

IBM VisualAge for C++ for Windows

S33H-5034-00

Visual Builder User's Guide

Version 3.5

IBM

IBM VisualAge for C++ for Windows

S33H-5034-00

Visual Builder User's Guide

Version 3.5

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xi.

First Edition (February 1996)

This edition applies to Version 3.5 of IBM VisualAge for C++ for Windows (33H4979, 33H4980) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

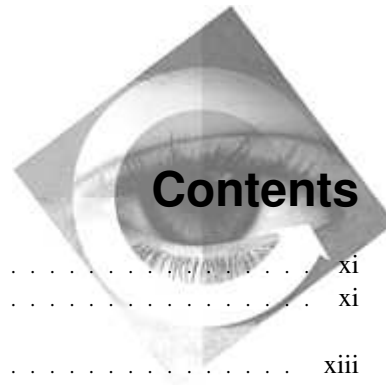
A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See “Communicating Your Comments to IBM” for a description of the methods. This page immediately precedes the Readers’ Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1992, 1996. All rights reserved. Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.



Notices	xi
Trademarks	xi
 About This Book	 xiii
Who Should Use this Book	xiii
How to Use this Book	xiii
How This Book Is Organized	xiv
Highlighting Conventions	xv
How to Get Help	xv
Getting Help Inside VisualAge for C++	xvi
Getting Help from the Command Line	xvi
Getting Help for a Keyword or Construct	xvii
Online Documents Available in VisualAge for C++	xvii
 Part 1. Getting Started	 1
 Chapter 1. What Is Visual Builder?	 3
What Are the Benefits of Using Visual Builder?	3
What Are the Key Concepts?	4
 Chapter 2. Creating a Simple Visual Builder Application	 7
Creating the To-Do List Application	7
Starting Visual Builder for the To-Do List Application	8
Creating a New Visual Part for the To-Do List Application	8
Placing Parts in the Application Window	10
Resizing and Aligning the Parts	12
Connecting the Parts	16
Generating the C++ Code for Your Application	20
Building the Application	21
Running the Application	21
Exiting the Composition Editor and Visual Builder	22
 Chapter 3. Starting Visual Builder	 23
Starting Visual Builder from a Command Prompt	23
Starting Visual Builder from the Tools Folder	24
Starting Visual Builder from a WorkFrame Project	24
 Part 2. Touring Visual Builder	 27

Chapter 4. Getting Acquainted with the Visual Builder Window	29
Getting to Know the Visual Builder Window	29
Working with the Files for Storing Parts, .vbb Files	31
Loading Part Files	31
Unloading Part Files	32
Selecting All Part Files	33
Deselecting All Part Files	33
Customizing the Information Area	34
Seeing the Base Files	34
Seeing Where Part Files Are Located	34
Seeing the Type List	35
Using File Allocation Table (FAT) File Names	36
Setting Make File Generation	36
Setting the Working Directory	36
Refreshing the Display	37
 Chapter 5. Getting to Know the Visual Builder Editors	 39
The Editor Symbols	39
The Composition Editor	40
The Tool Bar	41
The Parts Palette	44
The Free-Form Surface	48
The Class Editor	49
Entering a Description of a Part	50
Moving a Part to a Different Part File	51
Seeing the Base Class of a Part	51
Modifying a Part's Constructor	51
Specifying Your Own Constructor Code	52
Specifying Your Own Destructor Code	52
Specifying a Library File	52
Specifying a Starting Resource ID	53
Specifying a Unique Icon for Your Part	53
Specifying the Names of Your Code Generation Files	54
Specifying Files to Include When You Build Your Application	55
The Part Interface Editor	56
The Attribute Page	57
The Event Page	63
The Action Page	69
The Promote Page	73
The Preferred Page	75

Part 3. Developing Visual Builder Applications	79
---	----

Chapter 6. Designing Effective Applications	83
Designing Parts for a Single Purpose	83
Taking the Object-Oriented Approach	84
Identifying Reusable Parts	85
Using Inheritance	86
Using Abstract Classes	88
Keeping Your Components Small	89
Designing Parts for the OASearch Application	90
Using a Simulated Database	90
Designing Nonvisual Parts	90
Designing Visual Parts	94
How the Sample Application Was Built	99
Chapter 7. Creating Nonvisual Parts	101
Defining the Part Interface	101
Defining the Part Interface Using the Part Interface Editor	102
Adding Code to Your Part	105
Generating Feature Code	106
Modifying the Generated Feature Code	109
Modifying Code for the getContractor Action	109
Modifying Code for the putContractor Action	110
Modifying Code for the parseName Action	110
Modifying Code for the refreshID Action	110
Adding Code to Set the activeStatus Attribute	110
Adding Code to Set the contractorID Attribute	111
Adding Code Created Outside Visual Builder	112
Chapter 8. Learning to Use Parts	113
Working with Parts in the Visual Builder Window	113
Displaying Part Names	113
Selecting All Parts	114
Deselecting All Parts	114
Importing Part Information	114
Exporting Part Information	116
Creating a New Part	117
Opening Parts	119
Copying Parts from One Part File to Another	121
Moving Parts to a Different Part File	121
Deleting Parts from a Part File	123
Renaming Parts in Part Files	124
Working with Parts on the Free-Form Surface	124
Placing Parts on the Free-Form Surface	124
Guidelines for Placing Parts on the Free-Form Surface	127

