` property.

Changing the Toolbar Structure

The toolbar follows the structure defined by the MDI frame form, just as the Menu does.

For more information about the MDI frame form structure, see **Understanding Menu and Toolbar Command Handling**, page 271.

- The toolbar is a control of type `Toolbar`, from the Windows Common Controls library, with the name `tbrMain`.
- The button arrangement is defined at design time using the `Toolbar Control Properties` dialog. The images on the buttons are stored in the `ilstMain` image list control on the MDI frame form. Each toolbar button is linked to a specific numeric index in the image list.
- The `Tag` property of each toolbar button contains the command ID that is sent by that toolbar button. The command IDs may be the same as or different than those used on the menu. These `Tag` properties are set up with the following code in the form's `Load` event:

Example of Tag properties defined in the Load event

```
With tbrMain
    .Buttons("OPEN").Tag = CMD_FILE_OPEN
    .Buttons("CUT").Tag = CMD_EDIT_CUT
End With
```

The previous example uses a string key to uniquely identify each toolbar button. This key makes it easy to get a reference to a specific toolbar button.

Note: Another way to set the Tag property is by using the Toolbar Control Properties dialog, although this solution is less desirable. First, in the Toolbar Control Properties dialog you must specify a hard-coded value in the dialog, whereas in code you would use a named constant. Second, if you hand code the value in the dialog, you cannot use Visual Basic's search function to search for it, making your code more difficult to review, change, and scan for dependencies.

- When a toolbar button is clicked, the `ButtonClick` event checks whether the button should be enabled or not, and then sends the command ID if it is enabled. This code is generic and does not have to be changed if the buttons on the toolbar are changed.

The following example uses the `ButtonClick` event to check whether the button is enabled or not and sends the command ID if it is enabled.

Example of checking the button's state

```
Private Sub tbrMain_ButtonClick(ByVal Button As Button)
    Dim msg As String

    If Button.Tag <> "" Then
        With gUICmds.Command(Button.Tag)
            .GetCurrentState
            If .Enabled Then
                gUICmds.SendCommand Button.Tag
            Else
                msg = "This command is not available"
                If .DisabledReason <> "" Then
                    msg = msg & " because " & .DisabledReason
                End If
                DisplayStatusBarMessage msg
            End If
        End With
    End If
End Sub
```

Example of Adding Buttons to the Toolbar

In this example, two new buttons are placed on the toolbar to correspond to the menu commands `Insert Row` and `Delete Row` on the `Edit` menu. These commands already have command IDs and command handlers.

For more information about defining command ID constants and command handlers, see **Example of Changing the Menu Bar and Its Menu Items**, page 294.

- To add two buttons to the toolbar:
 - 1 Display the `Image List Control Properties` dialog for the image list control called `ilstMain` and add the bitmaps of your choice to the two new buttons. Make a note of the numeric index of these bitmaps.
 - 2 Display the `Toolbar Control Properties` dialog and add the new buttons. Give each button a string key, `ToolTip` text, and assign the image number from the first step.

- 3 Set the buttons' Tag properties in the MDI frame form's Load event. The `mnuEditInsertRow` and `mnuEditDeleteRow` controls in the Click event send the command IDs `CMD_EDIT_INSERT_ROW` and `CMD_EDIT_DELETE_ROW`, respectively. Use these command IDs when assigning the Tag properties:

```
With tbrMain
    ...
    .Buttons("INSERT_ROW").Tag = CMD_EDIT_INSERT_ROW
    .Buttons("DELETE_ROW").Tag = CMD_EDIT_DELETE_ROW
    ...
End With
```

There are now two new buttons on the toolbar that behave identically to the Insert Row and Delete Row commands on the Edit menu.

Multiple-Document Interface (MDI) Frame Form

The Multiple-Document Interface (MDI) frame form includes a standard menu bar, toolbar, and status bar for your application. The MDI frame form is provided for you to use as a starting point for creating your own menu or tailoring your toolbars. You will need to customize the menu and toolbars for each application unless the MDI frame form is suitable as is.

All generated maintenance dialogs are displayed as child windows within the MDI frame form.

The following table describes the MDI frame form supplied with Construct Spectrum:

Component	Description
MDIFrame.frm	Contains the MDI frame form which includes a menu, a toolbar, and a status bar.
Menu Bar	Contains File, Edit, Actions, Window, and Help menus, each containing the standard menu commands.
Toolbar	Contains buttons that correspond to most of the menu commands and can be customized by the user.
Status Bar	Contains panels for a message, various status indicators, and the current date and time.

For more information, see **Multiple-Document Interface (MDI) Applications** in *Microsoft Visual Basic Programmer's Guide*.

Object Factory

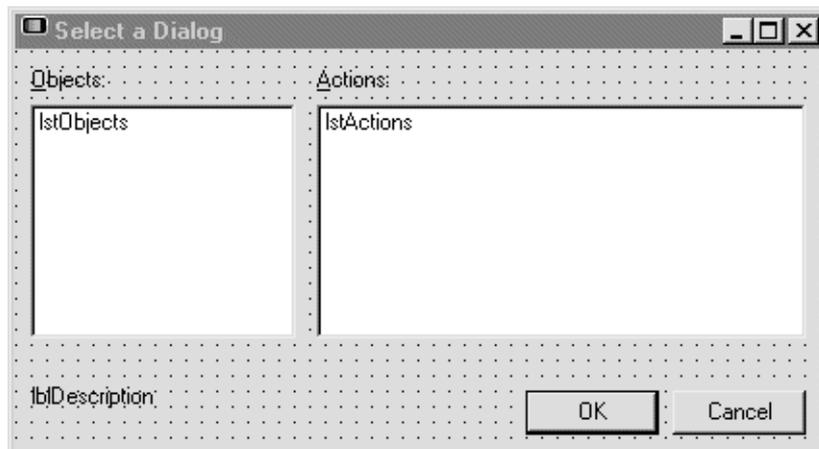
The object factory client framework components are used by many of the generated Construct Spectrum modules, as well as by other client framework objects and forms. The purpose of the object factory is to make the client portion of your application aware of all of its Visual Basic business objects and their associated actions.

The Open dialog enables you to select dialogs by using the object factory to display a list of all its available Visual Basic business objects.

The following table describes the object factory and Open dialog supplied with Construct Spectrum:

Component	Description
Open.frm	Contains the Open dialog.
OpenAction	Describes a single action of a Visual Basic business object in your application.
OpenObject	Describes a single Visual basic business object in your application.
OpenObjects	Contains all of the Visual Basic business objects in your application.
OFactory.bas	Contains the object factory.

The objects and actions are displayed in the Open dialog that is part of the client framework. The user first selects an object, then selects one of the actions for that object, and finally clicks OK to display the form.



Default Open.frm Supplied with your Construct Spectrum Project

Understanding the Open Dialog

The Open dialog provides the user with a convenient method of selecting the type of window he or she would like to open, such as “Maintain Customers,” “Display Overdue Accounts,” or “Create a New Order.” The dialog displays two lists: one showing the main business objects such as customers, accounts, and orders, and the other showing the actions available for the business object selected in the first list. For example, if “Orders” is selected in the first list, then “Maintain,” “Browse,” “Show pending orders,” and “Print end-of-day report” might appear in the second list. Each item in each list can have a short description, which is shown when the user selects the item.

You do not need to change the Open dialog. You must, however, update the object factory by providing the list of objects that should appear in the first list and the associated actions for the second list, and writing the code that is executed when each object and action combination is selected by the user. This code will then load and display a form generated by Natural Construct.

The Open dialog uses the object factory for two purposes:

- To determine which objects and actions are supported by an application
- To instantiate a form

For more information, see **Customizing the Object Factory**, page 307, and **Customizing the Menu and Toolbar in the Client Framework**, page 289.

Understanding the Object Factory

Every Construct Spectrum application contains a client framework component called the object factory. The object factory is the central repository in your application where instances of Visual Basic business objects are created for use by other portions of the application. The super model generates the initial object factory based on the objects defined to the model. Later, as new objects are added to your application, the object factory is typically extended by hand-coding new object references.

Because the creation of all Visual Basic business objects occurs in the object factory, all other application components that use the services of these objects can compile and execute, even if the Visual Basic business objects that they interact with have not yet been added to your application.

For more information about using the object factory to instantiate Visual Basic business objects that have not yet been added to your application, see **Example of Using the Object Factory**, page 306.

Application components that use the services of the object factory include:

- Construct Spectrum client framework components:
 - Open dialog
 - BrowseManager class
- For more information, see **Using the BrowseManager**, page 233, and **Understanding Browse Command Handlers**, page 238.

- Visual Basic maintenance business object
For more information about the forms and classes generated by the VB-Maint-Object model, see **Creating and Customizing Maintenance Dialogs**, page 137. For more information about the forms and classes generated by the VB-Maint-Dialog model, see **Strategies for Customizing a Maintenance Dialog**, page 155.
- Visual Basic browse business object
For more information about the forms and classes associated with Visual Basic browse business objects, see **About Browse Dialogs**, page 216 and **Understanding Browse Command Handlers**, page 238.
- Custom-created modules such as browse command handlers
For more information, see **Understanding Browse Command Handlers**, page 238 and **Understanding Browse Command Handlers**, page 238.

Application components that require a specific form or object to implement a service (for example, creating a browse dialog that allows your users to browse customer records) use the object factory. Instead of each component creating its own instances of these objects, components send a request to the object factory to create the objects and return a reference.

Using the Object Factory

The object factory exposes four procedures (functions and subroutines) that are global to your application. As you create your application, use these procedures to:

- Make the application aware of all its Visual Basic business objects.
- Create instances of Visual Basic business objects (forms or objects).
- Query the availability of Visual Basic business objects.

The following table describes the procedures in the object factory:

Service	Description
InitializeOpenDialog()	Creates a list of the application's Visual Basic business objects and the actions they support. The Open dialog uses this service.
CreateForm(<i>formID</i>) As Form	Creates a form to support a Visual Basic business object (either a Visual Basic browse object or a Visual Basic maintenance object) and returns a reference to the form. The Open dialog uses this service.
BrowserExists(<i>TableName</i>) As Boolean	Confirms with True or False whether a Visual Basic browse object exists for a database table.
GetBrowser(<i>TableName</i>) As BrowseManager	The object factory creates a specific Visual Basic browse object for a database table. Next, the specified browse object creates and initializes a browse base object. Finally, the object factory returns a reference to the BrowseManager object.

Example of Using the Object Factory

An Order Maintenance form offers the ability to invoke an Order Browse form. To accomplish this, the Order Maintenance form uses the services of the Order Browse object.

If an Order Maintenance form directly instantiates an Order Browse object (instead of using the object factory), it could not be compiled without including the Order Browse object as part of the application. However, by conditionally creating an Order Browse object with the object factory, you will be able to compile the form, even if the Order Browse object has not yet been added to the application.

At execution time, the Order Maintenance form uses a global function, `BrowserExists(tablename)`, exposed by the object factory, to determine if the object factory can create an instance of the Order Browse object. Only if the object factory returns `True` to this request does the form enable the features supported by the Order Browse object.

The `TableName` parameter used with the `BrowserExists()` function is the name of the database table implemented by a Visual Basic business object.

Customizing the Object Factory

When you add new business objects to your application, such as maintenance forms or browse objects, you must update the object factory to make the application aware of these new objects. You must either add code manually to the standard object factory module or generate a new object factory using the super model.

If you generated and downloaded the object factory (OFACTORY, although you may have given it a different name), you should be able to run your application, choose `Open` on the `File` menu, and see the objects and actions you generated.

Downloaded forms are added to your Construct Spectrum project. However, if you did not generate the object factory or if you are adding a form to an existing project, you must write a small amount of code by hand to link each new form to the client framework. Once linked, the `Open` dialog is able to load, initialize, and display the form. The following sections describe how to code the object factory by hand.

Setting Up

The client framework uses an object-action metaphor to select a particular form to display. You, as the application developer, must decide which types of objects can be manipulated by the application, such as `Customers`, `Orders`, and `Inventory`. Next you must decide which actions will be supported for each object, such as `Maintain`, `Browse`, or `Show Delinquents`. Each object-action combination will have a form associated with it, either generated or created by hand.

You must write code to define all of the objects, the actions for each object, and which form to load and initialize for each object-action combination. All of the code resides in a module called `OFactory.bas` in your Construct Spectrum project.

```

ObjectFactory
Object: (General) Proc: InitializeOpenDialog

Public Sub InitializeOpenDialog()

    ' Creates a global collection of business package descriptions.

    ' Each business package description contains the package's name,
    ' a descriptive comment, and a list of actions supported by the package.

    ' Each entry in the list of supported services, contains the service
    ' name, a description, and an ID which uniquely identifies the
    ' service within the entire application.

    Dim obj As OpenObject

    ' Create a new global instance of the OpenObjects collection.
    Set gOpenObjects = New OpenObjects

    ' Add a new business package to the OpenObjects collection.
    'Set obj = gOpenObjects.Add("Customer", "Customer")

        ' Add the services supported by this business package.
        'obj.Add "Maintenance", "Customer Maintenance", "Customer_M1"
        'obj.Add "Browse", "Browse Customers", "Customer_B1"

    ' To add a new business package: (1) Copy this code block ----- >>
    ' (2) Uncomment and modify the lines in this code block as required.
    ' Add a new business object to the OpenObjects collection.
    'Set obj = gOpenObjects.Add("<object name>", "<object description>")

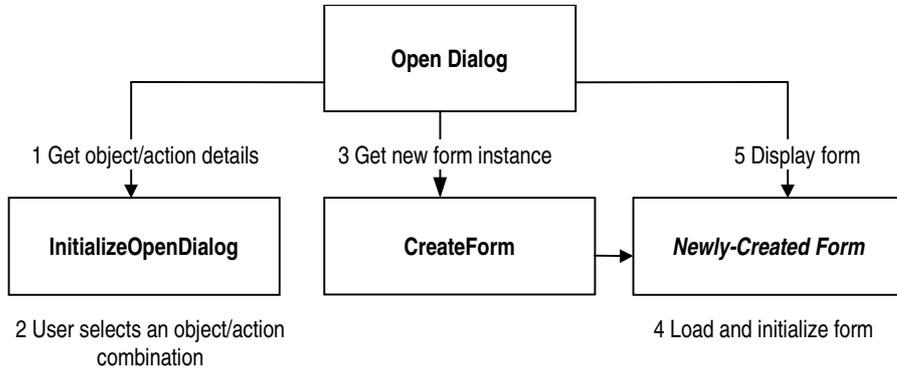
        ' Add the services supported by this business package.
        ' obj.Add "<action1 name>", "<action1 description>", "<action1 ID>"
        ' obj.Add "<action2 name>", "<action2 description>", "<action2 ID>"
    ' ----- <<

End Sub

```

Default Code in the OFactory.bas

The relationship between the Open dialog, the procedures in the OFactory.bas file, and the newly-created form are important to understand as you plan the customization of the object factory. The following diagram clarifies these relationships:



Interaction Between Open Dialog, Procedures in OFactory.bas, and Newly-Created Form

Example of the default OFactory.bas client framework component

```

Option Explicit

'=====
' P U B L I C   Module Variables
'=====

Public gOpenObjects As OpenObjects

'=====
' P U B L I C   Procedures
'=====

Public Sub InitializeOpenDialog()

    Dim obj As OpenObject

    Set gOpenObjects = New OpenObjects

    Set obj = gOpenObjects.Add("Customer", "These are our customers.")
    obj.Add "Maintain", "Customer maintenance", "CUSTMAINT"
    obj.Add "Browse", "Display a list of all customers.", "CUSTBROWSE"

    Set obj = gOpenObjects.Add("Order", "These are our orders.")
    obj.Add "Maintain", "Order maintenance", "ORDERMAINT"
    obj.Add "Browse", "Display a list of all orders.", "ORDERBROWSE"

End Sub

Public Function CreateForm(FormID As Variant) As Form
    Dim frm As Form

    Select Case FormID
    Case "CUSTMAINT"
        Set frm = New frmCustomerMaint

    Case "CUSTBROWSE"
        Set frm = New frmCustomerBrowse

    Case "ORDERMAINT"
        Set frm = New frmOrderMaint

    Case "ORDERBROWSE"
        Set frm = New frmOrderBrowse
    
```

```
' Add additional form variants here.
'Case ...
Case Else
    ASSERT False, "The Object Factory was passed an " & _
                "unknown form ID: " & FormID
    Exit Function
End Select
Set CreateForm = frm

End Function
```

Making your Application Aware of New Business Objects

When you add new business objects to your application, such as maintenance forms or browse objects, you must update the object factory to make the application aware of these new objects. You must either add code manually to the standard object factory module or generate a new object factory using the super model.

For more information about creating the object factory using the super model, see **Using the Super Model to Generate Applications**, page 93.

If you choose to update the object factory manually, you will have to update each of the associated object factory procedures. The steps outlined below describe how to update these procedures.

The steps required to link your object factory module with the client framework are summarized in the following table:

Steps	Description
Step 1	Update the InitializeOpenDialog procedure.
Step 2	Update the CreateForm procedure.
Step 3	Update the GetBrowser procedure.
Step 4	Update the BrowserExists procedure.

Each of these steps are described in the following sections.

Step 1 — Update the InitializeOpenDialog Procedure

The purpose of this procedure is to create a list of all the Visual Basic business objects known to the application. This list is implemented as a Visual Basic collection of OpenObjects types, and the objects contained in this collection are of OpenObject types. Both of these class definitions are supplied with the Construct Spectrum client framework. You can use the Object Browser in Visual Basic to view the public methods and properties of these objects.

For more information, see the *Construct Spectrum Reference Manual*.

Example of the InitializeOpenDialog procedure

```
Public Sub InitializeOpenDialog()

    Dim obj As OpenObject

    ' Create a new global instance of the OpenObjects collection.
    Set gOpenObjects = New OpenObjects

    ' Add the Customer business object and its actions.
    Set obj = gOpenObjects.Add("Customer", "Customer")
    obj.Add "Maintenance", "Customer Maintenance", "Customer_M1"
    obj.Add "Browse", "Browse Clients", "Client_B1"

    ' Add the Order business object and its actions.
    Set obj = gOpenObjects.Add("Order", "Order")
    obj.Add "Maintenance", "Order Maintenance", "Order_M1"
    obj.Add "Browse", "Browse Orders", "Order_B1"

    ' To add a new business object copy this code block and
    ' uncomment and modify lines as required ----->>

    'Set obj = gOpenObjects.Add("<object name>", "<description>")
    'obj.Add "<action1 name>", "<description>", "<action1 ID>"
    'obj.Add "<action2 name>", "<description>", "<action2 ID>"
    ' <<----->>

End Sub
```

In the above example, there are two Visual Basic business objects known to the application — Customer and Order.

- To add a new object to the application:
- 1 Copy the commented lines delimited by the arrows (shown in bold above).
 - 2 Uncomment the line to add a new business object to the `OpenObjects` collection. Change the object name and description to pertain to your Visual Basic business object.
 - 3 For each action supported by your Visual Basic business object (such as Maintenance, Browse, or Reports), copy and uncomment a line to add the action.
 - Change the action name and action description to pertain to the specific action.
 - Change the form ID to uniquely identify the action within the entire application.

Step 2 — Update the `CreateForm` Procedure

This function takes a form ID as a parameter and returns a reference to a Construct Spectrum form that implements the requested business action.

You must add a new case statement for each form ID that you have added to your `InitializeOpenDialog` procedure to handle the creation of the Visual Basic form that implements the action.

Example of the CreateForm procedure

```

Public Function CreateForm(FormID As Variant) As Form

    Dim frm As Form
    Dim BrMgr As BrowseManager

    ' For every possible action supported by the business objects in
    ' the application, instantiate a form to service the action.

    Select Case FormID

        ' Copy this case for each new maintenance form ----->>
        Case "Customer_M1"
            ' Create a new Customer maintenance form.
            Set frm = New frm_Customer
        ' <<-----

        ' Copy this case for each new browse form ----->>
        Case "Customer_B1"
            ' Create a new Browse Manager object for the Customer Browse
            ' Object.
            Set BrMgr = GetBrowser("NCST-CUSTOMER")
            ' Ask the Browse Manager object to create a new Customer
            ' Browse form.
            Set frm = BrMgr.MDIBrowserForm
        ' <<-----

        Case Else
            ASSERT False, "The Object Factory was passed an " & _
                "unknown form ID: " & FormID
            Exit Function
        End Select

    Set CreateForm = frm

End Function

```

- To add support for a new Visual Basic maintenance business object action:
- 1 Copy the commented code block delimited by the arrows for the maintenance action (as shown in bold above).
 - 2 Modify the line to add a case statement for the action. Change the FormID in the Case line to match the ID of the Visual Basic maintenance business object's action.

- 3 Modify the line that creates a maintenance form. Change the name of the form to the name of the form generated by the VB-Maint-Dialog model for the new Visual Basic business object.
- To add support for a new Visual Basic browse business object action:
 - 1 Copy the commented code block delimited by the arrows for the browse action (as shown in bold italics above).
 - 2 Uncomment the line that adds a case statement for this action. Change the FormID in the case statement to match the form ID of the Visual Basic browse business object's action.
 - 3 Uncomment the line that uses the GetBrowser(TableName) function to return a reference to an initialized BrowseManager object. Change the TableName parameter to the name of the database table for which the Visual Basic business object was generated.
 - 4 Uncomment the line that uses the MDIBrowser method of the BrowseManager object to return a reference to an MDI browse form.

Step 3 — Update the GetBrowser Procedure

When you add a new Visual Basic business object that supports a browse action to your application, you must add a new case statement to this function to initialize and return a BrowseManager object.

Use the GetBrowser procedure to return a reference to an initialized BrowseManager object. Client framework components use this function to request a reference to a BrowseManager object used to request browse services (such as displaying a MDI browse or modal browse form or performing a lookup request).

For more information about the BrowseManager, see **Customizing the Generic Browse Dialog**, page 232.

Example of the GetBrowser() function

```

Public Function GetBrowser(TableName As String) As Browser
Dim BrMgr As New BrowseManager
    ' Return a browser object for the requested table.
    Select Case TableName

        ' Copy this code block to add support for a new Browse ----- >>
        Case "NCST-CUSTOMER"

            ' Create a New Customer Browse Object.
            Dim CustomerBrowse As New CustomerBrowse

            ' Set the BrowseManagers base object to the Customer
            ' Browse Object's BaseObject.
            Set BrMgr.BrowseObject = CustomerBrowse.BaseObject

            ' Assign the Caption property of the BrowseManager.
            BrMgr.Caption = "Query Customers"
        ' ----- <<
        Case "NCST-ORDER-HEADER"
            Set BrMgr.BrowseObject = New OrderBrowse
            BrMgr.Caption = "Query Orders"
        End Select
    Set GetBrowser = BrMgr
End Function

```

- To add support for a new Visual Basic browse business object action:
- 1 Copy the commented code block delimited by the arrows (as shown in bold above).
 - 2 Modify the line that adds the new case statement. Change the name of the table to the name of the database table implemented by the new Visual Basic browse business object.
 - 3 Modify the line that creates the new specific browse object. Change the instance name and class name of the specific browse object to the name of the class that was generated by the VB-Browse-Object model for the new business object. This is the class that initializes a generic base browse object, with for example the column names, formats, captions, and key names specific to a particular Visual Basic browse business object.
 - 4 Modify the line that sets the BrowseManager's BaseObject property to the BaseObject property of the specific browse. Change the specific browse object name to the class name of the specific browse object generated by the VB-Browse-Object model for the new Visual Basic browse business object.

- 5 Modify the line that assigns BrowseManager's Caption property. Change the caption to describe the Visual Basic browse business object.
- 6 If your application supports multiple languages at runtime, see **Internationalizing Using the Client Framework**, page 369, for more information about how you can internationalize the caption.

Step 4 — Update the BrowserExists Procedure

When you add a new Visual Basic business object that supports a browse action to your application, you must add a new case statement to the BrowseExists procedure to make the browse known to all the other components in your application.

Other application components never refer directly to a specific Visual Basic browse business object. Instead, they refer to the browse via the tablename for which the specific Visual Basic browse business object has been implemented. This allows application components that use the services of browse objects to compile and execute even if the browse objects have not yet been added to your project.

Example of the BrowserExists procedure

```
Public Function BrowserExists(Table_name As String) As Boolean

    ' Optimistic
    BrowserExists = True

    ' Check if there is a browse object for the requested table name.
    Select Case Table_name

        ' Copy this line to add support for a new browse ---- >>
        Case "NCST-CUSTOMER"
        ' ----- <<
        Case "NCST-ORDER-HEADER"
        Case Else
            BrowserExists = False
    End Select

End Function
```

Note: The table names used in this function must match those in the GetBrowser function. Table names must be the view names documented in Predict.

- To add support for a new browse:
- 1 Copy the code block delimited by the arrows (as shown in bold above).
 - 2 Modify the line that adds a new case statement. Change the database table name to the name of the table implemented by the Visual Basic business object. Make sure that this is the same table name that is referred to in the GetBrowser() function for this Visual basic browse business object.

Spectrum Dispatch Client Support

The Spectrum Dispatch Client (SDC) client framework components provide functionality that integrates the rest of the client framework and the generated code with the Spectrum Dispatch Client. Consider these client framework components to be helper components that simplify using the Spectrum Dispatch Client.

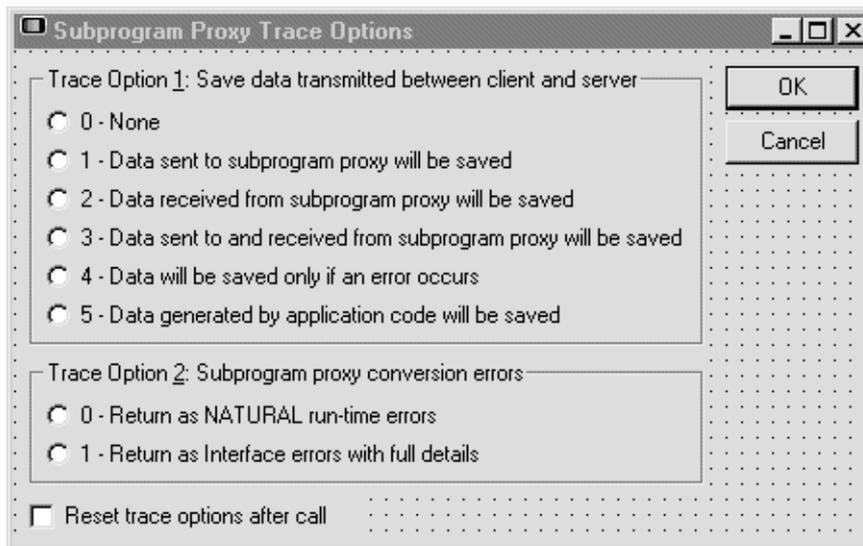
The Spectrum Dispatch Client uses one generic dialog to display varying information based on need. A Construct Spectrum application uses the dialog in three distinct ways:

- To prompt the user for a Construct Spectrum user ID and password when a remote CallNat returns a security error
- To display communication error messages to your user
- To prompt the user to specify a dispatch service for the application

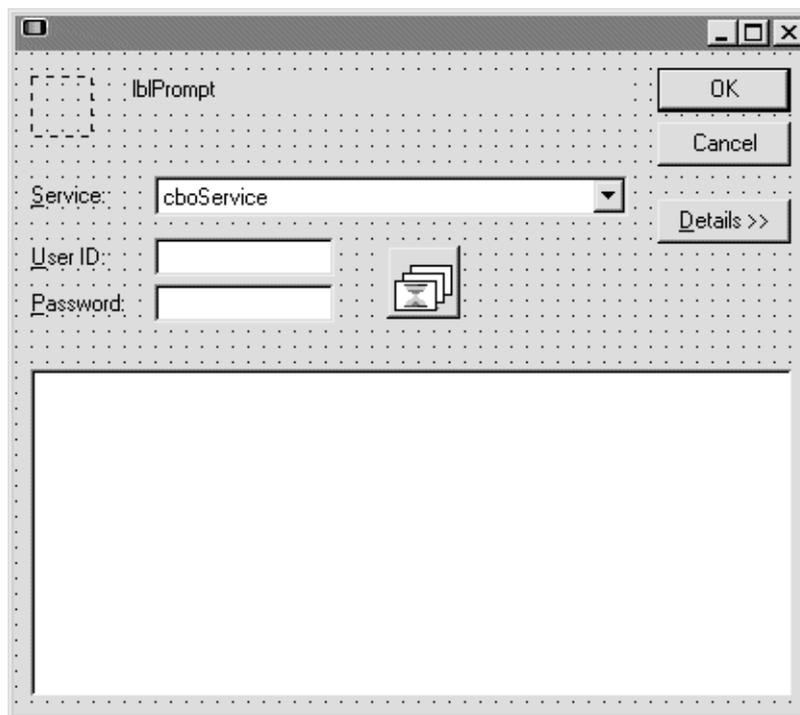
Each of these components is described in the following sections.

The following table describes the Spectrum Dispatch Client dialog client framework components supplied with Construct Spectrum:

Component	Description
SDCDialog.frm	Prompts the user for logon credentials, selects dispatch services, and displays errors arising in the Spectrum Dispatch Client.
TraceOptions.frm	Sets trace options for a remote call. For more information, see Debugging Your Client/Server Application , page 201, in <i>Construct Spectrum Programmer's Guide</i> .
SDCSupport.bas	Encapsulates common Spectrum Dispatch Client procedures.



TraceOptions.frm Supplied With Your Construct Spectrum Application



SDCDialog.frm supplied with Construct Spectrum Client Framework

The client framework uses the SDCDialog.frm to supply all three of these features.

Logon Dialog

The Logon dialog provides a convenient way of obtaining a user ID and password from the current user. The user ID and password are required for all calls to back-end Natural services to ensure that the user is authorized to access each service.

By default, the Logon dialog displays when the application starts and whenever a “No Permission to Execute Function” error occurs.

Error Messages

Error messages returned by the Spectrum Dispatch Client are displayed by the client framework using the SDCDialog form. For more information about messages, see *Construct Spectrum Messages*.

Dispatcher Selection Dialog

The client framework displays the Dispatcher Selection dialog to allow users to select which dispatcher to associate with their current application.

For more information about the Spectrum Dispatch Client, **Spectrum Dispatch Client Components**, page 249, in *Construct Spectrum Programmer's Guide*.

Utility Procedures

The utility procedures in the client framework are functions and subroutines accessed by many other components of the client framework. For example, client framework components access the utility procedures to center a form on the screen, parse strings, calculate minimum and maximum values, test assertions, and set the mouse pointer appearance.

The following table lists the client framework component containing the utility procedures:

Component	Description
CSTUtils.bas	Contains a collection of utility procedures and global constants.

The following table provides a brief description of each utility procedure:

Utility	Description
AppendSlash	Appends a backslash to the end of a directory name, if necessary.
ArrayDimensions	Returns the number of dimensions in an array.
ASSERT	Tests an assertion.
CenterForm	Centers a form relative to the screen or to another form.
CreateArray	Creates and returns a one-, two-, or three-dimensional array of variants.
CreateStringArray	Creates and returns a one-, two-, or three-dimensional array of variants, but creates an array of strings.
CSTFormatMessage	Formats a message in a CDPDA-M or CSASTD data area by performing the substitutions.

Utility	Description (continued)
CSTSelectContents	Highlights the contents of a TextBox control by setting the SelStart and SelLength properties. This procedure can be called in the GotFocus event for the TextBox to simulate Windows behavior of selecting text when you Tab to a field.
CSTSubst	Substitutes values into a string marked with the Construct :n: substitution place holders.
FileExists	Tests if a file exists by attempting to open the file.
FindFirst	Searches a string for the first occurrence of a character in a set of characters.
FixupRTF	Changes any embedded backslash characters in a string to two backslashes so that the string can be displayed properly in a RichTextBox control.
GetPrivateProfileStringVB	Reads a string value in a Windows .INI file. This procedure is a Visual Basic wrapper around the Windows GetPrivateProfileString function.
GetWindowsDirectoryVB	Returns the name of the Windows directory. This procedure is a wrapper around the Windows GetWindowsDirectory function.
IsForegroundApplication	Returns True if the application is currently the foreground application and False if not. Use this function to execute code only if the application is currently active.
IsMDIChild	Returns whether or not a form is an MDI child window.
Max	Returns the maximum of two values.
Min	Returns the minimum of two values.
MoveFormSafely	Moves a non-MDI child form to a new location on the screen, ensuring that the entire form is displayed.
PadLeft	Pads a string on the left with spaces or any character to a specified width.

Utility	Description (continued)
PadRight	Pads a string on the right with spaces or any character to a specified width.
ResizeForm	Resizes a form so that its client area is the specified size. If you know how big the client area needs to be, call this procedure to resize the form.
SetUppercaseStyle	Sets the Windows style bit for a TextBox control so that the control converts all text to upper case.

For more information about the utility procedures, see **Utility Subroutines on the Client**, page 477, in *Construct Spectrum Reference Manual*.

VALIDATING YOUR DATA

This chapter outlines the data validation facilities provided with Construct Spectrum.

The following topics are covered:

- **Overview**, page 328
- **Client Validation**, page 331
- **Creating Verification Rules in Predict**, page 336
- **Order of Precedence in Data Validation**, page 339
- **Validation Error Handling**, page 340

Overview

Construct Spectrum-generated applications provide a framework for data validation designed to ensure the integrity of your information. Construct Spectrum applies four levels of data validation. Before adding or changing any data, Construct Spectrum applies basic data type checking, business data type checking, local business validation, and business object validation.

Errors arising from any of these data validation levels are displayed on the client.

Basic Data Type Validation

The Spectrum Dispatch Client performs basic data type validation. It uses the format and length associated with each field in your object PDA to ensure that the value being assigned to a field will not result in a type mismatch, an overflow condition, or an underflow condition.

Business Data Type Validation

The second level of validation is business data type (BDT) validation. BDTs allow data to be displayed in a format that is based on business language conventions rather than on programming language conventions. For example, a variable with a Visual Basic data type of Double will display as a phone number if it is assigned the BDT named BDT_PHONE.

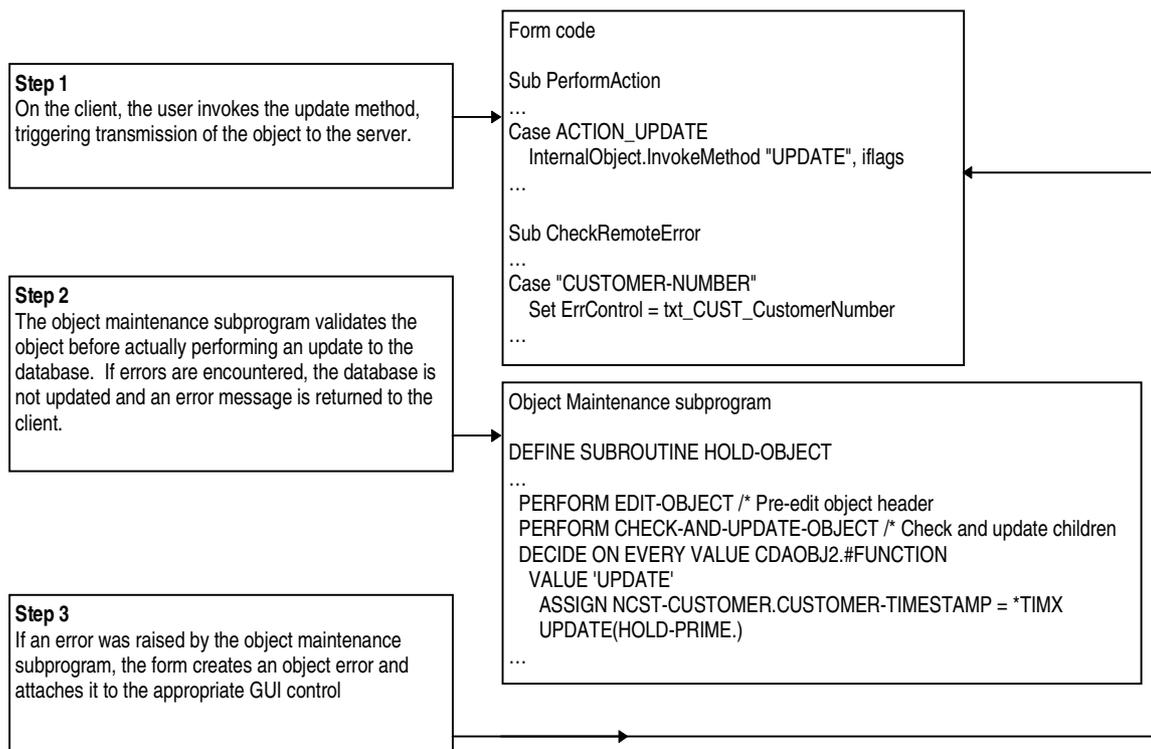
BDT validation ensures that the user input conforms to the Visual Basic data type and to the business semantics attached to the BDT. In the example above, BDT validation checks that the user input makes sense as a phone number.