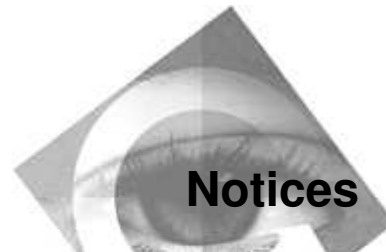
Selecting and Deselecting Parts	129
Manipulating Parts	131
Arranging Parts	134
Changing Settings for a Part	138
Using the Generic Settings Notebook	144
Listing Parts within a Composite Part	146
Changing Depth Order within a Composite Part	146
Performing Other Operations on Parts in the Parts List Window	148
Setting the Tabbing Order	148
Editing Parts Placed on the Free-Form Surface	151
Promoting a Part's Features	153
Tearing Off an Attribute	156
Undoing and Redoing Changes in the Composition Editor	157
Constructing a GUI: the OASearch Application	157
Adding Basic Visual Parts	158
Adding a Variable to a Composite Part	160
 Chapter 9. Learning to Use Connections	 163
Connection Type Summary	170
Making the Connections	171
Determining the Source and Target	171
Browsing a Part's Features	172
Connecting Features to Features	174
Supplying Parameter Values for Incomplete Connections	175
Connecting Features to Member Function Connections	179
Adding an Event-to-Member Function Connection	180
Using Browser Information	182
Connecting Features to Custom Logic	184
Adding a Custom Logic Connection	184
Connecting Exception Events to Actions and Member Functions	188
Manipulating Connections	189
Changing Settings for a Connection	189
Reordering Connections	194
Deleting Connections	195
Showing and Hiding Connections	195
Rearranging Connections	196
Selecting Connections	197
Deselecting Connections	198
Changing the Source and Target of Connections	198
Making Connections for the OASearch Application	199
Connecting the Variable Part	199
Enabling Push Buttons When an Entry Field Contains a Value	200
Passing Exceptions to Message Boxes	203

Enabling a Window to Be Cleared of All Entry Values	204
Chapter 10. Displaying Objects in a List Box	207
Copying the ToDoList Part	208
Creating the ToDoItem Nonvisual Part	208
Replacing and Modifying the List Box	211
Placing and Modifying an Object Factory Part	212
Placing and Modifying an IVSequence* Part	212
Making the New Connections	213
Generating the Source Code for Your Visual Part and main() Procedure	214
Building and Running the Modified Application	215
Chapter 11. Creating Resizable Windows	217
What Are the Benefits of Using Multicell Canvases?	217
Adding a Multicell Canvas	219
Adding Parts to the Multicell Canvas	219
Changing the Multicell Canvas Grid	222
Adding Rows or Columns Using the Contextual Menu	222
Deleting Rows or Columns Using the Contextual Menu	223
Extending a Part to Span More than One Cell	223
Adding a Group Box	225
Changing the Settings for a Multicell Canvas	227
Moving Dropped Parts	227
Changing Default Minimum Size for Rows or Columns	228
Adding Rows or Columns Using the Settings Notebook	229
Deleting Rows or Columns Using the Settings Notebook	229
Making Rows or Columns Expandable	229
Chapter 12. Constructing Containers and Notebooks	231
Adding Container Parts	231
Setting Up the Container	232
Adding Container Columns	234
Filling the Container	236
Clearing the Container	237
Adding Notebook Parts	237
Adding the Notebook	239
Adding Notebook Pages	240
Chapter 13. Adding Menus to Your Application	243
Types of Menus and Menu Items	243
Adding a Menu Bar	244
Connecting the Menu Bar to the Window	245
Adding Menu Choices	246

Adding Menu Separators	246
Connecting Menu Choices to Actions	247
Chapter 14. Adding Help to Visual Builder Applications	249
Creating the Help File	250
Providing Context-Sensitive Help	252
Providing General Help	253
Providing the Application Help Window	254
Providing Help for Factory-Generated Frame Windows	256
Providing a Help Push Button	257
Displaying Fly-Over Help When the Mouse Pointer Is Over a Part	258
Displaying Help in an Information Area	259
Adding an Information Area to a Frame Window	260
Displaying Help for Menu Choices in an Information Area	260
Displaying Long Fly-Over Text In an Information Area	260
Displaying Information about Successful Actions	261
Chapter 15. Integrating Visual Parts into a Single Application	263
Adding Nonvisual Support Parts to the Primary Part	263
Adding Static Parts	264
Adding Visual Parts as Dynamic Instances	264
Adding and Setting Object Factory Parts	265
Adding Variable Parts	266
Connecting to the Object Factory Parts	266
Connecting the Nonvisual Parts to the Variables that Represent Them	267
Connecting the Object Factory Parts to Their Corresponding Variable Parts	267
Completing the Menu Bar	269
Resetting Your Application's Main Resource Library	270
Chapter 16. Generating Source Code for Parts and Applications	273
Preparing for Source Code Generation	273
Setting Up Visual Builder to Generate Make Files	273
Setting Up Visual Builder to Generate HTML Documentation	274
Generating C++ Source Code for Individual Parts	274
Source Files Created during Part Code Generation	275
Generating Source Code for Your Application's main() Function	276
Source Files Created during Generation of main() Function Code	276
Preparing Generated Files for Compilation	277
Specifying Additional Libraries in the Make File	278
Specifying the Option to Generate Browser Information	278
Specifying Debug Options for the Compiler and Linker Programs	278
Packaging Runtime DLLs with Your Application	279
Compiling and Linking Your Application	279

Chapter 17. Sharing Parts with Others	283
Providing part files (.vbb)	283
Providing Part Information Files (.vbe)	284
<hr/>	
Part 4. Extending Visual Builder Applications	285
Chapter 18. Using Existing C and C++ Code with Visual Builder	287
Defining the Part Interface Using Part Information Files	287
Creating a Part Information File	287
Importing the Part	289
Creating Composers and Primitive Visual Parts	290
Chapter 19. Adding Categories and Parts to the Parts Palette	293
Preparing Icons for the Parts Palette	293
Preparing a Resource DLL (OS/2 Version Only)	294
Preparing a Resource DLL (Windows Version Only)	295
Adding a Category to the Parts Palette	296
Specifying a Unique Icon for a Part You Add to the Parts Palette	298
Adding a Part to the Parts Palette	299
Adding a Part That Is Selected in the Visual Builder Window	299
Adding the Part That You are Currently Editing	300
Adding Any Part Whose .vbb File Is Loaded	301
Deleting a Category or Part from the Parts Palette	302
Saving Parts Palette Changes	302
Chapter 20. Enabling National Language Support	303
Using Resource Files for Translation	303
The Resource File (.rci)	303
The Resource Header File (.h)	304
Guidelines for Specifying Starting Resource IDs	305
Using Canvases to Adjust Size for Translated Text	305
Specifying Parts with Country-Sensitive Formatting	306
Providing Double-Byte Character Support for Asian Languages	306
Chapter 21. Using Direct-to-SOM (DTS) Objects	309
Creating and Importing the Part Information File	310
Using DTS Objects in a Visual Builder Application	312
Bypassing DTS Limitations	313
Chapter 22. Hints and Tips for Using Visual Builder	315
Glossary	317

Bibliography	325
The IBM VisualAge for C++ Library	325
C and C++ Related Publications	325
Non-IBM Publications	325
Related Information	325
OO Programming and Design	325
User Interface	325
Index	327



Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights can be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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

IBM Director of Licensing,
IBM Corporation,
500 Columbus Avenue,
Thornwood, NY 10594,
USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

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

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries:

AIX
BookManager
Common User Access
CUA
IBM

IBMLink
Library Reader
OS/2
OS/2 Warp
PROFS
QuickBrowse
SAA
System Object Model
VisualAge
WorkFrame
Workplace Shell

Windows is a trademark of Microsoft Corporation.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks of others.

IBM's VisualAge products and services are not associated with or sponsored by Visual Edge Software, Ltd.



About This Book

Welcome to Visual Builder—the quickest and easiest way to create applications using the C++ programming language! This book, *Visual Builder User's Guide*, introduces parts, tools, and features that you can use to build Visual Builder applications.

Visual Builder is a tool provided by VisualAge for C++. It is based on the *construction-from-parts paradigm*, a software development paradigm in which applications are assembled from reusable and existing software components, called *parts*. You can extend Visual Builder by adding your own reusable, custom parts and then using these parts in your applications as you need them.

Visual Builder gets you started by providing a set of parts as well as interactive visual programming tools to work with those parts. You create your applications by visually assembling and connecting these prefabricated parts. In many cases, you do not even have to write any code.

Who Should Use this Book

Programmers who want to develop C++ applications using Visual Builder should read this book. Knowledge of object-oriented (OO) concepts, although not required unless you are interested in extending the Visual Builder parts to include your own parts, is highly recommended. This product incorporates OO concepts and knowledge of them will allow you to get the maximum use from the product, as well as an understanding of the terminology used in this book.

How to Use this Book

If you are new to Visual Builder, read through the first section completely. If you have used Visual Builder before, you can skim that section.



You will find shortcut techniques and other tips wherever you see .

This book refers to sample code that might have been modified after this book went to press. For the most current information, refer to the sample files used in this book.

How This Book Is Organized

This book is grouped into the following parts and appendixes:

- Part 1, “Getting Started,”

The first part introduces you to Visual Builder and its concepts. It also shows you how to start Visual Builder, walks you through your first application, and tells you how to set up a WorkFrame project to use Visual Builder.

- Part 2, “Touring Visual Builder”

This part shows you how to work with .vbb files, the files you store your parts in, and provides an overview of the Visual Builder editors.

- Part 3, “Developing Visual Builder Applications”

In this part, you travel through various aspects of application development and see how you can begin developing your own Visual Builder applications. You also learn how to use the Visual Builder window to work with parts.

- Part 4, “Extending Visual Builder Applications”

This part provides information that pertains to special aspects of application development that may not apply to all applications.

- Glossary and Reader’s Comment Form

The book ends with a glossary and information on sending us your comments and questions about Visual Builder.

Highlighting Conventions

This book uses the following highlighting conventions:

Bold : Key interface items in code listings; areas in code examples that are described in accompanying text. Example: Select **Tools** from the menu bar.

Monospace : C++ coding examples; text that the user enters; messages within text. Examples follow:

The following code from the `IAAddress` class illustrates ...

The `street` member function returns the current street.

Italics : Emphasis of words; the first time a glossary term is used; titles of books. Examples follow:

... stored in *persistent objects* ...

Refer to *Object-Oriented User Interface Design – IBM Common User Access Guidelines*.