Local Business Validation

Local business validation applies simple business rules to data. This level of validation is coded within the Visual Basic maintenance object and is performed on the client. Typical local business validations include range checking, domain checking, and calculating required values. Database access is not recommended within local business validations.

Business Object Validation

Business Object Validation is performed in the maintenance object subprogram on the server. The maintenance object subprogram is responsible for ensuring that the data entered by a user is correct before it is committed to the database. Any local business validation should also be coded in the maintenance object subprogram. Coding on both client and server is crucial if client applications written for another environment (for example, a character-based interface) share the same maintenance object subprogram for data access.



Typical Client Validation Cycle

You can write custom validation code directly in user exits of the maintenance object subprogram or you can attach Predict verifications that the Object-Maint-Subp model will include in the generated module. For more details on entering Predict verifications, see **Creating Verification Rules in Predict**, page 336.

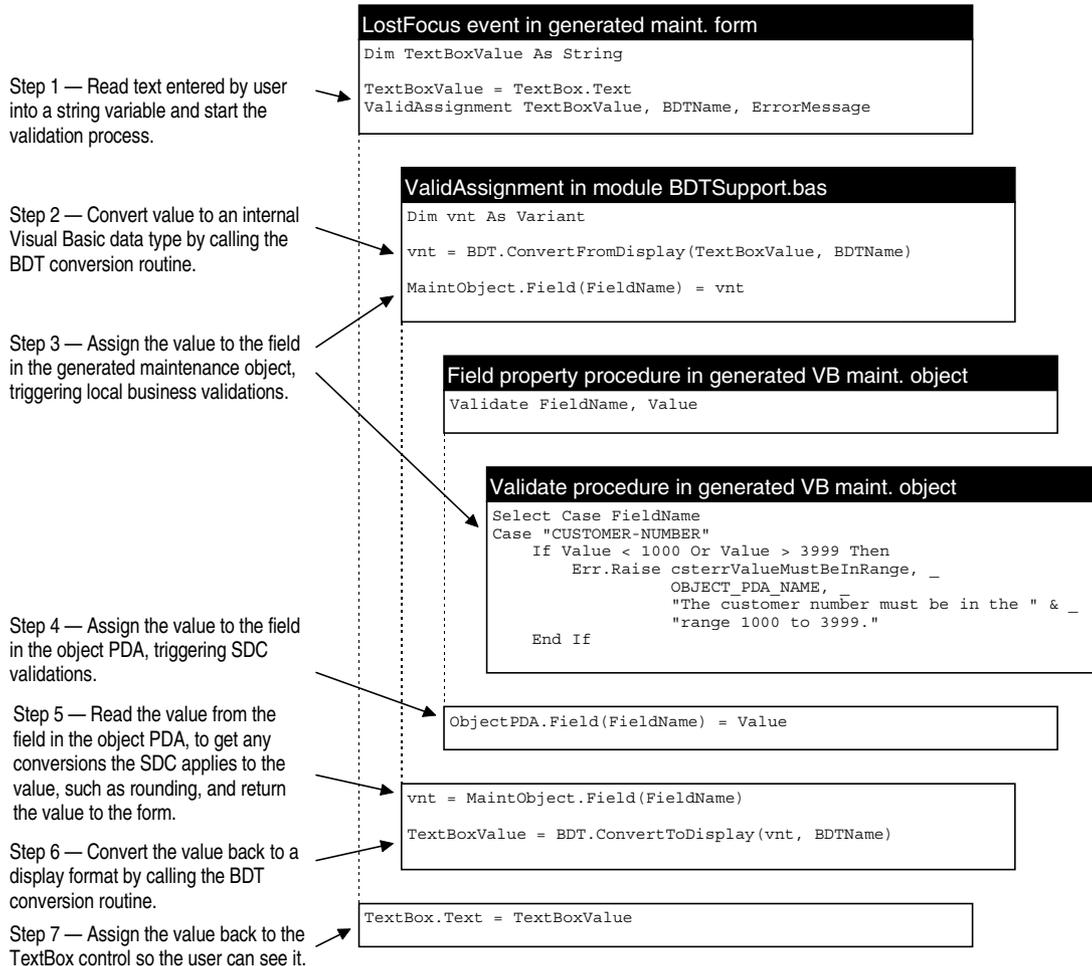
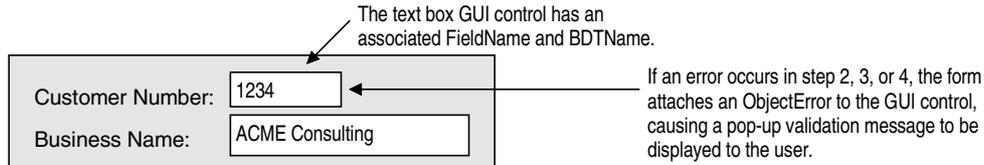
Tip: If you have both GUI and character dialogs, both can access database information using the maintenance object subprogram. Ensure that any client validations are replicated in the maintenance object subprogram.

Client Validation

Data assignment from the form to the client's copy of the object PDA triggers three types of client validation: basic data type validation, BDT validation, and local business validation. It is the attempt to update the object PDA that triggers the validations. The form keeps the client's object PDA up to date by attempting to update its data when:

- a LostFocus event occurs on a TextBox
- a Click event occurs on a CheckBox, ComboBox, or OptionButton
- an AfterColumnEdit event occurs on a grid column

The following example illustrates the data validation logic initiated when one of these events is triggered:



Triggering Validation in the Form

Validation in Maintenance Dialogs

All validation is triggered from the form. Form code is responsible for linking BDT validations to specific GUI controls and for responding to validation errors.

Using BDTs

The VB-Maint-Dialog model generates default BDT assignments for each GUI control on your form. You can override these assignments by attaching your own BDT keywords to Predict field definitions. For details on linking BDTs to GUI controls within Predict, see **Customizing on the Server**, page 159.

You can override BDT assignments directly in the generated form. However, this method is not recommended. Overriding BDTs within the form is a customization that will be lost when you replace the existing form with a newly generated version.

If there are no BDTs that provide the business semantics your application needs, you can create a custom BDT. For details on creating custom BDTs, see **Using Business Data Types**, page 153, in *Construct Spectrum Programmer's Guide*.

Hand-Coded Validations in Generated Dialogs

If you have specialized validations that must be executed immediately in response to an event, write the code in a maintenance dialog to perform the validations.

If you write hand-coded validations, you can still take advantage of the form's standard error handling technique. This technique is described in **Validation Error Handling**, page 340.

Note: Hand-coding validations is not recommended under most circumstances. These customizations will be lost if you replace the existing form with a newly generated version. To keep your validations after regeneration, write validation code in the user exit.

The maintenance dialog invokes a Validate method in the Visual Basic maintenance object every time a GUI control attempts to update a value in the client's copy of the object PDA. Writing validation code in the Validate method rather than directly in the form should meet most of your validation requirements. The dialog also contains standard code which checks for validation errors raised in the Visual Basic maintenance object.

Validation in Visual Basic Maintenance Objects

You can code local business validations in Visual Basic maintenance objects. Each time the maintenance dialog attempts to update a value in the Visual Basic maintenance object, it invokes a standard validation subroutine (Validate) in the Visual Basic maintenance object. You can hand-code validations in the CLIENT-VALIDATIONS user exit of the Validate subroutine, or you can use Predict verification rules to validate data.

Regardless of how it gets into the Validate subroutine, there are two basic components to the validation:

- a case statement indicating the field requiring validation. This statement includes the test for a particular condition.
- code which raises an error if the field value fails the validation

Adding Validations in the CLIENT-VALIDATION User Exit

Use the CLIENT-VALIDATIONS user exit located in the Validate subroutine of a VB-Maint-Object model to write custom validations. Although this custom code can be entered into the user exit on the server, you can also use Visual Basic's GUI editing environment to supply your code. The following illustration shows a typical entry in the CLIENT-VALIDATIONS user exit:

Example of validation code in CLIENT-VALIDATION user exit

```
'SAG DEFINE EXIT CLIENT-VALIDATIONS
  Case "CUSTOMER-NUMBER"
    If Value = 1010 And _
      m_ObjectData.Field("CREDIT-LIMIT") > 1000 Then
      Err.raise Number:=csterrCustomerOnProbation, _
        Description:= "Credit limit too high, on probation", _
        Source:=OBJECT_PDA_NAME
    End If
'SAG END-EXIT
  End Select
...
```

In this example, the value for the field to be updated in the client's object PDA is stored in the `Value` variable. If you require the values from other fields in the object PDA for your validation, use the Spectrum Dispatch Client's `Field` or `GetField` methods as illustrated in the previous code example.

Warning:

If your validations require remote database access, it is strongly recommended that you do not code these validations in the Visual Basic maintenance object. A Construct Spectrum application operates in a synchronous manner, which means the user must wait for validations in the Visual Basic maintenance object to complete execution before control returns to the dialog for further interaction.

Validations from Predict

Generated validations that are based on Predict verification rules are checked immediately after your hand-written validations in the CLIENT-VALIDATIONS user exit. These generated validations use the same structure that is shown in the hand-written code example earlier in this chapter. For more information on creating verification rules in Predict, see **Creating Verification Rules in Predict**, page 336.

Creating Verification Rules in Predict

Verification rules that you create in Predict to use with applications generated by Construct Spectrum follow the same guidelines that traditional Natural Construct applications use. For example, all verification rules intended for use during generation must be of type N.

Note: To set verification rules to type N in Predict, use the GEN CST command in the Predict rule editor.

For a complete discussion on using verification rules with traditional Natural Construct applications, see **Use of Predict in Natural Construct**, page 941, in *Natural Construct Generation User's Manual*.

Construct Spectrum uses verification rules to generate GUI control definitions as well as to generate business validations that might be implemented in either the maintenance object (in Visual Basic), the object maintenance subprogram (in Natural), or in both. The validations are duplicated to provide immediate feedback on the client and to have a centralized implementation of validations on the server.

When creating Predict verification rules for applications using Construct Spectrum, take advantage of new syntax that makes your verification rules easier to reuse and easier to define in Predict.

Deciding Where To Implement a Validation Rule

Conventionally, validation rules are kept together in a single module. However, since sending the client's object data to the server for validation takes time, validating a rule on the client can save transmission time.

You can implement a validation rule in the object maintenance subprogram only, or you can implement it both in the object maintenance subprogram and in the Visual Basic maintenance object. To decide on which of these two options to choose, determine what types of information a rule requires to do its validation. Use the following guidelines to help you decide:

- If the rule needs to look up data on a foreign file, implement the rule in the object maintenance subprogram for ready access to the foreign file.
- If the rule performs calculations on data within the object's data, it may be more efficient to perform this validation in the Visual Basic maintenance object.

Include the rules placed in the Visual Basic maintenance object in the object maintenance subprogram for use by character interface applications.

Coding User Type Rules

Construct Spectrum introduces a new syntax convention for coding type U (User) rules. This convention allows a single rule to contain a Visual Basic implementation or a Natural and Visual Basic implementation.

Rules defined in Visual Basic are delimited by code blocks. Use the following syntax in the Predict rule editor to create a code block for a Visual Basic rule:

Example of code block for a Visual Basic rule

```
>>BEGIN RULE VB  
Visual Basic implementation of the VE rule here.  
>>END-RULE
```

Any rule code that is not delimited within a language-specific code block will be assumed to be a rule coded in Natural, since Natural rules do not require code block delimiters. To keep code looking consistent, Natural rules can also be delimited.

Example of code block for a Natural rule

```
>>BEGIN RULE NATURAL  
Natural implementation of the VE rule here.  
>>END-RULE
```

A rule can consist of several code blocks for both Visual Basic and Natural.

Example of code blocks for using both Visual Basic and Natural

```
>>BEGIN RULE VB
1st part of Visual Basic implementation of the VE rule.
>>END-RULE
```

```
>>BEGIN RULE NATURAL
1st part of Natural implementation of the VE rule.
>>END-RULE
```

```
>>BEGIN RULE VB
2nd part of Visual Basic implementation of the VE rule.
>>END-RULE
```

```
** By default, this code is Natural code because it is
** not delimited by a language-specific code block.
3rd part of Natural implementation of the VE rule.
```

When combining Visual Basic and Natural rules, you cannot use nested language-specific code blocks. For example:

Use This

```
>>BEGIN RULE VB
1st part of Visual Basic rule...
>>END-RULE

>>BEGIN RULE NATURAL
1st part of Natural rule...
>>END-RULE
```

NOT This

```
>>BEGIN RULE NATURAL VE rule...
>>BEGIN RULE VB
This VB code block is invalid
>>END-RULE
>>END-RULE
```

Order of Precedence in Data Validation

Data validation is triggered under two conditions: attempted assignment to the client's copy of the object PDA and attempted database update using the Update or Add method of the maintenance object subprogram.

Each of these conditions triggers different layers of the Construct Spectrum data validation model:

- data assignment to the client's object PDA.
In this stream of data validation, the order of validation is executed as follows:
 - BDT validation
 - local business validation
 - basic data type validation
- database update using the maintenance object subprogram
In this stream of data validation, only Business Object Validation is executed.

For clarification, see the illustrations in **Business Object Validation**, page 329.

Validation Error Handling

Client validation is always initiated with a call to the generic `ValidAssignment` subroutine. This call occurs in an event code block (usually a lost focus event) that assigns a GUI control's value to the client's object PDA. There are a number of steps to follow for each assignment.

- To assign a GUI Control's value to the client's object PDA:
 - 1 Hide any error tips that may be attached to this GUI control. This is accomplished by calling the `HideErrorTip` subroutine in `CSTUTILS`.
 - 2 Remove any Error Objects from the GUI control. This is accomplished by calling the subroutine `RemoveUnneededControlErrors`.
 - 3 Initiate local data validation and assign the value to the client's object PDA. This is accomplished by calling the `ValidAssignment` subroutine.
 - 4 Test to see if any validation errors occurred during the assignment attempt. This is accomplished by checking whether `ErrorMsg` contains a value.
 - If errors occurred, attach an Object Error to the GUI control by calling the `ParseErrorString` and `SetObjectError` subroutines.

Framework Components

The validation error handling framework components are used to implement the mechanism that displays pop-up validation errors on browse and maintenance dialogs.

For example, when the user enters data into a field and cursors to the next field, the data is checked to ensure it is valid. If the data is not valid because it violates a business rule or cannot be interpreted properly (such as when non-numeric data is entered into a numeric field), the field that contains the error is highlighted with an error color and a pop-up message is displayed next to the field. The user is not locked into the field until the error is corrected and can continue entering or editing data in other fields. At any point, the user can return to the highlighted field or fields and correct the errors.

The following table describes the validation error handling components in the Construct Spectrum client framework:

Component	Description
ErrorPreferences.frm	Allows users to customize how validation errors are presented.
ErrorTip.frm	Displays the pop-up validation error message.
ObjectError	Keeps track of the information for a single validation error on a form.
ObjectErrors	Tracks the validation errors on a generated maintenance form; each generated maintenance form declares one instance of this class.

Handling Business Object Validation Errors

Business Object Validation errors are returned to the form in the message PDA, CDPDA-M. If an error was returned from the server, the CheckRemoteError subroutine in the form tests the value of the ERROR-FIELD variable to match it up with a GUI control.

If the field is associated with a GUI control, an Object Error is attached to the GUI control. Otherwise the form displays a message box showing the description of the general error.

The following code illustrates this process:

```
Select Case InternalObject.Msg.Field("ERROR-FIELD")
Case "BUSINESS-NAME"
    Set ErrControl = txt_CUST_BusinessName
Case "PHONE-NUMBER"
    Set ErrControl = txt_CUST_PhoneNumber
...
End Select
If ErrControl Is Nothing Then
    MsgBox cstFormatMessage(InternalObject.Msg), vbInformation
Else
    With InternalObject.Msg
        SetObjectError Me, ErrControl, .Field("MSG-NR"), ErrMsg, _
            ERROR_SOURCE_SERVER, ErrColumn, _
            .Field("ERROR-FIELD-INDEX1"), _
            .Field("ERROR-FIELD-INDEX2"), _
            .Field("ERROR-FIELD-INDEX3")
    End With
End If
```

UNDERSTANDING THE BROWSE AND MAINTENANCE INTEGRATION

This chapter explains how browse and maintenance functions are integrated. It includes information about linking and using browses from a maintenance dialog.

The following topics are covered:

- **Overview**, page 344
- **Design Objectives**, page 349
- **Overview of Foreign Key Field Relationships**, page 351
- **Foreign Field Support Provided By Maintenance Dialogs**, page 355

Overview

Providing applications with tightly integrated browse and maintenance functions makes it easier for users to navigate through an application and to find the information they need. The two main benefits that integrated browse and maintenance functions provide are:

- Drill-down capabilities from a browse dialog. For example, to invoke a maintenance dialog or another browse from within a browse dialog.
- Active help from maintenance dialogs to aid in selection of primary and foreign fields.

These topics are discussed in the following sections.

Drill-Down Capabilities from a Browse Dialog

Users commonly use browse dialogs to navigate within an application. For example, a user might select a customer from a Customer browse dialog, drill-down to another browse dialog to see outstanding orders for the customer, select an order, and drill-down to a maintenance dialog to update the order.

You can support this functionality with Construct Spectrum by hand-coding a browse command handler to define the commands supported by a particular browse dialog. You must also add code to the target of these commands, which are typically other application components such as a maintenance dialog or a Visual Basic maintenance object.

For information about creating Browse Command Handlers, see **Understanding Browse Command Handlers**, page 238.