How to Get Help

There are three kinds of online information available to you while you are using VisualAge for C++:

Online documents

These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge for C++. For your convenience, the online documents are presented in:

- Standard format (.INF files). See “Getting Help Inside VisualAge for C++” on page xvi for instructions on opening standard format documents from inside VisualAge for C++. See “Getting Help from the Command Line” on page xvi for instructions on opening standard format documents from the command line. For a list of the VisualAge for C++ documents that are available in standard format, see “Online Documents Available in VisualAge for C++” on page xvii.

Contextual help

Contextual help is available throughout VisualAge for C++. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I help

Many of the common tasks that you want to perform with VisualAge for C++ are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge for C++, as well as individual task lists for each of its components.

Getting Help Inside VisualAge for C++

All three kinds of help are available directly within the VisualAge for C++ interface:

- To get general contextual help for the component of VisualAge for C++ that you are using, press F1 anywhere in the window.
- To get contextual help on a particular menu, menu item, or button, highlight the element and press F1.
- To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:
 - **Help Index**, an alphabetical list of all of the help topics that are available from this window
 - **General Help**, overall help for the window
 - **Using Help**, general information about the help facility
 - **How Do I...**, the How Do I help for the component
 - **Product Information**, a dialog that shows the level of VisualAge for C++ being used

In addition, there are selections that let you open all of online documents that are available in VisualAge for C++.

- To get detailed information, open the **Online Information** notebook in the VisualAge for C++ folder. In this notebook you will find tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++. To open a particular online document, select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the `iview` command. The installation routine stores the online document files in the `\IBMCPW\HELP` directory. To view the Language Reference, for example, make `C:\IBMCPW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for `C:`) and enter the following command:

```
IVIEW CPPLNG.INF
```

If you want to get information on a specific topic, you can specify a word or a series of words after the file name. If the words appear in an entry in the table of contents or the index, the online document is opened to the associated section. For example, if you want to read the section on operator precedence in the Language Reference, you can enter the following command:

```
IVIEW CPPLNG.INF OPERATOR PRECEDENCE
```

Getting Help for a Keyword or Construct

If you are editing a file using the Editor, you can get help for a keyword or construct by moving the cursor to the word and pressing Ctrl+H. In the other tools, you can get help for a keyword or construct by highlighting the word and pressing Ctrl+H.

Online Documents Available in VisualAge for C++

The following documents are available in standard format:

Building VisualAge for C++ Parts for Fun and Profit	Open Class Library Reference
C Library Reference	Open Class Library User's Guide
Editor Command Reference	Programming Guide
Frequently Asked Questions	SOM Programming Guide
Installation Guide & Product Overview	SOM Programming Reference
IPF User's Guide	User's Guide
IPF Guide and Reference	Visual Builder User's Guide
Language Reference	Visual Builder Parts Reference

Part 1. Getting Started

This part describes Visual Builder, its benefits, and its key concepts. It also shows you how to start Visual Builder, walks you through your first application, and tells you how to set up a WorkFrame project to use Visual Builder.

Chapter 1. What Is Visual Builder?	3
What Are the Benefits of Using Visual Builder?	3
What Are the Key Concepts?	4
 Chapter 2. Creating a Simple Visual Builder Application	7
Creating the To-Do List Application	7
 Chapter 3. Starting Visual Builder	23
Starting Visual Builder from a Command Prompt	23
Starting Visual Builder from the Tools Folder	24
Starting Visual Builder from a WorkFrame Project	24

Getting Started



Chapter 1. What Is Visual Builder?

Visual Builder is a visual programming tool that can help you create object-oriented (OO) programs using the C++ programming language. With Visual Builder, you can create applications faster and easier than you ever could using a text editor. Visual Builder provides a powerful visual editor, the Composition Editor, which enables you to create complete applications, often without writing code.

What Are the Benefits of Using Visual Builder?

With Visual Builder, you can quickly create applications with advanced graphical user interfaces (GUIs). You can use Visual Builder to build OO applications by assembling and connecting parts.

Visual Builder is part of VisualAge for C++, a state-of-the-art C++ application development product that includes the IBM Open Class Library, WorkFrame, code browsing and editing tools, and a sophisticated debugger.

With Visual Builder, you can adopt OO technology immediately and learn to use it at the pace that is best for you. Its benefits include the following:

- When you work with Visual Builder's visual programming tools, you are creating OO applications.
- You can use Visual Builder to enhance and extend your applications because Visual Builder supports C++, an OO programming language.
- You can also access other logic written in C and C++ using Visual Builder's support for external C and C++ program logic.
- You can shorten your application development cycle time considerably by creating reusable parts.
- You can change the way a part does its work without affecting its external interface by encapsulating your application into parts. Encapsulating your application into parts also helps you deploy your business logic where it needs to be. Critical calculations can be moved into a dynamic link library.
- By its nature, Visual Builder caters to all skill levels, so that programmers using Visual Builder can build not only simple applications but also complex ones.

Getting Started

What Are the Key Concepts?

Before continuing, we will look at some key Visual Builder concepts.

Parts

In Visual Builder, a *part* is a C++ class that, in addition to the usual elements of a C++ class, includes a well-defined *part interface*. The part interface defines how the part can interact with other parts.

Three kinds of *features* make up a part interface. The following list provides a brief description of each kind:

- | | |
|-------------------|--|
| attributes | The logical data, often stored in data members, that other parts can access. This data can represent any logical property of a part, such as the balance of an account, the size of a shipment, or the text of a push button. |
| actions | Services or operations that a part can perform. Actions, such as placing an order or displaying a window, can be triggered by connections from other parts. |
| events | Signals that a part can send to notify itself or other parts that a change has occurred. When events are connected to attributes, actions, or member functions of other parts, the connections monitor these events and trigger the target features when the events occur. For example, when a push button's <i>buttonClickEvent</i> feature is connected to an action of another part, the other part's action is called when the push button is clicked. |

To get you started, Visual Builder provides a set of base parts for you. These parts are included in the `vbbase.vbb` file, which Visual Builder loads each time it is started. The parts in this file are based on the classes in the IBM Open Class Library. Visual Builder does not allow you to modify these parts, but you can create parts of your own by subclassing or using them in parts that you create.

Connections

To define how the parts interact with each other, you can make the following kinds of connections. The definitions that follow apply to most cases in which these connections are used. Exceptions and special cases are noted in Chapter 9, "Learning to Use Connections" on page 163.

- | | |
|-------------------------------|---|
| Attribute-to-attribute | Connections that link two data values together. When one changes, the other also changes. |
|-------------------------------|---|

Getting Started

Event-to-attribute	Connections that change the value of an attribute when a certain event occurs.
Event-to-action	Connections that start an action when a certain event occurs.
Attribute-to-action	Connections that start an action whenever an attribute's event identifier is signaled.
Event-to-member function	Connections that call a member function whenever a certain event occurs.
Attribute-to-member function	Connections that call a member function whenever an attribute's event identifier is signaled.
Custom logic	Connections that call your customized C or C++ logic whenever an event or an attribute's event identifier is signaled.
Parameter	Connections that provide a parameter value for an action or member function. The parameter value can be provided by connecting a parameter to an attribute, an action, a member function, or custom logic.

Source code generation

Visual Builder can generate C++ code for the GUI that you design in the Composition Editor, as well as for all of the connections that you make between parts. It can also generate C++ code for any new parts that you create. You can then use the code that Visual Builder generates when building your application. This capability allows you to concentrate on what your application does instead of spending time writing code for the GUI and its connections.

Besides saving you time and effort, additional advantages of letting Visual Builder generate your code instead of writing it yourself include the following:

- Easier code modifications.

Do you want to replace a multiline entry field with a list box? Delete the entry field, drop the list box in its place, make any necessary connections, and regenerate the code.

- Fewer errors.

Because Visual Builder can generate the majority of the code for your application, there is less opportunity for human errors, such as typographical and

Getting Started

syntax errors, to creep into your code. That means you spend less time debugging your code for minor errors.

- Support for pre-existing C and C++ code.

Using Visual Builder's Class Editor, you can specify files that contain existing C or C++ code that you want to use in your application. Then, when you generate the code for your application, those files are included.

In addition, you can create .vbe files that contain information about your C++ classes. You can then import that information into Visual Builder so that you can use those classes as parts. Refer to *Building VisualAge for C++ Parts for Fun and Profit* for information about using .vbe files.

- Standard format.

Another advantage is that a standard format is applied to all generated code. The code that Visual Builder generates is uniformly structured, indented, and commented.

Chapter 2. Creating a Simple Visual Builder Application

The best way to learn about Visual Builder is to see how you can use it. The following sections take you through an example of how you might use Visual Builder to develop an application for creating and maintaining a simple to-do list. Figure 1 shows what the application looks like when it is finished.



Figure 1. Finished To-Do List Application

Creating the To-Do List Application

Creating the To-Do List application consists of the following steps:

1. Starting Visual Builder for the To-Do List application
2. Creating a new visual part for the To-Do List application
3. Placing parts in the application window
4. Resizing and aligning the parts
5. Connecting the parts
6. Generating the C++ code for your application
7. Building the application
8. Running the application
9. Exiting the Composition Editor and Visual Builder

Getting Started

Starting Visual Builder for the To-Do List Application

Before you create the To-Do List application, start Visual Builder. For this sample application, start Visual Builder from the **Tools** folder, as follows:

1. Double-click on the **VisualAge for C++** folder icon on your desktop. The **VisualAge for C++** folder opens.
2. Double-click on the **Tools** folder icon. The **Tools** folder opens.
3. Double-click on the **Visual Builder** icon. Visual Builder displays the Visual Builder window, as shown in Figure 2.

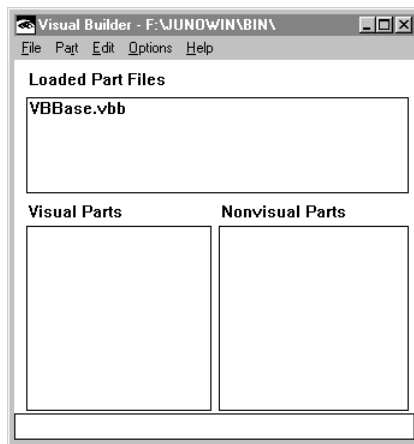


Figure 2. Visual Builder Window

Next, you can create a new visual part for the To-Do List application.

Creating a New Visual Part for the To-Do List Application

Before you create your new visual part for the To-Do List application, set the option for generating make files by selecting **Options**→**Generate make files**, if it is not already selected. Otherwise, when you finish the application and generate the code, Visual Builder will not generate the make file. By selecting this option now, you will not have to come back to the Visual Builder window to select it later.