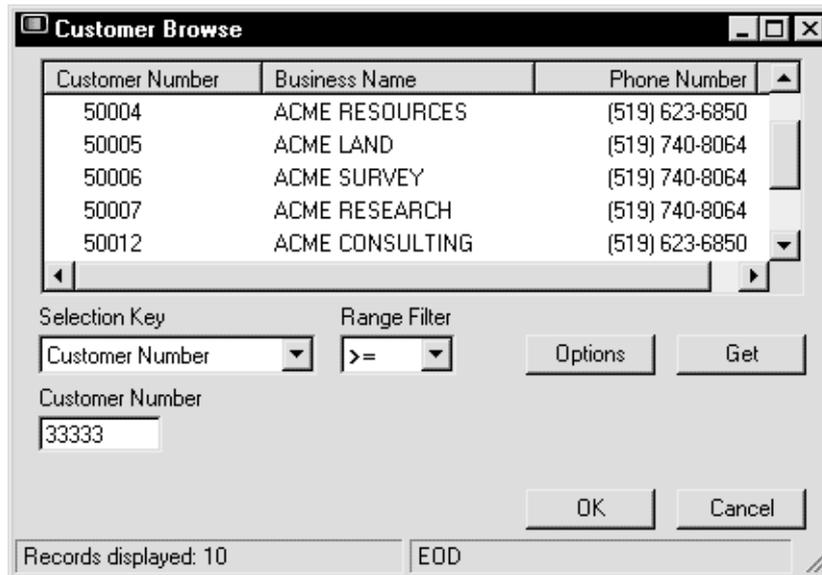
Tip: To see examples of browse command handler source code, review the `CustomerBrowseCommands.cls` and `OrderAsBrowseTarget.cls` files in your Construct Spectrum Order Entry demo project.

Active Help on Maintenance Dialogs

Users can select valid values from dialog fields that are enabled with active help. Construct Spectrum maintenance dialogs provide built-in support for two types of active help: primary key field and foreign key field active help.

Primary Key Field Active Help

Primary key field active help is available for all business objects for which maintenance and browse dialogs were generated. When a maintenance dialog is opened, it verifies whether a browse was generated for its primary key field. If one was, it enables the browse toolbar button and browse menu command on the MDI frame. When a user clicks the browse toolbar button or selects the browse menu command, a modal browse dialog for the business object is displayed:



Modal Browse Dialog

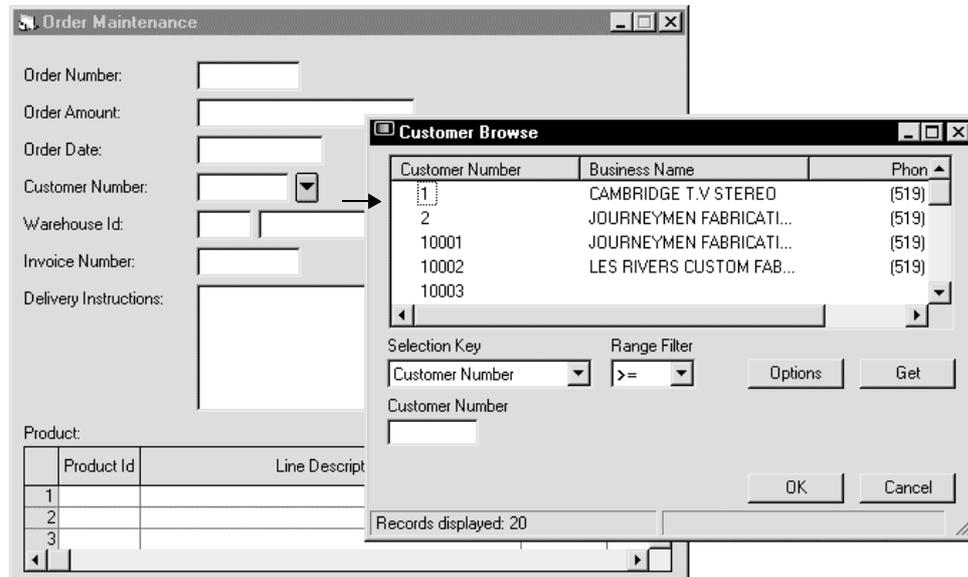
The browse dialog displays a list of existing records in the database. Users can select to maintain a record by double-clicking the record or by highlighting a row and clicking OK.

Foreign Field Active Help

Most maintenance dialogs are linked by foreign relationships. These relationships, also known as inter-object relationships, link a field in a dialog to the primary field of another dialog. In the demo application, for example, the Order dialog has a Customer Number field. To be valid, the Customer Number must exist on the Customer database table. This rule is defined by an inter-object relationship that specifies the two tables involved (Order and Customer), the linked fields, the cardinality, and other optional information.

Maintenance dialogs automatically support active help for foreign fields in the following ways:

- By providing a button beside the text box.
When a user clicks the button, a dialog is displayed to select foreign values:



Active Help From a Foreign Field

Descriptive information can also be returned with the selected value. For example, the customer's name could be returned with the customer number.

For more information about returning descriptions with foreign fields, see **Displaying Descriptions for a Foreign Field**, page 359.

- By automatically refreshing a foreign field description when a user types a value directly into a foreign field.
When the `LostFocus` event occurs in the field, the foreign field is looked up and the description is updated on the maintenance dialog.
- By retrieving all of the values and descriptions for a foreign field that are in the database.
This method is used by the maintenance dialog to create a drop-down list of all the allowed values for a foreign field. This feature is used only if the foreign file contains a small set of stable records.

Warehouse Id:	111	TORONTO CENTRAL WAREHOUSE
Invoice Number:	231201	DON'S GOLD DEPOT
Delivery Instructions:	Must be	FRESCO FREEZER LTD
Product:		WATERLOO WAREHOUSING LTD.
		KITCHENER-WATERLOO WAREHOUSE
		BRANT WAREHOUSING AND STORAGE
		///// CENTRAL WAREHOUSE
		A.B. SMITH LIMITED
		CANADA FOOD INC.

Product	Quantity	Unit Cost	Total
---------	----------	-----------	-------

Active Help From a Drop-down List

Design Objectives

Construct Spectrum meets two design objectives that simplify the integration of maintenance and browse components:

- Application component independence
- Simplified generated components

These objectives are discussed in the following sections.

Application Component Independence

An important design objective when integrating discrete application objects like maintenance and browse dialogs is to limit the impact this has on existing application objects. To achieve this, there must be a minimal amount of coupling between application components. Less coupling means that changes to one application component are less likely to affect the other.

To achieve minimal coupling, Construct Spectrum uses the object factory as the single integration point for all new application components. Only the object factory needs to be aware of new application objects. As new business objects are added to your application, they are published as available for use by other business objects through the object factory interface.

For more information about the object factory, see **Object Factory**, page 302.

Tip: To view the source code for the demo application's object factory, open the OFACTORY.bas file in the Construct Spectrum demo project.

Maintenance dialogs request browsing services through the object factory interface. Using parameters such as table names or relationship names, the maintenance dialog specifies which file is required for the browse. If the file is not available, the object factory informs the requesting maintenance dialog, allowing it to disable that functionality. This architecture allows an application to be developed incrementally so that you can test it throughout the development cycle.

To view an example of how this code works, see the code for the EnableForeignKeys subroutine in the CUSTMCDV.frm maintenance dialog form in the demo project.

Simplified Generated Components

Another objective is to reduce the complexity of generated components, making them easier to customize. The amount of code required to integrate maintenance and browse processes is greatly reduced by using the `BrowseManager` framework class. It encapsulates most of the common functionality involved in using browse processes.

To see how the `BrowseManager` has been implemented, review the code in the `BrowseManager.cls` client framework class. For more information about the `BrowseManager`, see **Understanding Browse Command Handlers**, page 238.

Overview of Foreign Key Field Relationships

A foreign key field with an update constraint is a field on a maintenance dialog that must be set to a value that already exists in a foreign file. This field is the foreign file's primary key field.

A foreign key field relationship links two independent files such as an Order and Customer file. This is also called an inter-object relationship. Conversely, intra-object relationships define relationships within a file, for example, a relationship between two fields in a Customer file.

Foreign key field relationships are business rules that can define both update and delete constraints. However, with respect to integrating maintenance and browse functions, only foreign key field relationships that define update constraints are important.

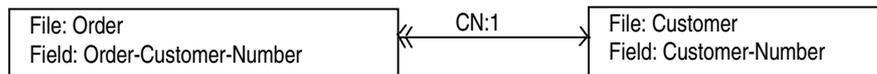
For more information on inter-object and intra-object relationships see **Design Methodology**, page 231, in *Natural Construct Generation User's Manual*.

Fields that can be Used in a Foreign Key Relationship

This section describes the foreign field relationships supported by the Object-Maint-Subp model. Relationships supported by Construct Spectrum are also noted.

Simple Field

This is the simplest type of foreign field relationship in which the format and length of the fields on both sides of the relationships are equal and the fields are not repeating. Simple field relationships are supported by Construct Spectrum.

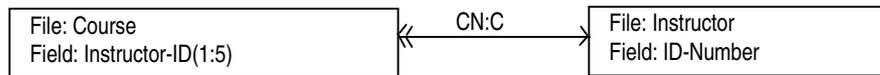


Simple Field Relationship

The relationship shown in the previous diagram is between an order and a customer file. The update constraint is placed on the order. The business rule says each order must have exactly one customer number to be a valid order, and a customer number can be referenced by zero or many orders.

Repeating Field

This is a relationship between a one-dimensional repeating field and either a scalar field or another one-dimensional repeating field. Repeating field relationships are supported by Construct Spectrum.



Repeating Field Relationship

The relationship shown in the previous diagram is between a course and an instructor file. The update constraint is placed on the course. The business rule says a course can have zero to five instructors. An instructor can teach zero or many courses.

Note: The format and length of the relationship fields must be the same on both sides of the relationship.

When Not to Use a Foreign Field Relationship

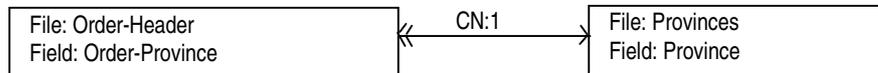
This section describes situations where defining a foreign field relationship is not a good solution. For each situation described, a better alternative is given.

Do not use foreign field relationships to enforce valid values when:

- the list of values is static
- the list of values is small
- there are only two choices

List of Values is Static

In most foreign relationships, both files involved in the relationship are dynamic. It is not a good solution to create a file for the sole purpose of enforcing that valid values are entered from a static list. For example, you would not create a Province file to contain a list of valid provinces that could be entered on an order as shown in the following diagram:



An Unlikely Foreign Field Relationship

A better solution is to attach a table verification rule to the Order-Province field. Construct Spectrum generates a drop-down list for the Order-Province field and populates it with the valid provinces in the verification rule.

There may be valid reasons to create a Province file. For example, to maintain province-specific business rules for calculating sales tax. In this case, a foreign field relationship is appropriate.

List of Values is Small

Another case where you would not use a foreign field relationship is to enforce a small set of values for a field. For example, a Payment-Type field might only have possible values of Cash, Check, MC, Visa, or AMEX.

Again, defining a table verification rule is a more appropriate solution. Using the verification rule, Construct Spectrum would generate option buttons for this field.

List of Values Contains Two Choices Only

If there are only two choices for a given field, do not define a foreign field relationship. Instead, link a verification rule to the field. Construct Spectrum generates either option buttons or a check box for the field.

Foreign Field Support Provided By Maintenance Dialogs

This section describes the foreign field support provided by maintenance dialogs generated with Construct Spectrum.

Two main objectives of linking foreign field lookup support into maintenance dialogs are to:

- Provide a way for users to select valid values for a foreign key field on a maintenance dialog.
When a field value is selected, it must be returned and displayed on the dialog, optionally, with other descriptive fields.
- Provide a way for updating the maintenance dialog with descriptive information associated with the foreign field.
When a foreign field value is entered on a maintenance dialog without using the browse mechanism (for example, by typing directly into a textbox), any values associated with the foreign field, such as a descriptive field, must be updated on the maintenance dialog automatically.

GUI Control Representations of Foreign Fields

This section describes the GUI controls Construct Spectrum uses to represent foreign field relationships on maintenance dialogs. Construct Spectrum deals with foreign fields differently depending on whether the foreign field is located on the primary part of the maintenance dialog or on a secondary, tertiary, or quaternary part of the dialog. Secondary, tertiary, and quaternary information is always represented on a grid control in a maintenance dialog. This section describes how foreign fields are represented in each case.

Foreign Fields On the Primary Part of a Maintenance Dialog

The primary part of a maintenance dialog is any location on the dialog that is not part of a grid. A foreign field on the primary part of a maintenance dialog that has a link to a foreign file can be of any data type.

All foreign fields on the primary part of a maintenance dialog can be represented by a single text box type GUI control. Any GUI Control override keywords that have been specified in Predict to force the type of control that should represent a field are ignored if the field is linked to a foreign file.

For more information about GUI Control Overrides, see **Overriding GUI Controls**, page 160.

Tip: Construct Spectrum does not generate browse support for Boolean fields on a maintenance dialog. Validations for Boolean fields are better handled with verification rules or by adding validation code to the Visual Basic maintenance object.

To provide users with a method to look up valid values for foreign fields from a maintenance dialog, use a button or drop-down list. The following example shows a foreign field using a button:



Foreign Field as Text Box and Lookup Button

When a user clicks the button, a browse dialog listing the foreign field values is displayed. If a descriptive field is associated with the foreign field, a description is also displayed:



Foreign Field as Text Box, Lookup Button, and Description

The following example shows a foreign field with a drop-down list:



Foreign Field as a Drop-Down List

The drop-down list contains a list of the foreign field values or their descriptions. If a descriptive field has been associated with the foreign field, the drop-down list contains the descriptions:

Warehouse Id: 555 DON'S GOLD DEPOT

Foreign Field as a Text Box With Descriptions In Drop-down List

GUI Controls in a Grid

Grids are used on a maintenance dialog to display secondary, tertiary, and quaternary information. Consider an Order business object that is normalized by linking an Order Header file record to 1 to 30 Order Line file records, creating a complex business object. The Order Lines part of this business object is represented by a grid control on the maintenance dialog. A foreign field relationship could be defined between the Order-Line-Product-Id field in the Order Line file and the primary field, Product-Id, in the Product file.

Note: This discussion also applies to foreign field relationships that are linked with repeating fields, since these fields are represented as grid controls.

When a column in a grid represents a foreign field value, a button is placed in the grid to support looking up new values. Either a new value can be typed directly into the grid cell or the button can be clicked to invoke a modal browse dialog:

	Product Id	Line Description	Quantity	Unit
1	187361	CAT NUGGETS	10	
2				
3				

Foreign Field in a Grid with Lookup Button Displayed

Note: Currently, drop-down lists for foreign field values are not supported within grids. Description fields are also not supported for foreign fields within a grid.

How Construct Spectrum Determines Which GUI Control to Use

When generating a GUI control to represent a foreign field on the primary part of a maintenance dialog, Construct Spectrum searches for special properties of the foreign file to determine the type of control to use. Depending on these properties, either a drop-down list or a lookup button is used.

Note: Foreign fields within a grid control are always represented with a lookup button that opens a modal browse dialog when clicked.

A drop-down list is generated for a foreign field if both of the following conditions are met:

- 1 The data dictionary specifies that the *average record count* property of the foreign file contains on average X records, where X is below the threshold determined by the model to be the limit for using a drop-down list. The default value is 50. A value of zero will be ignored.

Tip: You can change the default value of 50 by changing the value of FK-AS-COMBO-THRESH-HOLD in the Natural Construct model defaulting subprogram, CSXDEFLT.

Note: The VB Maintenance Dialog model copies the FK-AS-COMBO-THRESH-HOLD default value into the #PDA-FK-AS-COMBO-THRESH-HOLD variable of the model PDA (CUMDPDA) in the model's pre-generation subprogram (CUMDPR).

- 2 The data dictionary specifies that the *file volatility* property of the foreign file is either *Stable* or *Fixed*.

If both of these conditions are not met, or you have not set these file properties, or you are using a version of Predict that is prior to 3.3.2, a lookup button is generated instead. A lookup button displays a modal browse dialog when it is clicked.

Displaying Descriptions for a Foreign Field

At generation time, the VB-Maint-Dialog model searches for a descriptive field associated with any foreign field. If a descriptive field is found, it is displayed on the dialog with the foreign field. To see how the VB-Maint-Dialog model displays descriptive fields, refer to **Foreign Fields On the Primary Part of a Maintenance Dialog**, page 355.

Note: Construct Spectrum can generate only one descriptive field value for each foreign key value on a maintenance dialog.

You can designate that a field be descriptive whenever it is referenced in a foreign file relationship, or you can designate the field as descriptive only when it is referenced by a particular file. This is useful when different descriptions are needed for different foreign field relationships.

Note: Descriptive fields are not available for foreign fields in a grid.

- To make a field descriptive in all situations:

In Predict, attach the “DESCRIPTION” keyword to the field in the foreign file. All such fields are displayed whenever the file is referenced in a foreign field browse.

- To make a field descriptive only when referenced by a particular file:

In Predict, attach a keyword to the field that matches the name of the file. For example, to make the WAREHOUSE-NAME field descriptive only when a user selects to browse from a dialog that was generated from the ORDER file, link the ORDER keyword to the WAREHOUSE-NAME field in the foreign file.

Note: It may be necessary to define keywords in Predict prior to attaching them to the descriptive fields.

Examples of Descriptive Fields

Suppose your application contains a CUSTOMER file having the following fields:

```
CUSTOMER-ID (N6)  
CUSTOMER-NAME (A20)  
PHONE-NUMBER (N10)  
ADDRESS (A50)
```

Whenever the CUSTOMER-ID field is used in a foreign field browse, you probably want to show the customer name to help identify the customer. To achieve this, link the DESCRIPTION keyword to the CUSTOMER-NAME field. The CUSTOMER-NAME field is now set up as a descriptive field whenever CUSTOMER-ID is used as a foreign field.

Suppose the CUSTOMER-ID field is a foreign field in the ORDER file. When the customer ID is entered for an order, you want to display the address instead of the name. To achieve this, add the ORDER keyword to the ADDRESS field. The ADDRESS field is now descriptive only when referenced by the ORDER file.

Supporting Multiple Descriptive Values and Derived Values

You can retrieve multiple values with a foreign field lookup. For example, you may want to retrieve additional descriptive information or you may need to derive or calculate values in other fields on the maintenance dialog based on values in the foreign file.

Construct Spectrum enables you to do this because each foreign field lookup returns a reference to the BrowseDataCache object containing the row that was selected through the foreign field lookup.

- To retrieve additional values with a foreign field lookup:

Add some code to extract the descriptive value out of the BrowseDataCache object.

Base your code on the sample in the `grd_OrdM_NcstOrderHasLines_ButtonClick` event procedure on the `Ord-Mcdv.frm` maintenance dialog form. This form is located in the Construct Spectrum demo project.

- To derive or calculate values in your maintenance dialog based on the foreign lookup information:

Add code to the AFTER-FOREIGN-KEY-LOOKUP user exit in the VB-Maint-Object to code the updates to your business object.

This ensures that the cached copy of your business object's data which is maintained on the client reflects what is displayed in the maintenance dialog.

Base your code on the sample in the AFTER-FOREIGN-KEY-LOOKUP user exit in Visual Basic maintenance object, Ord-Mcpv.cls, which is located in the Construct Spectrum demo project.

How Foreign Field Descriptions Are Refreshed

Any control on a maintenance dialog affected by a change in value of a foreign field needs to be refreshed when a Get or Clear action occurs. This includes foreign field descriptions as well as any field whose value is derived from a foreign field.

Generated maintenance dialogs include a function called RefreshForeignKeys. This function refreshes the foreign field description when a Get or Clear action occurs. The RefreshForeignField function calls the server and retrieves a description each time a Get or Clear action occurs. This reduces application performance slightly. To avoid this extra call, you can do hand-coding to have the description returned directly from the object subprogram when a Get or Clear action occurs.

- To refresh a foreign field descriptions without an extra call to the server:
 - 1 In the object subprogram, add code to the PARAMETER-DATA user exit to define an extra parameter data area (PDA). Within this PDA, add a parameter for each foreign field that requires a description.
 - 2 In the object subprogram, add code to the EXTENDED-RI-VIEWS user exit to define the views of the foreign file.
To view an example of this code, see the ORD-MSO object subprogram in the SPECDEMO Natural library.
 - 3 In the object subprogram, add code to the AFTER-GET user exit to populate the parameter you added in step 1 with foreign field descriptions after a Get action occurs.
 - 4 In the Visual Basic maintenance object, specify the name of the extra parameter data area that you added in step 1 in the Extra PDA parameter.

To view example code that uses the Extra PDA parameter, see the code supplied in the Visual Basic maintenance object (Ord-Mcpv.cls) in the demo project. In this example, the extra PDA, ORD-XPDA is defined.

- 5 In the maintenance dialog form (.frm file), add code to the RefreshForeignKeys subroutine to extract the description values from the Visual Basic maintenance object when the user selects a Get or Clear action occurs.

The following code example is taken from the Ord-Mcdv.frm Order dialog in the Construct Spectrum demo project. In the following example code, the dialog is updated with the description of the Customer Number foreign field when a Get or Clear action occurs. The name of the Customer Number field is NcstCustomerOrderHeader and the name of the description field is ORDER-BUSINESS-NAME.

Note: The Order dialog has another foreign field, the Warehouse ID field (NcstWarehouseorderheader). Because this field is set up as a drop-down ComboBox, both the warehouse ID and warehouse description values already exist on the client. Therefore, no hand-coding is required to avoid a call to the server for a Get or Clear action.

Example of updating a foreign field description after a Get or Clear action

```
Private Sub RefreshForeignKeys()
    ' RefreshNcstCustomerorderheader
    RefreshNcstWarehouseorderheader

    ' Post generate code ----->>
    ' This code is added to optimize foreign key description
    ' handling.
    With InternalObject

        ' Customer Business Name
        lbl_OrdM_NcstCustomerOrderHeader.Caption = _
            BDT.ConvertToDisplay(.GetField("ORDER-BUSINESS-NAME"), _
                BDT_ALPHA, "A30")

    End With
    ' Post generate code -----<<
End Sub
```

Note: The RefreshNcstCustomerorderheader sub is still used on a lost-focus event for the Customer Number field to lookup a new Customer Number field description.

Supporting Code for Drop-Down lists

This section explains how Construct Spectrum supports a drop-down list for foreign fields on a maintenance dialog. Read this section before hand-coding foreign field drop-down lists.

Initializing a Drop-Down List

Maintenance dialogs that use drop-down lists to support foreign field lookups use instances of a Construct Spectrum framework class called the ComboClass class. One instance of this class is instantiated for each foreign field drop-down list used to support a foreign field. A ComboClass object contains value description pairs. Each pair holds the foreign field value and its corresponding description.

For more information on the ComboClass class see **Maintenance Classes**, page 249, in *Construct Spectrum Reference Manual*.

Code is generated in the dialog's Load event to read all the rows from each referenced foreign file. The Load event uses the Visual Basic browse object to read the rows. The Load event then populates each drop-down list with the foreign field descriptions or foreign field values.

Note: Each referenced foreign file must have a corresponding Visual Basic browse object. Otherwise, the dialog Load event cannot read records from the foreign file.

Populating the foreign field drop-down lists in this way delays the initial opening of a dialog until all foreign field records are retrieved from the remote database. However, the object factory is optimized to read the remote database only the first time it is requested by the application. Thereafter, the data is cached globally, so that there is no delay when the dialog is opened again and the same data is required.

A VB-Browse-Object, generated for the foreign file that is intended to be looked up, must be available in order to support lookups. Since an application can be built incrementally, there is a possibility that a required VB-Browse-Object is not yet available. In this case, such a list will be disabled.

Support for Value Selection

Event code is generated to support selecting foreign field values from either the drop-down list or by typing a new value. In both cases, the list and the text box controls are synchronized with the choice made. For example, clicking on a description in a foreign field drop-down list updates the contents of the foreign field text box to match the field value for the selected description. Likewise, typing a new value into the foreign field text box will, on a `LostFocus` event, cause the corresponding description to display in the list.

If you enter an invalid value when typing directly in the foreign field text box, the list displays a blank indicating that this value is not in the local cache of valid values. Subsequent edit checks in the server object subprogram when the user selects the Update action will either pass or fail the value based on a live check in the foreign file's database.

Supporting Code for Command Buttons

This section explains how Construct Spectrum supports command buttons for foreign fields on a maintenance dialog. One situation where you may want to add command buttons for a foreign field is when other fields on the dialog derive their values from the foreign field. You could add a command button to allow users to update derived fields when a foreign field value changes. Read this section before hand-coding command buttons for foreign fields.

Initializing a Command Button

The maintenance dialog Load event is used to enable all the foreign field lookup command buttons on the dialog. This code verifies that a Visual Basic browse object exists to support each foreign field lookup button on the dialog. With incremental development, it is possible that some required Visual Basic browse objects are not available in the application. If a required Visual Basic browse object is not found, the button is made invisible.

Click Events on the Command Button

If a maintenance dialog contains a foreign field lookup button, it also contains event code to handle the button's click event. This code invokes the `BrowseByForeignKey` method of the Visual Basic maintenance object, passing the name of the foreign field relationship as a parameter.

A Visual Basic maintenance object handles all the logic required to work with a browse dialog linked to a foreign field. For example, when a user selects a new foreign field value from a foreign field browse dialog, the selected value is updated by the VB-Maint-Object in its internal Natural PDA. It also passes back a reference to a `BrowseDataCache` object. If the user does not select a value, the `BrowseDataCache` is set to `Nothing`.

Methods exposed by the `BrowseDataCache` object and its dependent objects are used by the maintenance dialog code in the `Click` event following the `BrowseByForeignKey` call to retrieve the newly selected foreign field descriptions and update these on the dialog.

INTERNATIONALIZING YOUR APPLICATION

This chapter describes the tools provided by Construct Spectrum to help you write internationalized applications. It also describes how to use each tool. Preparing applications so they readily translate into different languages ultimately saves development time.

The following topics are covered:

- **Planning Your Internationalized Application**, page 368
- **Internationalizing Using the Client Framework**, page 369
- **Resource File Syntax**, page 372
- **Using the Client Framework's Internationalization Components**, page 374
- **Hints for Developers**, page 380

For related information, see:

- **Resource Classes**, page 363, in *Construct Spectrum Reference Manual*

Planning Your Internationalized Application

Whether you are creating your Construct Spectrum application in two or more languages or considering translating the application in the future, design the application to take advantage of the internationalization client framework components supplied with Construct Spectrum.

Tip: You do not need to build internationalization components into your design when creating small applications or applications used in one location only. These internationalization components are optional.

To write internationalized applications, identify all text strings and graphics in the application that must be translated. These text strings and graphics include:

- window titles
- labels and prompts
- menu commands
- messages displayed to the user
- formatting strings for dates, times, and currency values
- toolbar button bitmaps
- icons

Organizing the text strings and graphics and copying them to external files is the first step in preparing an application for internationalization. You can then write code to load the files into the application at runtime. Translating the files into the required language localizes the application. Using this approach to localization means you alter the application's executable file only when adding another language option.

Internationalizing Using the Client Framework

Your Construct Spectrum project is supplied with internationalization client framework components, making it easy to create applications you can deploy in more than one language.

The client framework stores text and graphics for an application separate from the compiled executable code. This allows you to change these attributes without accessing source code for the application. To provide this feature, forms are designed to contain as little code as possible.

The two internationalization client framework components included with your Construct Spectrum project are:

- Resource, which reads resources from resource files.
- ResourceGroup, which returns a list of resources in a resource group.

The following list describes the components and how to use them:

- Text strings and graphics copied into external files are referred to as resources, the external files as resource files. To localize an application, translate the resource files into the required language.
- Each resource is identified by a resource identifier (RID) and has a type (string or binary) and value.
- Resources are collected into resource groups. Assign each resource group a resource group identifier (RGID).
- Both resource groups and their resources are defined in resource files. Each resource file has a name, which is the same as the filename without the path or extension. For example, a resource file may have a filename such as the following:

```
c:\MyProjects\SpectrumDemo\Forms.1
```

where:

filename	is Forms
path	is C:\MyProjects\SpectrumDemo
extension	is .1

Note: Resource files have a proprietary format. They are coded differently from Windows resource files maintained in a Windows resource editor.

- Resource files are organized in language sets. There is one language set for each user language (such as English, German, or French) the application supports. Each set contains one or more resource files. Each user language is identified with a 1-, 2-, or 3-character language code which is also used for the filename extension. All the resource files in a language set have the same filename extension.

An application uses only one language set at a time. The current language setting determines which language set the application uses. You can specify to use the same language codes as Natural (1=English, 2=German, 3=French...).

- Language sets, resource files, resource groups, and resources form a four-level hierarchy, as shown in the following example:

Example	Type
English (language code "1")	Language set
Framework.1	File
frmOpen	Group
lblObjects.Caption	Resource
lblActions.Caption	Resource
cmdOK.Caption	Resource
cmdCancel.Caption	Resource
frmBrowseDialogOptions	Group
lblLogicalKeyPrompt.Caption	Resource
frmAbout	Group
imgApplicationBitmap.Picture	Resource
GeneratedForms.1	File
frmCustomer	Group
lblCustomerName.Caption	Resource
frmOrder	Group
lblOrderNumber.Caption	Resource

Example (continued)	Type
Messages.1	File
General	Group
EndOfData	Resource
ActionInvalid	Resource
German (language set with language code "2")	Language set
Framework.2	File
frmOpen	Group
lblObjects.Caption	Resource
lblActions.Caption	Resource
cmdOK.Caption	Resource
cmdCancel.Caption	Resource
frmBrowseDialogOptions	Group
lblLogicalKeyPrompt.Caption	Resource
frmAbout	Group
imgApplicationBitmap.Picture	Resource
GeneratedForms.2	File
frmCustomer	Group
lblCustomerName.Caption	Resource
frmOrder	Group
lblOrderNumber.Caption	Resource
Messages.2	File
General	Group
EndOfData	Resource
ActionInvalid	Resource
French (language set with language code "3")	Language set
...	

- The client framework uses a resource file path (similar to a DOS file search path) to search for resource files. The path is specified in the application startup code.
- Instead of providing a type and a value for a resource, you can link it to another resource. When the resource is accessed, the application gets the type and value by following the link. The type and value can link to another resource with its own type and value, and so on.

Links allow you to specify the value for a resource once and use that value in many locations. For example, if you have OK and Cancel buttons on many different dialogs and you want to change the captions on these buttons on all dialogs, you could define two resources that provide the captions and link to them from all the dialogs.

Note: Links must terminate in a type and value pair. Circular links are not allowed.

Resource File Syntax

Resource files are text files that use a syntax identical to Windows INI files. Resource groups are specified like INI file sections, and resources are specified like INI file keys.

Specify resource IDs to the left of the equal sign, and specify resource values to the right of the equal sign.

Text Values

Specify text values with quotation mark delimiters, for example:

```
EndOfDataMsg="There are no more records that match the search criteria."
```

To include non-printing characters in text values, specify them with one of the escape sequences listed below. Note that these escape sequences are case-sensitive:

Escape Sequence	Non-printing Character
<code>\nl</code>	CR-LF character combination (ASCII 13 ₁₀ 10 ₁₀)
<code>\cr</code>	CR character (ASCII 13 ₁₀)
<code>\lf</code>	LF character (ASCII 10 ₁₀)
<code>\tb</code>	Tab character (ASCII 9 ₁₀)
<code>\nnn</code>	Character corresponding to ANSI code <i>nnn</i> ₁₀ The “10” notation above indicates decimal numbering.
<code>\\</code>	Backslash character.

Binary Values

Specify binary values as either a sequence of hex characters or as a reference to an external file. For a sequence of hex digits, use the value "BIN:" followed by the byte values. For an external file, use the value "FILE:" followed by the filename and an optional hex starting position and hex length, for example:

```
Image1=BIN:01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10
Image2=FILE:FileOpen.bmp
Image3=FILE:Icons.dat,1F00,0300
```

Note: External files must reside in the same directory as the resource file.

Links

A resource value may be linked to another resource. To create a link, specify "LINK:" followed by the name of the resource file (optional), the resource group (optional), and the resource ID. Use commas to separate the names of the resource file, group and ID, for example:

```
cmdOK.Caption=LINK:Global,GUIControls,OKButton
```

The commas must be included even if you omit an optional name, for example:

```
lblHeader(1).Caption=LINK: , lblPrompt(1).Caption
```

If you omit the resource group, the resource file must be omitted too. In this case, the resource ID is assumed to be in the same resource file and group. If the resource file is omitted, but the resource group is provided, the resource ID is assumed to be in the same resource file, for example:

```
lblPrompt(1).Caption=LINK: , frmCustomerBrowse, lblPrompt(1).Caption
```

Using the Client Framework's Internationalization Components

The Resource class provides methods to read resources from resource files and to reduce the effort needed to localize an application.

The Construct Spectrum client framework declares and initializes an instance of this class in Startup.bas, for example:

```
Public Res As New CST.Resource
```

For more information about these methods and properties, see **Resource Classes**, page 363, in *Construct Spectrum Reference Manual*.

Methods

The Resource class uses the following methods to localize applications:

- GetResourceGroup
- LocalizeForm
- LoadBinaryResource
- LoadStringResource
- Message
- MessageEx
- SetDefaultMessageGroup

GetResourceGroup

This method creates a ResourceGroup object that returns a list of the resources in a resource group.

The syntax is:

```
Set result = object.GetResourceGroup(ResourceFile, ResourceGroup)
```

If the resource file does not exist or if the resource group does not exist in the resource file, this method returns “Nothing”.

LocalizeForm

This method localizes a form by iterating through all of the resources in the specified resource group and loading each resource into a corresponding control property.

The syntax is:

```
Sub LocalizeForm(Form As Form, _  
                ResourceFile As String, _  
                ResourceGroup As String)
```

This method works with text and graphic properties. For example, the resources might look like this:

```
Form.Caption="Construct Demo Application"  
mnuFile.Caption("&File"  
mnuFileOpen.Caption("&Open..."  
imgApplicationBitmap.Picture=FILE:App.ico  
...
```

This method is very powerful; one line of code in your form will localize all the visual GUI controls on your form. To use this method, call it from your form's Load event. The following example uses a resource file called Forms which contains resource groups with the same names as the forms in your application (Me.Name):

```
Private Sub Form_Load ()  
    Res.LocalizeForm Me, "Forms", Me.Name  
End Sub
```

LoadBinaryResource

This method loads the specified resource and returns it as a Byte array. It returns Null if the resource cannot be found.

The syntax is:

```
Function LoadBinaryResource(ResourceFile As String, _  
                           ResourceGroup As String, _  
                           ResourceID As String) As Variant
```

LoadStringResource

This method loads the specified resource and returns it as a string. It returns an empty string if the resource cannot be found.

The syntax is:

```
Function LoadStringResource(ResourceFile As String, _  
                           ResourceGroup As String, _  
                           ResourceID As String) As String
```

Message

This method returns a resource identified by a resource ID. The resource file and resource group are not specified in this method; they are specified by calling the SetDefaultMessageGroup method.

The syntax is:

```
result = object.Message(ResourceID, DefaultMessage, Substitutions...)
```

Before using this method, you must set the default resource file and resource group by calling the SetDefaultMessageGroup method. Once you have set the default resource file and group, you can call the Message method repeatedly without having to specify the resource file and resource group each time.

The Substitutions argument is optional. Use it to pass as many substitution parameters as are required by the message. If you do not pass enough substitution parameters, the remaining ones in the message will be replaced by “***”.

MessageEx

This method returns a resource identified by a resource file, resource group, and resource ID.

The syntax is:

```
result = object.MessageEx(ResourceFile, ResourceGroup, ResourceID, _  
                          DefaultMessage, Substitutions...)
```

The Substitutions argument is optional. Use it to pass as many substitution parameters as are required by the message. If you do not pass enough substitution parameters, the remaining ones in the message will be replaced by “***”.

SetDefaultMessageGroup

This method sets the default resource file and resource group used by the Message method when loading resources.

The syntax is:

```
object.SetDefaultMessageGroup ResourceFile, ResourceGroup
```

Properties

This section discusses the properties of the Resource class used in localizing an application. These properties include:

- Language
- LanguageRegistryKey
- LanguageINIKey
- ResourceFilePath

Specifying Language, LanguageRegistryKey, and LanguageINIKey properties sets the language code used for all resource lookups. The most recently set of these three properties overrides the settings of the other two properties. Use ResourceFilePath to specify a search path for resources.

Language

This property sets the language code used for all resource lookups.

The syntax is:

```
Language As String
```

You must define a mapping between language codes and user languages. For example, you could choose to use the same language codes that Natural uses (1 for English, 2 for German, 3 for French...).

When accessing a resource, the Resource class uses this language code as a filename extension to obtain the filename of the resource file. For example, if Language contains “1” and you use the following method:

```
strResource = Res.LoadStringResource("Forms", "frmOpen", "Caption")
```

the resource class would look for a file called “Form.1” in the resource path.

Read this property to obtain the current language setting if either LanguageRegistryKey or LanguageINIKey has been used to specify the language setting.