The next thing to do when creating the To-Do List application is to create the main part, a new visual part, as follows:

1. Select **Part**→**New**. Visual Builder displays the Part-New window, as shown in Figure 3 on page 9.

Getting Started

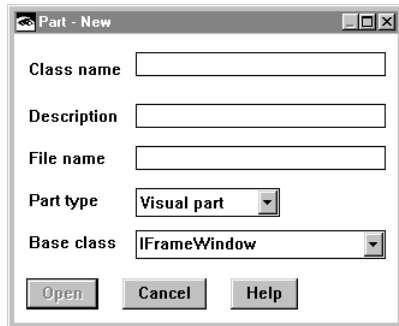


Figure 3. The Part–New Window

The Part–New window provides the following fields in which you can enter information about your part:

- The **Class name** field, where you enter the name of your part. Each composite part must have a name.

For the To-Do List application, enter the following:

ToDoList

- The **Description** field, where you enter a description of your part.

For this example, enter the following:

The To-Do List application

- The **File name** field, where you enter the name of the .vbb file in which you want Visual Builder to store your part.

For this example, you can either leave this field blank or enter the following:

todolist.vbb

This causes Visual Builder to store the ToDoList part in a file named todolist.vbb. If you leave this field blank, Visual Builder uses the part name as the name of the .vbb file by default, so the result is the same.

- The **Part type** field, where you indicate the type of part. This field initially contains Visual part. But you can specify a different type of part to create by selecting one from the field’s drop-down list box.

For this example, do not select a different part type because you want to create a visual part for the To-Do List application.

- The **Base class** field, where you specify the class that you want to be the base class for the part you are creating. The *base class* is the part from which your part inherits attributes, events, and actions. The **Base class** field

Getting Started

contains the default base class name of `IFrameWindow`, which Visual Builder uses when you specify that you want to create a new visual part.

For this example, leave `IFrameWindow` as the base class.

2. Select the **Open** push button to create a visual part named `ToDoList` whose parent is `IFrameWindow`. This causes Visual Builder to display the Composition Editor. For more information about the Composition Editor, see “The Composition Editor” on page 40.

If you look back at the Visual Builder window, you see that the file `todolist.vbb` is now included in the list of loaded files. This file was created for you when you selected the **Open** push button in the Part-New window.

Before you place any parts in the application window, first edit its title.

Changing the title of the To-Do List application window

The new visual part that you just created contains an `IFrameWindow*` part. This will be the To-Do List application window. Change the title of this window by doing the following:

1. Double-click on the part. The part settings notebook for the part opens.
2. Enter the new title text in the **Title text** field, such as `To-Do List`.
3. Click on **Apply**.


Placing Parts in the Application Window

After creating your new visual part and editing the title of the application window, you can place the other parts of the To-Do List application in the application window.


Placing a static text part in the window

The To-Do List application needs two static text parts. Follow these steps to place the first static text part in the To-Do List application window:



1. Select , the Data entry category, from the row of icons on the left-hand side of the parts palette.



2. Select , the `IStaticText*` icon, from the row of icons on the right-hand side of the parts palette. When you move the mouse pointer over the free-form surface, you see that it has changed to crosshairs. This means the mouse pointer is loaded with the `IStaticText*` part.


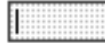
Getting Started

3. Place the crosshairs in the upper-left corner of the To-Do List application window's client area and click mouse button 1. A static text part is placed in the window.
4. Change the text of the static text part to To-do item. Use the same method for changing text that you learned previously when you changed the title of the To-Do List application window.

Placing an entry field in the window

The To-Do List application needs an entry field part. Follow these steps to place an entry field part in the To-Do List application window:



1. Select , the Data entry category, from the row of icons on the left-hand side of the parts palette.
2. Select , the **IEntryField*** icon, from the row of icons on the right-hand side of the parts palette.
3. Place the crosshairs beneath the first static text part and click mouse button 1. The entry field part is placed beneath the static text part.

Placing another static text part in the window


The To-Do List application needs another static text part. Follow these steps to place and modify this part:

1. Place the second static text part in the To-Do List application window. Use the same method for placing a static text part that you learned previously when you placed the first static text part in the To-Do List application window.
2. Change the text of the static text part to To-do list. Use the same method for changing text that you learned when you changed the title of the To-Do List application window.

Placing a list box in the window


Because the to-do list is to consist of a list of text strings, you want to store that list in an IListBox* part. Follow these steps to place a list box part in the To-Do List application window:



1. Select , the Lists category, from the row of icons on the left-hand side of the parts palette.

Getting Started





2. Select , the **IListBox*** icon, from the row of icons that Visual Builder displays on the right-hand side of the parts palette.
3. Place the crosshairs below the second static text part and click mouse button 1. The list box part is placed beneath the second static text part.

Placing the push buttons in the window

The To-Do List application needs two push buttons, one for adding items to the list and one for removing items from the list. Follow these steps to place two push button parts in the To-Do List application window:



1. Select , the Buttons category, from the row of icons on the left-hand side of the parts palette.
2. Select , the **IPushButton*** icon, from the row of icons on the right-hand side of the parts palette.
3. Place the crosshairs below the lower-left corner of the list box and click mouse button 1. The first push button part is placed in the window.
4. Select the **IPushButton*** icon again.
5. Place the crosshairs to the right of the first push button and click mouse button 1. The second push button is placed in the window.
6. Change the text of the first push button to Add. Use the same method you used to change the text in the title bar.
7. Change the text of the second push button to Remove.

Resizing and Aligning the Parts

Now that you have placed all of the parts you need in the application window, it is time to resize and align them. When you have finished, your application window should look like Figure 1 on page 7.