LanguageRegistryKey

The language code is automatically read from this Windows Registry key.

The syntax is:

```
LanguageRegistryKey As String
```

Use LanguageRegistryKey to specify a valid registry key, beginning with one of:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

and ending with a value name. For example:

```
.LanguageRegistryKey = "HKEY_CURRENT_USER\" & _  
                      "Software\" & _  
                      "SAGA SOFTWARE INC.\" & _  
                      "CST Frameworks\" & _  
                      "Language"
```

For every call to LocalizeForm, LoadStringResource, or LoadBinaryResource, the current value of this setting will be read to determine which language set to use.

LanguageINIKey

This property is similar to `LanguageRegistryKey`, but the language setting is automatically read from this .INI key.

The syntax is:

```
LanguageINIKey As String
```

Use `LanguageINIKey` to specify a valid .INI file, section, and key name, each separated by a Tab character. For example:

```
.LanguageINIKey = "C:\Windows\CST411.INI" & vbTab & _  
                 "Settings" & vbTab & _  
                 "Language"
```

ResourceFilePath

This property sets the resource file path used to search for resource files.

The syntax is:

```
ResourceFilePath As String
```

Paths are separated by the semicolon character. For example:

```
.ResourceFilePath = "\\SERVER\Resources;" & _  
                  "C:\Program Files\Demos\Demo1"
```

Setting the `ResourceFilePath` property allows resource files to reside in multiple locations. You will want to store resource files used by many different applications on a shared network resource and store application-specific resource files in that application's directory.

Hints for Developers

The following sections provide information to help you use Construct Spectrum's internationalizing features to the maximum advantage.

Automatically Setting the Language

The Resource class reads the current language setting and uses that information to access the language set. This choice is made before the Resource class loads any resources. This structure allows you to centralize the language setting and have changes to that setting automatically reflected across all applications.

To set language automatically, ensure that all applications using the Resource class share a standard LanguageRegistryKey or LanguageINIKey. If all applications standardize on a specific Registry key or .INI file key to store the current language, then changing the language in one application sets the language in all applications.

Strategy for Using Resource Files and Groups

To organize resource files and groups efficiently, use one resource file for each major component (or layer) of the application being localized. For example, you might separate your resources into the following files:

- resources used by all framework components
- resources used by all application-specific components
- resources shared by all application components and layers, for example, OK and Cancel button prompts. Link other resources to the resources in this file.

Within each resource file, consider using one resource group for the GUI controls of each form. This approach makes it easy to use the LocalizeForm method. Then use another resource group for messages and other resources that are not necessarily linked to GUI control properties, for example, .Caption or .Text.

Construct Spectrum supplies two resource files that implement internationalization of framework components and shared application components. These files are called Fwk.* and Global.* respectively.

The Visual Basic maintenance models (VB-Maint-Dialog and VB-Maint-Object) are designed to generate code that looks for resources in a resource file called App.*. Partition the resources for your application using this scheme.

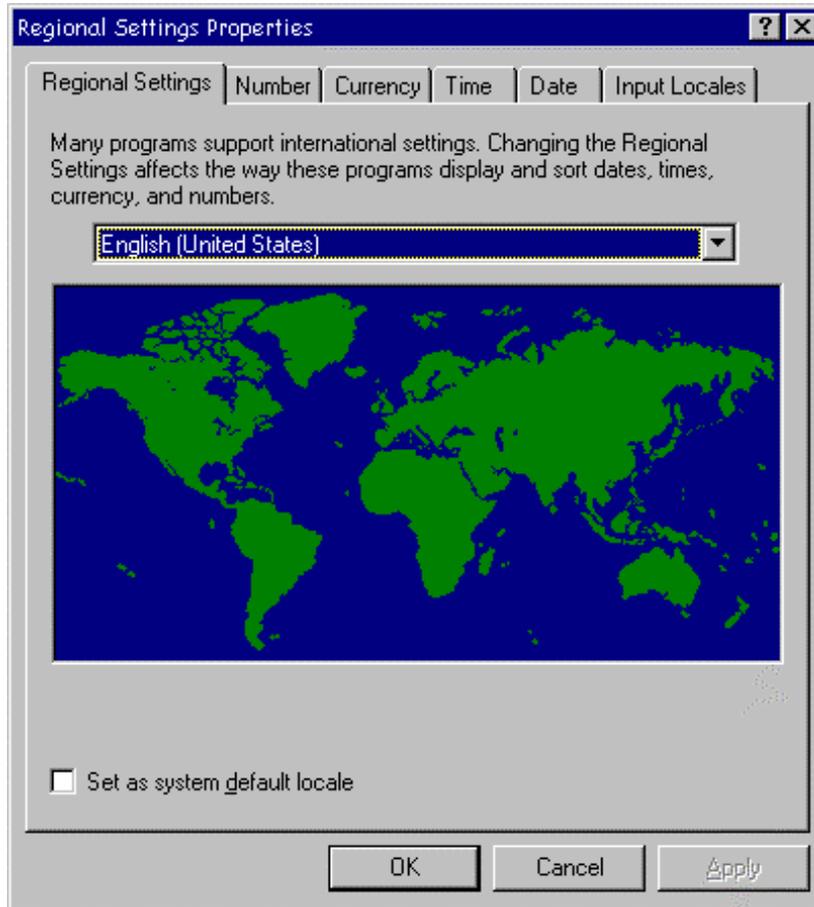
By default, the Construct Spectrum framework looks for resource files in the application directory. If you are developing an international application, you will need to ensure that all necessary resource files reside in the application directory. If you follow the recommended partitioning of resources described above, you need to copy the Fwk.* and Global.* resource files from the Framewrk directory to your application directory. Next, you need to create App.* resource files and create resources for your application-specific forms and messages.

Starting an Application in a Specific Language

Construct Spectrum applications automatically provide the ability to start in a specific language. By interrogating the Windows locale setting and mapping it to a specific language code, you can specify a language other than English. When each form in the application is loaded, its Form_Load event calls a Localize method. The Localize method converts the form so it is displayed in the language indicated by the Windows locale setting.

You may want to test your application with a different Windows locale setting to ensure that all captions on the application forms are properly formatted.

- To change your Windows locale setting:
 - 1 On the Windows Start menu, click **Settings** and then **Control Panel**. The Control Panel dialog is displayed.
 - 2 Select **Regional Settings**. The Regional Settings Properties dialog is displayed:



Specifying the Language in the Regional Settings Properties Dialog

- 3 Select the **Regional Settings** tab.
- 4 Select the desired locale from the drop-down list.

Associating Windows Locale Setting with a Language

The Windows locale setting is mapped to a language code by the `GetUserDefaultNATLangCode` function (located in `CSTUtils.bas`). This function returns a Language code, using the same language codes as Natural (for example, 1=English and 2=German) based on the Windows locale setting. Use this value to set `Res.Language`, where `Res` is a global reference to the Resource class. The mapping of locale setting to language code is implemented with the `MAPPING` constant, as depicted in the following code example:

Example of using the MAPPING constant

```
Public Function GetUserDefaultNATLangCode() As Integer
...
' This constant defines the mapping between Windows language IDs and
' Natural language codes. Entries have the format nn=ww, where nn is
' the Natural language code and ww is the Windows language ID.
Const MAPPING = "01=09,02=07,03=12,04=10,05=16,06=19,07=31,... "
...
End Function
```

Changing Language at Runtime

To support changing the user language at runtime:

- The user interface must include a function to change the language, for example, a menu command, keystroke combination, or button.
- Each form must implement a localization procedure that localizes the form, perhaps by calling the `LocalizeForm` method.
- The localization procedure must be called both when the form loads and whenever the user changes the language at runtime. To implement changing the language at runtime, declare the localization procedure as public. When the user changes the language, the event code iterates through all loaded forms and calls their localization procedures, as shown in the following example:

```
Public Sub LocalizeAllLoadedForms
    ' Called whenever the user changes the language at run-time.
    Dim frm As Form
    For Each frm In Forms
        ' Use an error handler in case the form doesn't have a
        ' Localize procedure.
        On Error Resume Next
        frm.Localize
        On Error Goto 0
    Next
End Sub
```

Note: The client framework includes the `LocalizeAllLoadedForms` procedure and all generated forms support the `Localize` method. However, you must code the user interface command to invoke this procedure if you are developing an application that can change language at runtime.

APPENDIX: MODIFYING SPECTRUM MODELS

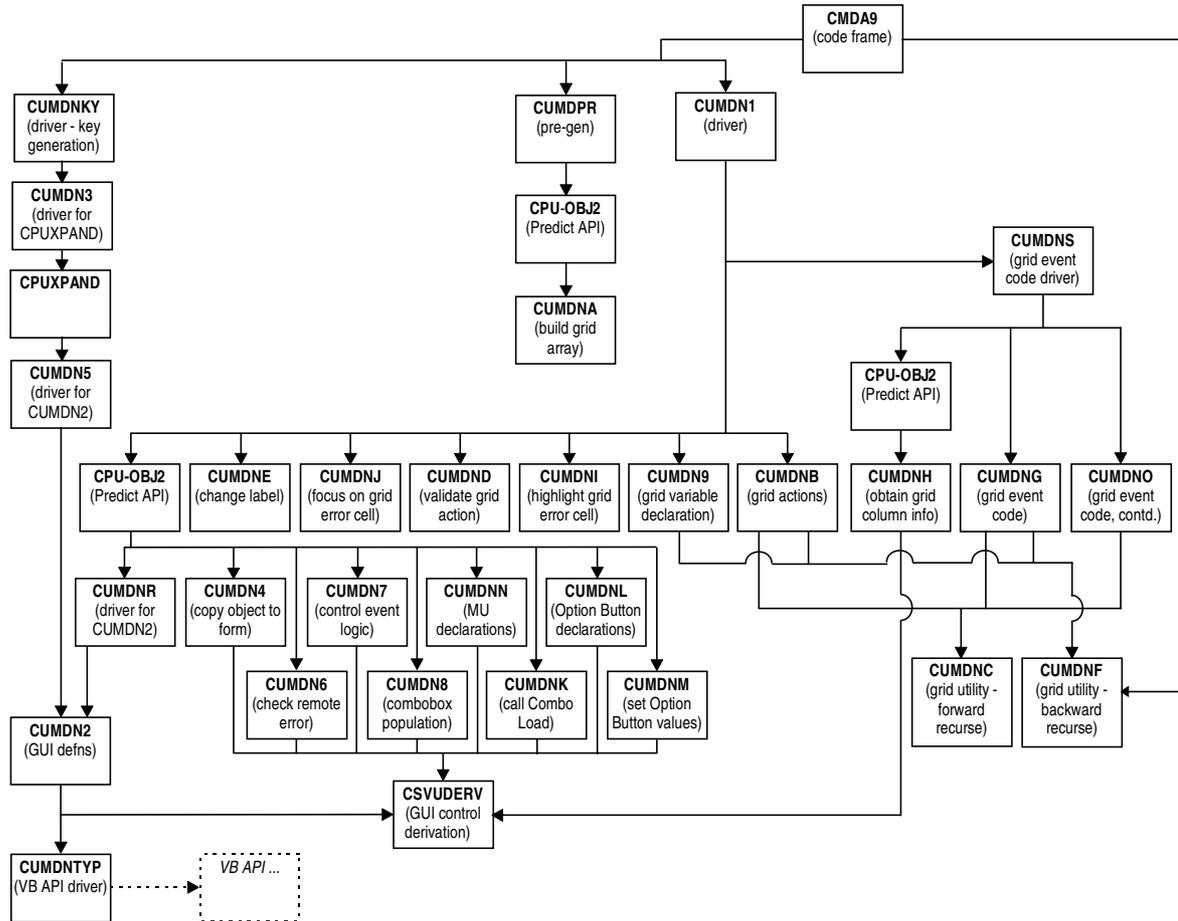
This appendix provides a guideline to follow when creating new models based on the VB-Maint-Dialog model. Use this appendix to learn about the relationships among the components used to generate maintenance dialogs.

The following topics are covered:

- **VB-Maint-Dialog Model**, page 386
- **VB API**, page 388
- **How the VB API Works**, page 389
- **GUI Controls with VB API**, page 391
- **Parameter Data Area (PDA) Used**, page 397

VB-Maint-Dialog Model

A variety of components participate in the generation of VB maintenance dialogs. The illustration of the model architecture for the VB-Maint-Dialog model shows the relationships among these components. Use this illustration as a guide if you plan to change the VB-Maint-Dialog model or create your own GUI models:



Architecture of the VB-Maint-Dialog Model

As the illustration shows, many of the routines are called by CPU-OBJ2. CPU-OBJ2 accepts a Predict file name and a subprogram name. CPU-OBJ2 calls this subprogram for each field in the Predict file. The subprograms generate segments of code based on the Predict information that is passed by CPU-OBJ2. For example, CUMDN4 generates Visual Basic code that copies the contents of each field to a related GUI control.

Example of generated code

```
Private Sub CopyObjectToForm

    InhibitValidations = True
    On Error GoTo FormAssignmentError
    With InternalObject
        txt_Empl_PersonnelId.Text = _
            BDT.ConvertToDisplay(.Field("PERSONNEL-ID"), _
                NatFormatLength:="A8")
        txt_Empl_FirstName.Text = _
            BDT.ConvertToDisplay(.Field("FIRST-NAME"), _
                NatFormatLength:="A20")
    ...

```

VB API

The VB-Maint-Dialog model uses a series of Natural subprograms that generate Visual Basic definitions into the source area. Collectively, these Natural subprograms are called the VB API. The VB-Maint-Dialog model uses the VB API to generate the visual definition — the various GUI controls — of a VB maintenance dialog. If your models generate Visual Basic forms, they can also use the VB API.

Components of the VB API

Three components exist for each type of GUI control supported by the VB API:

- a subprogram to assign user-defined default values for the properties of a GUI control
- an LDA to store the Visual Basic default values for the properties of a GUI control
- a subprogram to write the GUI definition to the source area

A series of PDAs store property information for GUI control definitions. GUI control properties are grouped by function into different PDAs. For example, all GUI control properties related to font are stored in the CSVAFONT PDA. Any GUI control that implements font properties declares the font PDA, CSVAFONT.

To see the list of GUI controls supported by the VB API, see **GUI Controls with VB API**, page 391. For each GUI control, the table in this section indicates:

- the subprogram responsible for assigning user defaults
- the subprogram responsible for writing the GUI definition to the source area
- the PDAs that must be passed to these subprograms

How the VB API Works

- You use the VB API with a model you create:
- 1 Call the user default subprogram.
The user default subprogram assigns your organization's defaults for GUI control properties. With this subprogram, you can write code to assign default values once — not in every subprogram that uses the VB API. For example, suppose your organization requires the field captions on all dialogs to be in an eight-point MS Sans Serif font. Writing the following code in the user default subprogram CSVBDLBL (for Label GUI controls) assigns the organization's required values.

Example of code in the user default subprogram

```
COMPRESS #DOUBLE-QUOTE 'MS Sans Serif' #DOUBLE-QUOTE  
        INTO CSVAFONT.FONT_NAME LEAVING NO SPACE  
ASSIGN CSVAFONT.FONT_SIZE = 8
```

For more information, see **Setting Generation GUI Standards**, page 172.

- 2 Assign any GUI control properties that are application-specific.
For example, the Caption property of the Label GUI control varies because it is based on the name of the database field with which it is associated. Therefore, you would not want to assign this type of GUI control property in the control's default subprogram.

Example of assigning a value to the Caption property

```
CSVAFRMT.CAPTION := CPA-ODAT.FIELD-NAME
```

For another example, see the CUMDNTYP driver program for the VB-Maint-Dia-log model.

- 3 Call the Create subprogram that writes the GUI definition to the source area.
The Create subprogram compares the value assigned to a particular GUI control property with the default value used by Visual Basic. If the assigned value differs from the Visual Basic default value, the Create subprogram generates the property assignment into the source area. However, if the assigned value matches the Visual Basic default value, the Create subprogram saves source area space by suppressing generation of the property assignment.

Consider the FONT_NAME and FONT_SIZE properties set in the earlier example. Visual Basic's default property values for a Label GUI control are an eight-point font and a MS Sans Serif font. The Label GUI control definition (generated by CS-VBCLBL) shown below does not include assignments for the font name and size.

Example of using default values

```
Begin VB.Label lbl_Empl_PersonnelId
  Caption = "Personnel/id:"
  AutoSize = -1
  Left = 100
  Top = 295
  Height = 285
  Width = 1073
End
```

GUI Controls with VB API

The following table lists the GUI controls the VB-Maint-Dialog model uses. Also included are the subprogram names and parameter data areas (PDA) associated with each GUI control:

GUI Control	User Default	Create	PDAs
CheckBox	CSVBDCHK	CSVBCCHK	CUMDATYP CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSVATOGL CSASTD
3DCheckBox	CSVBD3CH	CSVBC3CH	CUMDATYP CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSVATOGL CSASTD
ComboBox	CSVDCBO	CSVCCBO	CSVACMBO CSVACOMN CSVAFOCS CSVAFONT CSVALBOX CSVALCTN CSVAMOUS CSVATBOX CSASTD

GUI Control	User Default	Create	PDA's (continued)
CommandButton	CSVBDCMD	CSVBCCMD	CUMDATYP CSVACOMN CSVABUTN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSASTD
3Dcommand Button	CSVBD3CD	CSVBC3CD	CUMDATYP CSVALCTN CSVACOM CSVA3CMD CSASTD
Form	CSVBDFRM	CSVBCFRM	CUMDATYP CSVACOMN CSVADDE CSVAFONT CSVAFORM CSVAFRMT CSVALCTN CSVAMOUS CSVAWNDW CSASTD
Frame	CSVBFRA	CSVBCFRA	CUMDATYP CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSVAWNDW CSASTD

GUI Control	User Default	Create	PDA's (continued)
3DFrame	CSVBD3FR	CSVBC3FR	CUMDATYP CSVA3DI CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSASTD
Label	CSVBDLBL	CSVBCLBL	CUMDATYP CSVACOMN CSVADDE CSVAFOCS CSVAFONT CSVAFRMT CSVALABL CSVALCTN CSVAMOUS CSASTD
ListBox	CSVBDLST	CSVBCLST	CUMDATYP CSVACOMN CSVAFOCS CSVAFONT CSVALBOX CSVALCTN CSVAMOUS CSVAWNDW CSASTD
MDIForm	CSVBDMFM	CSVBCMFM	CUMDATYP CSVACOMN CSVAFOCS CSVAFRMT CSVALCTN CSVAWNDW CSASTD

GUI Control	User Default	Create	PDA's (continued)
Menu	CSVBDMNU	CSVBCMNU	CUMDATYP CSVACOMN CSVAFOCS CSVAFRMT CSVAMENU CSASTD
OptionButton	CSVBDOPT	CSVBCOPT	CUMDATYP CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSVATOGL CSASTD
3DOptionButton	CSVBD3OP	CSVBC3OP	CUMDATYP CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSVATOGL CSASTD
StatusBar	CSVBDSTA	CSVBCSTA	CUMDATYP CSVACOMN CSVA3DI CSVAFOCS CSVALCTN CSVASTAT CSASTD

GUI Control	User Default	Create	PDA's (continued)
TextBox	CSVBDTXT	CSVBCTXT	CUMDATYP CSVACOMN CSVADDE CSVAFOCS CSVAFONT CSVAFRMT CSVALCTN CSVAMOUS CSVATBOX CSVAWNDW CSASTD
Timer	CSVBDTMR	CSVBCTMR	CUMDATYP CSVACOMN CSVALCTN CSVATIME CSASTD
3DPanel	CSVBD3PN	CSVBC3PN	CUMDATYP CSVA3DI CSVA3DPN CSVACOMN CSVAFOCS CSVAFONT CSVAFRMT CSVALABL CSVALCTN CSVAMOUS CSASTD
TrueDBGrid	CSVBDGRD	CSVBCGRD	CUMDATYP CSVACOMN CSVALCTN CSVAFOCS CSVAGRID CSASTD



GUI Control	User Default	Create	PDA's (continued)
Toolbar	CSVBDTLB	CSVBCTLB	CUMDATYP CSVA3DI CSVACOMN CSVAFOCS CSVAFONT CSVAMOUS CSVATOOL CSASTD

Parameter Data Area (PDA) Used

The following table lists the PDAs used with the VB-Maint-Dialog model. Included are the properties associated with each PDA and the GUI controls that use the PDA. These PDAs are also cross-referenced by GUI control and subprogram in **GUI Controls with VB API**, page 391.

Some of the properties are identified with superscript numbers. When a GUI control is shown with a superscript number, the corresponding property is not used. For example, the first PDA in the following table has a BackColor property identified with a superscript number of 1. The GUI control 3DCheckBox field also has a superscript value of 1. This means that the 3DCheckBox field does not use a BackColor property.

PDA Name	Properties	Used By GUI Control
CSVACOMN (common information)	BackColor ¹ Enabled ² Index ³ Name ⁴ Tag ⁵ Visible ⁶	CheckBox 3DCheckBox ¹ ComboBox CommandButton Form ³ Frame 3DFrame ¹ Label ListBox MDIForm ^{1,3,6} Menu OptionButton 3DOptionButton ¹ StatusBar ^{1,2,3,4,5,6} TextBox Timer ^{1,6} 3DPanel TrueGridPro ToolBar

PDA Name	Properties	Used By GUI Control (continued)
CSVAFOCS (focus information)	HelpContextID ¹ TabIndex ² TabStop ³	CheckBox 3DCheckBox ComboBox CommandButton MDIForm ^{2, 3} Menu ^{2, 3} OptionButton 3DOptionButton StatusBar ^{1,3} TextBox 3DPanel ³ TrueGridPro ToolBar ^{2, 3}
CSVATOGL (toggle information)	Value	CheckBox 3DCheckBox OptionButton 3DOptionButton
CSVAFRMT (text formatting information)	Alignment ¹ BorderStyle ² Caption ³	CheckBox ² 3DCheckBox ² CommandButton ^{1,2} Form ¹ Frame ^{1,2} 3DFrame ² Label MDIForm ^{1,2} Menu ^{1,2} OptionButton ² 3DOptionButton ² TextBox ³ 3DPanel ² TrueGridPro ^{1,3}

PDA Name	Properties	Used By GUI Control (continued)
CSVAMOUS	DragIcon ¹ DragMode ² MousePointer ³	CheckBox 3DCheckBox ComboBox CommandButton Form ^{1,2} Frame 3DFrame Label ListBox OptionButton 3DOptionButton TextBox 3DPanel TrueGridProToolBar
CSVAFONT (font information)	FontBold ¹ FontItalic ² FontName ³ FontSize ⁴ FontStrikethru ⁵ FontTransparent ⁶ FontUnderline ⁷ Font3D ⁸ ForeColor ⁹	CheckBox ^{6,9} 3DCheckBox ⁶ ComboBox ^{6,8} CommandButton ^{6,8,9} Form ⁸ Frame ^{6,8} 3DFrame ⁶ Label ^{6,8} ListBox ^{6,8} OptionButton ^{6,8} 3DOptionButton ⁶ TextBox ^{6,8} 3DPanel ⁶ TrueGridPro ^{6,8} ToolBar ^{6,8,9}

PDA Name	Properties	Used By GUI Control (continued)
CSVALCTN (location information)	Left ¹ LeftDerive Top ² TopDerive Height ³ HeightDerive Width ⁴ WidthDerive	CheckBox 3DCheckBox ComboBox CommandButton Form Frame 3DFrame Label ListBox MDIForm OptionButton 3DOptionButton StatusBar TextBox Timer ^{3,4} 3DPanel TrueGridPro
CSVADDE (DDE information)	LinkItem ¹ LinkMode ² LinkTimeout ³	Form ^{1,3} LabelTextBox
CSVAFORM (form control information)	AutoRedraw ControlBox DrawMode DrawStyle DrawWidth FillColor FillStyle KeyPreview MaxButton MDIChild MinButton Picture	Form
CSVALABL (label control information)	AutoSize ¹ BackStyle ² WordWrap ³	Label 3DPanel ^{2,3}

PDA Name	Properties	Used By GUI Control (continued)
CSVAMENU (menu control information)	Checked ShortCut	Menu WindowList
CSVABUTN (command button control information)	Cancel Default	CommandButton
CSVALBOX (listbox control information)	Columns ¹ MultiSelect ² Sorted ³	ListBox ComboBox ^{1,2}
CSVASTAT (status bar information)	_Version _ExtentX _ExtentY _StockProps SimpleText	StatusBar
CSVATBOX (textbox control information)	HideSelection ¹ MaxLength ² MultiLine ³ PasswordChar ⁴ Text ⁵	TextBox ComboBox ^{1,2,3,4}
CSVATIME (timer control information)	Interval	Timer
CSVACMBO (combobox control information)	Style	ComboBox
CSVA3DI (3D information)	Align ¹ Outline ² ShadowColor ³ ShadowStyle ⁴	3DPanel ⁴ 3DFrame ^{1,2} ToolBar ^{3,4}

PDA Name	Properties	Used By GUI Control (continued)
CSVAWNDW (window information)	ClipControls ¹ Icon ² Scrollbars ³ WindowState ⁴	Form ³ Frame ^{2,3,4} MDIForm ¹ TextBox ^{1,2,4}
CSVA3DPN (3D panel information)	BevelInner BevelOuter BevelWidth BorderWidth FloodColor FloodPercent FloodShowPct FloodType RoundedCorners	3DPanel
CSVAGRID (TrueDBGrid control information)	OLEObjectBlob	Grid
CSVA3CMD		

INDEX

A

- Active help
 - diagram, 347
 - drop-down list
 - diagram, 348
 - for maintenance dialogs, 345
 - foreign field help, 346
 - primary key help, 345
- Adding a new field by hand
 - on maintenance dialog, 185
- Add-Ins Menu
 - options, 27
- Appendix
 - See* Modifying Spectrum models
- AppendSlash
 - utility procedure, 323
- Application interface
 - demo project, 59
- Application settings
 - AppSettings.bas
 - definition, 262
 - customizing, 262
 - Startup.bas
 - definition, 262
 - understanding, 262, 265
- ApplicationName
 - description, 263
- AppSettings.bas
 - description, 129
 - See also* application settings, 262
- Architecture
 - Construct Spectrum applications, 28

- ArrayDimensions
 - utility procedure, 323
- ASSERT
 - utility procedure, 323
- Assigning
 - corporate defaults, 170

B

- BDT_PHONE
 - business data type, 328
- Browse
 - modules
 - downloading to project, 229
 - support
 - ApplicationControl, 265
 - ApplicationControls, 265
 - BrowseBase.cls, 265
 - BrowseDataCache, 265
 - BrowseDataColumn, 265
 - BrowseDataColumns, 265
 - BrowseDataRow, 265
 - BrowseDataRows, 265
 - BrowseDialogBase.cls, 265
 - BrowseDialogOptions.frm, 265
 - BrowseManager.cls, 266
 - ColumnDisplay, 266
 - ColumnsDisplay, 266
 - FieldKey, 266
 - FieldKeys, 266
 - GenericBrowse.frm, 266
 - GenericMDIBrowse.frm, 266
 - KeyMatch, 266

- understanding, 265
 - Browse Command handlers
 - coding, 241
 - enabling browse commands, 242
 - example of code to assign command IDs, 242
 - example of code to mark updated rows, 244
 - example of code to update, 243
 - creating, 240
 - diagram of Browse Command handler interaction
 - process of browse command handler object interaction, 239
 - diagram of Browse command handler interaction, 239
 - Browse command handlers, 344
 - Browse dialogs
 - browse object subprogram, 217
 - browse object subprogram proxy, 218
 - components of
 - client framework components, 216
 - object browse subprogram, 216
 - object browse subprogram proxy, 216
 - Visual Basic browse object, 216
 - creating with individual models, 215
 - diagram of components, 217
 - drilling down from, 344
 - framework components, 219
 - integrating with maintenance dialogs, 343
 - see also* integrating browse and maintenance dialogs
 - modules required for, 102
 - prerequisites for generating with individual models, 221
 - purpose, 216
 - Visual Basic browse object, 218
 - data cache, 219
 - Browse object
 - See also* Visual Basic browse object
 - see* Visual Basic browse object subprogram
 - generating, 222
 - Browse subprogram proxy
 - generating, 222
 - BrowseManager class
 - BrowseManager methods
 - list of services, 237
 - BrowserExists procedure
 - (TableName) As Boolean, 306
 - example code, 317
 - updating, 317
 - Browsing for business objects
 - customizing browse options, 86
 - demo project, 82
 - Business data types
 - demo project, 76
 - setting up in Predict
 - example code for, 166
- ## C
- Calculated fields
 - code examples, 176
 - creating, 175
 - CenterForm
 - utility procedure, 323
 - CheckBox field
 - adding to maintenance dialog, 188
 - CheckBox grid column
 - adding to maintenance dialog, 198
 - Checklists
 - Construct Spectrum project, 126
 - creating browse dialogs with individual models, 221
 - creating maintenance dialogs with individual models, 140

- moving non-object based applications to Construct Spectrum, 248
- moving object-based applications to Construct Spectrum, 247
- super model generation, 97
- Client framework
 - customizing
 - application settings, 262
 - menu and toolbar
 - See* menu and toolbar, 289
 - object factory, 307
 - diagram of components, 255
 - internationalizing
 - See* Internationalizing, 374
 - introduction, 254
 - multiple-document interface, 301
 - object factory, 302
 - Resource class, 374
 - initializing an instance, 374
 - understanding and customizing, 253
 - utility procedures, 323
- Client modules
 - generation overview, 39
- Client/server applications
 - architecture, 28
- CLIENT-VALIDATION user exit
 - validating data, 334
- ComboBox GUI control
 - adding to maintenance dialog, 187
- Command buttons
 - foreign field support, 364
- Command handlers
 - browse, 344
- Commands
 - defining, sending, and handling, 273
- Compressing data
 - enabling for client to server transmissions, 145–146, 225
- Construct Spectrum
 - creating your application, 37
 - description, 24
 - documentation, 20
 - moving Natural Construct applications to, 18
- Construct Spectrum Add-In
 - overview, 49
- Construct Spectrum applications
 - diagram of architecture, 28
- Construct Spectrum project
 - creating, 127
 - downloading generated components to, 131
 - prerequisites, 126
 - setting up, 123
- Construct Spectrum SDK
 - documentation, 20
- Conventions
 - typographical, 19
 - used in this guide, 19
- Corporate defaults
 - assigning, 170
- Create a New Project dialog
 - description, 127
- CreateArray
 - utility procedure, 323
- CreateForm procedure
 - description, 313
 - example code, 314
 - updating, 313
- CreateStringArray
 - utility procedure, 323
- Creating
 - applications, 37
 - calculated fields, 175
 - Construct Spectrum applications, 18
 - Construct Spectrum Project, 127
- CSTFormatMessage
 - utility procedure, 323
- CSTSelectContents
 - utility procedure, 324

- CSTSubst
 - utility procedure, 324
 - CSTUtils.bas
 - utility procedures, 323
 - CSTVBFW.dll
 - customizing client framework components, 256
 - CSXDEFAULT
 - changing values in model default subprogram, 358
 - Customizing
 - application and environment, 40
 - browse dialog
 - display options, 86
 - recommendations for a new application, 232
 - browse dialogs, 238
 - BrowseManager methods, 237
 - diagram of internal structure, 236
 - on the client
 - See* Customizing on the client, 238
 - on the server
 - See* Customizing on the server, 238
 - understanding Browse Command handlers
 - See* Browse Command handlers, 238
 - using BrowseManager class, 234
 - business data types, 220
 - descriptive fields, 220
 - maintenance dialog
 - overriding default GUI control selection, 160
 - server options, 157
 - server tasks, 159
 - strategies for, 155
 - user-defined user exits, 157
 - Customizing browse dialogs
 - using the BrowseManager class, 233
 - diagram of interaction to display a browse dialog, 234
 - Customizing on the client
 - understanding the BrowseManager class
 - displaying the browse dialog, 232
 - returning a specific row of data, 233
 - returning all rows of data, 233
 - supporting a browse command handler, 233
 - Customizing on the server
 - browse object Predict setup, 238
- ## D
- Data compression
 - enabling for client to server transmissions, 145–146, 225
 - Data encryption
 - enabling for client to server transmissions, 145–146, 225
 - Data sources
 - defining alternate, 226
 - Database ID
 - specifying in a new project, 127
 - DBID
 - description, 264
 - number
 - specifying in a new project, 127
 - Debugging
 - client/server applications, 40
 - Default GUI derivation logic, 165, 167
 - diagram, 168–169
 - DefaultLibrary
 - description, 264

- Defaults
 - used by super model, 97
- Defining
 - alternate browse data sources, 226
 - example code, 228
 - general package parameters, 115
 - specific package parameters, 117
- Demo application
 - application interface, 59
 - browsing for business objects, 82
 - business data types, 76
 - customizing browse options, 86
 - drop-down grids, 80
 - foreign fields on a maintenance dialog, 84
 - generated modules, 53
 - grids, 78
 - maintaining a business object, 73
 - making the .EXE file, 58
 - nested grids, 79
 - opening a business object, 68
 - overview, 44, 50
 - remote dispatch service options, 66
 - running, 56
 - troubleshooting, 91
 - validations, 74
- Dependencies between models, 103
- Deploying
 - procedure, 41
- Deploying applications
 - overview, 41
- Derivation logic
 - GUI controls, 159
- Descriptions
 - foreign fields, 359
 - refreshing, 361
- Developing Client/Server Applications
 - how to use guide, 17
- Development environments
 - description, 25
- Development process
 - steps involved in developing an application, 33
- Dialogs, browse
 - see* Browse dialogs
- Dialogs, maintenance
 - see* Maintenance dialogs
- Dispatch service data
 - role on mainframe server, 30
- Dispatch services
 - options, 66
- Dispatcher
 - Selection dialog
 - See* Spectrum Dispatch Client, 322
- Dispatcher Selection dialog
 - customizing client framework components, 322
- Displaying
 - grids, 205
- Documentation
 - Construct Spectrum, 20
 - Construct Spectrum SDK, 20
 - Natural Construct, 21
 - related, 20
- Domains
 - setting up application environment, 36
 - specifying in super model, 116
- Downloading
 - browse modules to the client, 229
 - Download Generated Modules dialog, 131
 - Downloading Modules dialog, 152
 - generated components to project, 131
 - maintenance modules to the client, 151
- Drilling down from a browse dialog, 344
- Drop-down grids
 - demo project, 80

- Drop-down lists
 - active help from diagram, 348
 - foreign field support, 363
 - representing foreign fields, 356
 - dialog, 356

E

- Encrypting data
 - enabling
 - client to server transmissions, 145–146, 225
- Entire Broker
 - role on mainframe server, 30
- Error notifications
 - adding support for sound, 208
- ErrorPreferences.frm
 - description, 341
- ErrorTip.frm
 - description, 341
- EXE file
 - making for demo project, 58
- Existing applications
 - moving to Construct Spectrum, 245
- External data
 - accessing with the VB-Browse-Local-Data-Object model, 226
 - displaying in a generated combobox, 164
 - example code for accessing, 228