Matching the width of the list box to the width of the entry field

Follow these steps to match the width of the list box to the width of the entry field:

1. Move the mouse pointer over the list box.
2. In OS/2, press and hold mouse button 1. In Windows, select the list box. The selection handles appear at the four corners of the list box.

Getting Started

3. In OS/2, while holding down mouse button 1, move the mouse pointer to the entry field. In Windows, while holding down the Shift key, select the entry field. The selection handles on the list box become round, and black selection handles appear on the four corners of the entry field. This means that both parts are selected, but the entry field is the *anchor* part. Therefore, any sizing actions performed using the tool bar cause the list box to match the size of its anchor part, the entry field.



4. Select the Match Width tool, from the row of icons on the tool bar, located beneath the menu bar. The width of the list box changes to match that of the entry field.

Matching the width of the Add push button to that of the Remove push button

Using the techniques you learned in the preceding steps, match the width of the **Add** push button to that of the **Remove** push button.

Dragging and dropping parts in the application window

Before you align the parts, you might want to drag and drop some of them to put them in closer proximity to each other. For example, you might want the static text parts to be closer to the parts that they label. Follow these steps to drag and drop the parts in the application window:

Note: The following instructions are written for dragging and dropping multiple parts simultaneously. If you just want to drag and drop one part at a time, you can skip the first step.

1. Select all of the parts you want to drag using the technique that you learned previously when matching the width of the list box to the width of the entry field.
2. Move the mouse pointer over one of the parts that you selected to drag.
3. Press and hold mouse button 2 in OS/2 or mouse button 1 in Windows and move the mouse cursor. Visual Builder displays an outline of the parts that you are dragging.
4. Move the outline to the place where you want to drop the parts and release the mouse button. The parts are moved to their new location.

Resizing the application window

At this point, the parts in the application window are closer to the left window border than to the right window border. Follow these steps to resize the application window:

Getting Started

1. Select the application window by clicking mouse button 1 on the title bar.
2. Move the mouse pointer over the selection handle on the lower-right corner of the application window.
3. Press and hold mouse button 1.
4. Resize the application window by dragging the mouse pointer towards the left until the right border of the application window is approximately the same distance from the entry field and list box as the left border is.




To size the window in only one direction, either horizontally or vertically, hold down the Shift key while dragging the mouse pointer.

Centering the entry field and list box within the application window


The entry field and list box need to be centered within the application window. Follow these steps to center them:

1. Select the entry field.



2. Select , the Distribute Horizontally tool, from the row of icons on the tool bar. Visual Builder centers the entry field between the left and right borders of the application window.
3. Select the list box and then the entry field, making the entry field the anchor part. Use the multiple part selection technique you learned previously.



4. Select , the Align Center tool, from the row of icons on the tool bar. The list box is centered beneath the entry field.


Aligning the static text, entry field, and list box parts so their left edges are even

The static parts need to be aligned evenly with the left edges of the entry field and list box. Follow these steps to align them:

1. Select the first static text part and then select the entry field, making the entry field the anchor part. Use the multiple part selection technique you learned previously.

Getting Started




2. Select , the Align Left tool, from the row of icons on the tool bar. The first static text part is aligned even with the left edge of the entry field.
3. Repeat steps 1 and 2 for the second static text part and the list box. The second static text part is aligned even with the left edge of the list box.

Aligning the top edges of the push buttons

The push buttons need to be aligned so that their top edges are even. Follow these steps to align them:

1. Select the **Add** push button and then select the **Remove** push button. Use the multiple part selection technique you learned previously.



2. Select , the Align Top tool. Visual Builder aligns the **Add** push button even with the top of the **Remove** push button.

Centering the push buttons between the bottom edge of the list box and the bottom border of the application window

The push buttons need to be centered between the bottom edge of the list box and the bottom border of the application window. Follow these steps to center them:

1. Select both push buttons. You can make either push button the anchor part. Use the multiple part selection technique you learned previously.
2. Move the mouse pointer over either push button.
3. Press and hold mouse button 2 in OS/2 or mouse button 1 in Windows and position the push buttons midway between the bottom edge of the list box and the bottom window border.
4. When the push buttons are in place, release the mouse button.

Your application should now look like the one shown in Figure 1 on page 7.


Getting Started

Spacing the push buttons evenly across the application window

The push buttons need to be evenly spaced across the width of the application window. Follow these steps to space them:

1. Select both push buttons. You can make either push button the anchor part. Use the multiple part selection technique you learned previously.



2. Select , the Distribute Horizontally tool. Visual Builder spaces both push buttons evenly across the application window.

Connecting the Parts

You can now connect the parts so that your application can add items to and remove items from the to-do list. You need to connect the push buttons to the list box and entry field. The following steps show you how to do this.

Connecting the Add push button to the list box

The connection between the **Add** push button and the list box provides the information your application needs to add items to the list box.

1. With the mouse pointer over the **Add** push button and click mouse button 2. A pop-up menu is displayed.
2. Select **Connect**. A cascaded menu, called the *connection* menu, of the **Add** push button is displayed.
3. Select the *buttonClickEvent* feature. Selecting the *buttonClickEvent* feature means that you want something to happen whenever a user clicks this push button. The mouse pointer changes to look like a spider, indicating that it is ready for you to select another feature.
4. Move the mouse pointer to the list box and click mouse button 1. A pop-up menu is displayed showing the connection menu of the list box.
5. Select the *addAsLast* action. Selecting the *addAsLast* action means that you want new items to be added to the end of the to-do list whenever a user clicks the **Add** push button. The connection is shown in Figure 4 on page 17.