F

- Field help
 - active help, 345

- File number
 - specifying in a new project, 127
- FileExists
 - utility procedure, 324
- FindFirst
 - utility procedure, 324
- FixupRTF
 - utility procedure, 324
- FK-AS-COMBO-THRESH-HOLD
 - changing default value, 358
- FNR
 - description, 264
- FNR number
 - specifying in a new project, 127
- ForceLogonAtStartup
 - description, 263
- Foreign fields
 - active help, 346
 - diagram, 347
 - case for not using, 353
 - diagram, 353
 - corporate default threshold, 170
 - default GUI controls, 358
 - demo project, 84
 - displaying descriptions, 359
 - GUI controls used to represent, 355
 - multiple descriptive values, 360
 - refreshing descriptions, 361
 - repeating relationships, 352
 - diagrams of, 352
 - representing
 - grids, 357
 - representing in
 - drop-down lists, 356
 - lookup buttons, 356
 - representing in grids
 - diagram, 357
 - supported relationships, 351
 - diagram, 351
 - supporting code
 - command buttons, 364

drop-down lists, 363

G

G/R/O

in super model wizard, 111

Generated code

transferring to the project, 39

Generating

browse subprogram proxy, 222

individual models, 37

maintenance dialog, 147

maintenance subprogram proxy, 142

object factory

considerations for, 99

super model, 113, 119

diagram of, 95

new package, 109

overview, 94

packages and object factory, 106

specific packages, 109

super model wizard

Standard Parameters step, 105

Visual Basic browse object, 223

Visual Basic maintenance object, 142

Generation process

overview of server/client modules, 37

GetBrowser

TableName As BrowseManager, 306

GetBrowser procedure

example code, 316

updating, 315

GetPrivateProfileStringVB

utility procedure, 324

GetWindowsDirectoryVB

utility procedure, 324

Grids

column

adding to maintenance dialog, 189

demo project, 78

diagram

formatted grid, 205

unformatted grid, 205

displaying, 205

Grid Sizing Information dialog, 207

keyboard shortcuts, 81

representing foreign fields, 357

diagram, 357

resizing, 206

using, 202

GUI

generation standards

defining, 172

GUI controls

default controls for foreign fields, 358

default derivation logic, 165, 167

diagram, 168–169

derivation logic, 159

keywords, 162

naming conventions, 159

overriding default selection, 160

representing foreign fields, 355

GUI dialog

role on Windows platform, 32

GUI_ALPHA MULTILINE keyword

description, 162

GUI_CHECKBOX keyword

description, 162

GUI_COMBOBOX keyword

description, 162

GUI_NULL keyword

description, 163

GUI_OPTION BUTTON keyword

description, 163

GUI_PROTECTED keyword

description, 163

GUI_TEXTBOX keyword

description, 163

H

Help

See online help, 34

HKEY_CLASSES_ROOT

language registry, 378

HKEY_CURRENT_USER

language registry, 378

HKEY_LOCAL_MACHINE

language registry, 378

HKEY_USERS

language registry, 378

I

Individual models

when to use, 37

InitAppSettings procedure

example, 263

InitializeOpenDialog procedure

code example, 312

description, 306, 312

updating, 312

Integrating browse and maintenance dialogs, 343

design objectives, 349

drilling down from a browse

dialog, 344

overview, 344, 349, 351

see also Foreign fields, 343

Interface

demo project, 59

Internationalizing

generated applications, 367

hints for developers, 380

automatically setting the
language, 380

changing language at runtime, 383

using resource files and groups, 380

maintenance dialogs, 211

methods, 374

GetResourceGroup, 374

LoadBinaryResource, 374

LoadStringResource, 374

LocalizeForm, 374–375

Message, 374

MessageEx, 374

SetDefaultMessageGroup, 374

planning considerations, 368

list of translatable items, 368

properties, 377

Language, 377

LanguageINIKey, 379

LanguageRegistryKey, 378

ResourceFilePath, 379

related client framework components

Resource, 369

ResourceGroup, 369

using the client framework, 369

where to find related information, 367

Invoking

super model, 104, 114

IsForegroundApplication

utility procedure, 324

IsMDIChild

utility procedure, 324

K

Key field active help, 345

Keyboard shortcuts for grids, 81

Keywords

business data type, 166

GUI control, 162

verification rule, 164

L

- Label captions
 - GUI controls, 160
- Language sets
 - resource files, 370
- LanguageRegistryKey
 - description, 378
 - HKEY_CLASSES_ROOT, 378
 - HKEY_CURRENT_USER, 378
 - HKEY_LOCAL_MACHINE, 378
 - HKEY_USERS, 378
- Library image files
 - role on Windows platform, 31
- LoadBinaryResource method
 - description, 375
- LoadStringResource
 - description, 376
- Logon dialog
 - description, 321
 - See also* Spectrum Dispatch Client, 321
- Lookup button
 - representing a foreign field, 356
 - diagram of, 356

M

- Maintaining a business object
 - demo project, 73
- Maintenance dialogs
 - abbreviated object description, 160
 - active help for, 345
 - adding new field by hand, 185
 - controlling default size, 173
 - customizing on the server, 159
 - integrating with browse dialogs, 343
 - see also* Integrating browse and maintenance dialogs
 - internationalizing, 211
 - model
 - using, 147
 - modules required for, 101
 - object identifier, 160
 - prerequisites for generating with individual models, 140
- Maintenance modules
 - relationships between, 138
 - to download to project, 151
- Maintenance object
 - see* Visual Basic maintenance object
- Maintenance object subprogram
 - generating, 140
- Maintenance subprogram proxy
 - generating, 142
- Max
 - utility procedure, 324
- MDI
 - See* multiple-document interface, 301
- MDIFrame.frm
 - description, 301
- Menu
 - bar
 - definition, 301
 - structure
 - See* menus and toolbars, 289
- Menus and toolbars
 - command handling
 - class summary, 272
 - coding, 276
 - defining, sending, and handling, 273
 - linking commands, 277
 - understanding, 271
 - unhooking commands, 288
 - user interface controls, 278

- customizing, 289
 - menu bar example, 294
 - menu editor window, 295
 - menu structure, 289
 - toolbar button example, 299
 - toolbar structure, 298
 - demo application, 60
 - support
 - UICmd, 271
 - UICommandConstants.bas, 271
 - UICommands, 271
 - Message method
 - description, 376
 - MessageEx method
 - description, 376
 - Methods
 - coding the UICommandTarget(), 242
 - internationalizing
 - See also* Internationalizing, 374
 - marking updated rows, 243
 - Min
 - utility procedure, 324
 - Modal browse dialog
 - example, 345
 - Models
 - deciding which to use, 37
 - dependencies between, 103
 - Modifying
 - Spectrum models, 385
 - example of generated code, 387
 - GUI controls with VB API, 391
 - how the VB API works, 389
 - parameter data area (PDA) used, 397
 - VB API, 388
 - components
 - See also* VB API
 - VB-Maint-Dialog model, 386
 - VB-Maint-Dialog model architecture, 386
 - Modules
 - custom-created, 305
 - deciding which to generate with super model, 100
 - naming conventions, 97
 - diagram, 98
 - to download to project, 132
 - uploading changes to the server, 155
 - Monitor resolution
 - effect on dialog size, 173
 - MoveFormSafely
 - utility procedure, 324
 - Multi-column layout
 - creating on dialog, 181
 - example, 181
 - Multilingual support
 - See* Internationalizing, 211
 - Multiple descriptive values for a foreign field, 360
 - Multiple Generation utility
 - using with super model, 113
 - Multiple-document interface
 - MDIFrame.frm, 301
 - Menu Bar, 301
 - Status Bar, 301
 - Toolbar, 301
 - understanding, 301
- ## N
- Naming conventions
 - GUI controls, 159
 - super model, 97
 - diagram, 98
 - Natural Construct
 - documentation, 21
 - Natural Construct applications
 - moving to Construct Spectrum, 18

- Natural subprogram
 - role on mainframe server, 29
 - Nested grids
 - demo project, 79
 - diagram of relationships, 203
 - drop-down
 - diagram, 204
 - using, 204
 - using, 202
 - Non-object based applications
 - moving to Construct Spectrum, 248
- O**
- Object browse subprogram, 222
 - description, 102
 - key PDA description, 102
 - restricted PDA description, 102
 - row PDA description, 102
 - Object browse subprogram proxy
 - description, 102
 - Object factory
 - considerations for generating, 99
 - customizing, 307
 - new business objects, 311
 - OFactory.bas, 302
 - code, 310
 - OFactory.bas window, 308
 - Open dialog
 - understanding, 303
 - Open.frm
 - definition, 302
 - OpenAction, 302
 - OpenObject, 302
 - OpenObjects, 302
 - procedures
 - BrowserExists(TableName) As Boolean, 306
 - CreateForm(formID) As Form, 306
 - GetBrowser(TableName) As BrowseManager, 306
 - InitializeOpenDialog(), 306
 - relationship diagram, 309
 - selecting to generate in super model, 116
 - selecting to generate in super model wizard, 107
 - understanding, 302, 304
 - using, 305
 - example, 306
 - Object maintenance subprogram
 - description, 101
 - see* Maintenance object subprogram, 140
 - Object maintenance subprogram proxy
 - description, 101
 - Object-based applications
 - moving to Construct Spectrum, 247
 - Object-Browse-Subp model
 - description, 102
 - ObjectError
 - description, 341
 - ObjectErrors
 - description, 341
 - Object-Maint-Subp model
 - description, 101
 - OFactory.bas
 - description, 129, 302
 - example, 308
 - OLE automation server
 - customizing client framework components, 256
 - Online help
 - context-sensitive, 34
 - providing in client/server applications, 34
 - task-oriented, 34
 - window-level, 34

- Open dialog
 - overview, 302
 - relationship diagram, 309
 - understanding, 303
 - Open.frm
 - definition, 302
 - example screen, 303
 - OpenAction
 - description, 302
 - Opening a business object
 - demo project, 68
 - OpenObject
 - description, 302
 - OpenObjects
 - description, 302
 - Option button threshold
 - corporate default, 170
 - Overflow conditions
 - correcting, 174, 179
 - correcting on dialog, 148
 - example, 179
 - working with overflow frames, 180
 - Overriding default GUI control selection, 160
- ## P
- Packages
 - generating with super model, 100
 - specifying parameters
 - general, 106
 - general parameters, 115
 - specific parameters, 109, 117
 - specifying prefix in super model wizard, 110
 - PadLeft
 - utility procedure, 324
 - PadRight
 - utility procedure, 325
 - Parameter data areas
 - generating for browse object subprogram, 222
 - generating for maintenance object subprogram, 140
 - Planning your application
 - consistent style, 35
 - content of windows, 35
 - deciding what to show users, 33
 - number and structure of windows, 34
 - planning code, 35
 - setting up your project, 39
 - simple window design, 34
 - translation issues, 35
 - Predict definitions
 - setting up application environment, 36
 - Predict Modify Verification panel
 - description, 165
 - Predict set up tasks
 - default GUI controls, 155
 - headers, 155
 - keywords, 155
 - Prerequisites
 - Construct Spectrum project, 126
 - demo application, 45
 - developing client/server applications, 16
 - super model, 97
 - Preserving
 - user exits, 155
 - Primary keys
 - active help for, 345
 - Product integration
 - Adabas, 24
 - Construct Spectrum, 24
 - DB2, 24
 - Entire Broker, 24
 - Natural, 24

Predict, 24
VSAM, 24
Projects
 opening the demo, 46
 see also Construct Spectrum
 project, 123
Prompt to Open New Project dialog
 description, 129

R

Regenerating existing modules
 using super model, 116
 using super model wizard, 107
RegistryKey
 description, 263
Relationships
 between maintenance modules, 138
RememberUserID
 description, 263
Remote dispatch service options
 demo project, 66
Removing field
 by hand from maintenance dialog, 202
Repeating field threshold
 corporate default, 170
Repeating foreign fields
 represented in a grid, 357
 diagram, 357
 supported relationships, 352
 diagram of, 352
Replacing existing modules
 using super model, 116
 using super model wizard, 107
ResizeForm
 utility procedure, 325
Resizing
 grids, 206

Resizing grids
 controls on dialog, 148
Resource files
 composition, 369
 creating links, 373
 filename example, 369
 path
 purpose, 371
 specifying binary values, 373
 specifying text values, 372
 how to include non-printing
 characters, 372
 syntax, 372
Resource groups
 identifiers (RGID), 369
 purpose, 369
Resource identifiers (RID)
 composition, 369
Resources
 linking, 371

S

Save New Visual Basic Project dialog
 description, 128
Scalar field
 adding to maintenance dialog, 185
SDC
 See Spectrum Dispatch Client, 319
SDCDialog.frm
 description, 319
 dialog, 321
Security
 considerations for a new
 application, 135
 setting up domains, steplibs, users,
 and groups, 36
Server
 customizing maintenance dialogs, 159

- Server modules
 - generation overview, 38
- Server-based applications
 - moving to Construct Spectrum, 245
- SetDefaultMessage method
 - description, 377
- Setting up
 - Construct Spectrum project, 123
 - Predict file definitions, 160
- SetUppercaseStyle
 - utility procedure, 325
- Shortcuts
 - keyboard shortcuts for grids, 81
- Simple foreign field relationships, 351
 - diagram, 351
- Sound
 - adding to error notifications, 208
 - support for error notifications, 208
- Sound support
 - overview, 209
- Spectrum administration
 - role on mainframe server, 30
- Spectrum Dispatch Client
 - client framework support, 319
 - error messages, 322
 - SDCDialog.frm, 319
 - example dialog, 321
 - SDCSupport.bas, 319
 - TraceOptions.frm, 319
 - example dialog, 320
- Spectrum Dispatch Client (SDC)
 - overview, 24
- Spectrum dispatch service
 - overview, 24
 - role on mainframe server, 30
- Spectrum security services
 - role in Construct Spectrum applications, 30
- Startup.bas
 - See* application settings, 262
- State-dependent layout
 - creating on dialog, 183
 - example, 184
- Status bar
 - definition, 301
 - demo application, 65
- Steplib chains
 - setting up application environment, 36
- Strategies for customizing maintenance dialogs, 155
- Sub Main procedure
 - customizing client framework components, 262
- Subprogram proxies
 - generating for a browse dialog, 222
 - generating for a maintenance dialog, 142
- Subprogram proxy
 - role on mainframe server, 29
- Subprogram-Proxy model
 - description, 101–102
- Super model
 - defaults, 97
 - defining general package parameters, 115
 - defining specific package parameters, 117
 - General Package Parameters panel, 115
 - generating application modules, 37
 - generating from wizard, 113
 - generating in batch, 113
 - generation function, 119
 - generation overview, 94
 - diagram of, 95
 - invoking, 104, 114
 - invoking the model wizard, 104
 - Package modules grid in wizard, 110
 - prerequisites, 97
 - regenerating existing modules, 107

- replacing existing modules, 107
- Standard Parameters panel, 114
- troubleshooting, 121
- using message numbers, 105
- when to use, 94
- which modules to generate, 100

Super model wizard

- New Package step, 109
- Packages and Object Factory step, 106

T

Tabbed layout

- creating on dialog, 182
- example, 183

Testing applications

- recommendations for testing new application, 134

TextBox GUI control

- adding to maintenance dialog, 186

Thresholds

- foreign field, 171
- option button, 171
- repeating field, 171

Toolbar

- buttons, 282
- customizing
 - See* menus and toolbars, 298
- definition, 301
- demo application, 62

TraceOptions.frm

- dialog, 320
- example, 320

Transferring

- generated code to the client, 39

TWIPS monitor values

- description, 173

U

UICmd

- definition, 271

UICommandConstants.bas

- defining commands, 295
- definition, 271

UICommands

- class, 273
- definition, 271

Uploading

- changes to the server, 155
- Uploading Modules dialog, 213

User exits

- preserving changes by uploading to the server, 155
- user-defined for maintenance dialog, 157

User Exits panel

- VB-Browse-Object model, 226

User type rules

- coding, 337
- example of code for Natural rule, 337
- example of code for Visual Basic rule, 337
- example of code using Visual Basic and Natural, 338

Utility procedures

- AppendSlash, 323
- ArrayDimensions, 323
- ASSERT, 323
- CenterForm, 323
- CreateArray, 323
- CreateStringArray, 323
- CSTFormatMessage, 323
- CSTSelectContents, 324
- CSTSubst, 324
- CSTUtils.bas, 323
- description, 323
- FileExists, 324

- FindFirst, 324
- FixupRTF, 324
- GetPrivateProfileStringVB, 324
- GetWindowsDirectoryVB, 324
- IsForegroundApplication, 324
- IsMDIChild, 324
- Max, 324
- Min, 324
- MoveFormSafely, 324
- PadLeft, 324
- PadRight, 325
- ResizeForm, 325
- SetUppercaseStyle, 325

V

Validating data

- creating Predict verification rules, 336
- diagram of a validation cycle, 329
- examples in demo project, 74
- in maintenance dialogs, 333
 - hand-coding in generated dialogs, 333
 - using BDTs, 333
- in Visual Basic maintenance objects, 334
 - using CLIENT-VALIDATION user exit, 334
 - using Predict, 335
- on the client
 - diagram of triggering validation, 332
- order of precedence, 339
- typed of validations, 328
- types of data validation
 - business data type, 328
 - business object, 329
 - local business, 328

Validation error handling

- ErrorPreferences.frm, 341
- ErrorTip.frm, 341
- ObjectError, 341
- ObjectErrors, 341

Validation errors

- in business object validations, 341
 - example of code, 342
- on the client, 340

Variable names

- deriving, 159

VB API

- components, 388
 - LDA storing Visual Basic default values, 388
 - PDA's for GUI control definitions, 388
 - subprogram to assign default values, 388
 - subprogram to write GUI definition, 388
- description, 388
- GUI controls, 391
 - 3DCheckBox, 391
 - 3Dcommand Button, 392
 - 3DFrame, 393
 - 3DOptionButton, 394
 - 3DPanel, 395
 - CheckBox, 391
 - ComboBox, 391
 - CommandButton, 392
 - Form, 392
 - Frame, 392
 - Label, 393
 - Listbox, 393
 - MDIForm, 393
 - Menu, 394
 - OptionButton, 394
 - StatusBar, 394
 - TextBox, 395
 - Timer, 395
 - Toolbar, 396

- TrueDBGrid, 395
- PDA_s
 - CSASTD, 391
 - CSVA3CMD, 392
 - CSVA3DI, 393
 - CSVA3DPN, 395
 - CSVABUTN, 392
 - CSVACMBO, 391
 - CSVACOMN, 391
 - CSVADDE, 392
 - CSVAFOCS, 391
 - CSVAFONT, 391
 - CSVAFRMT, 391
 - CSVAGRID, 395
 - CSVALABL, 393
 - CSVALCTN, 391
 - CSVAMENU, 394
 - CSVAMOUS, 391
 - CSVASTAT, 394
 - CSVATBOX, 391
 - CSVATIME, 395
 - CSVATOGL, 391
 - CSVATOOL, 396
 - CSVAWNDW, 392
 - CUMDATYP, 391
- using with a custom model, 389
 - example of code in user default subprogram, 389
 - example of code to assign value to Caption property, 389
 - example of code using default values, 390
- VB-Browse-Local-Data-Object model
 - accessing alternate data sources with, 226, 228
- VB-Browse-Local-Data-Object model wizard
 - Standard Parameters step, 227
- VB-Browse-Object model
 - description, 102
- VB-Maint-Dialog model
 - description, 101
- VB-Maint-Object model
 - description, 101
- Verification rules
 - keywords, 164
 - Predict, 336
 - where to implement, 336
 - coding user type rules, 337
- Visual Basic browse
 - business object, 305
- Visual Basic browse object
 - adding support, 315–316
 - description, 102
 - generating, 223
- Visual Basic business objects
 - role on Windows platform, 32
- Visual Basic maintenance
 - business object, 305
 - object
 - generating, 142
- Visual Basic maintenance object
 - description, 101

W

- Windows platform
 - role of Entire Broker, 31
- Working environment
 - Construct Spectrum, 24