Getting Started

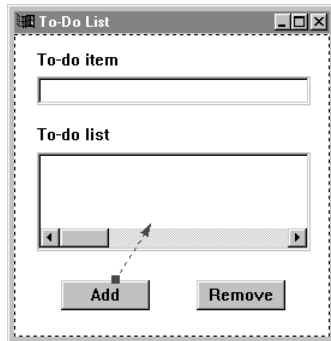


Figure 4. The Connection between the Add Push Button and the List Box

The line connecting the **Add** push button to the list box is dark green. It points from the push button to the list box, showing that the event that occurs when the push button is selected will cause the list box to perform an action.

Notice that the connection line is dashed instead of solid. A dashed line means that the connection is incomplete. The connection is supposed to add something to the list box when the **Add** push button is clicked, but you have not yet supplied what needs to be added. The next step does that.

6. Move the mouse pointer to the dashed connection line between the **Add** push button and the list box.
7. To take a shortcut to display connection menus, hold down the Alt key, click mouse button 2, and release the Alt key immediately afterward. The pop-up menu for the connection is displayed.
8. Select the *text* parameter. The *text* parameter is the reason the connection line is dashed. You need to give this parameter a value.
9. Move the mouse pointer over the entry field, and click mouse button 1. Visual Builder displays the connection menu for the list box.
10. Select the *text* attribute. Selecting the *text* attribute here means that you want to pass the text that a user enters in the entry field to the *text* parameter of the *addAsLast* action. This text string is added to the end of the to-do list whenever the *addAsLast* action is called, which occurs whenever the **Add** push button is clicked. The completed connection is shown in Figure 5 on page 18.

Getting Started

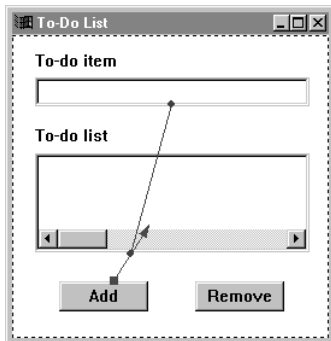


Figure 5. The Completed Connection for the Add Push Button

The line connecting the entry field to the connection between the **Add** push button and the list box is violet. The solid arrow head points to the entry field, showing that the *text* attribute is the target of the connection.

The round arrow head points to the connection line, indicating that the *text* parameter of the *addAsLast* action is the source of the connection. When the *text* parameter needs a value, which occurs when a user clicks on the **Add** push button, the connection invokes the get member function of the entry field's *text* attribute. The value of that attribute (the text in the entry field) is returned to the *text* parameter and the *addAsLast* action puts the text string in the list box.

Notice that both of the connection lines are solid. This means that the connection between the **Add** push button and the list box now has the information it needs to perform its function, so the connection is complete.

Connecting the Remove push button to the list box

The connection between the **Remove** push button and the list box provides the information your application needs to remove items from the list box.

1. With the mouse pointer over the **Remove** push button, hold down the Alt key and click mouse button 2. Visual Builder displays the connection menu for the **Remove** push button.
2. Select the *buttonClickEvent* feature.
3. Move the mouse pointer to the list box and click mouse button 1. Visual Builder displays the connection menu for the list box.
4. Select the *remove* action. Selecting the *remove* action means that you want your application to remove the selected item in the to-do list whenever a user clicks the **Remove** push button. Once again, the connection is incomplete.

Getting Started

5. Move the mouse pointer to the connection between the **Remove** push button and the list box.
6. Hold down the Alt key and click mouse button 2. Visual Builder displays the connection menu for the connection.
7. Select the *index* parameter. The *index* parameter is the reason the connection line is dashed. You need to give this parameter a value.
8. Move the mouse pointer over the list box and click mouse button 1. Visual Builder displays the connection menu for the list box.
9. Select the *selection* attribute. Selecting the *selection* attribute means that you want to pass the index of the selected item in the list box to the *index* parameter of the *remove* action. The *remove* action uses this index to determine which item to remove whenever the **Remove** push button is clicked.

Making this connection completes your application. It should now look like the one shown in Figure 6.

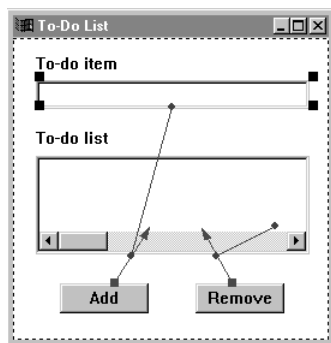


Figure 6. The Completed To-Do List Application

Note: In the preceding figure, we changed the shape of the connection between the **Remove** push button and the list box to make it easier for you to see. You can do this by selecting the connection and dragging the middle selection handle.

Now that you have made all of the connections, the next step is to generate your C++ source code.

Getting Started


Generating the C++ Code for Your Application

The first thing you must do to get your application ready to build is to generate the C++ code. This is a two-part process that consists of generating the source code for your new visual part and then generating the source code for your `main()` procedure.

Generating the source code for your visual part

To generate the C++ source code for your visual part, select **File→Save and generate→Part source**.



Another way to generate part code is to select , the Part Code Generation tool, on the tool bar. The results are the same.

Visual Builder generates the following files in the working directory:

todolist.cpp	The C++ code for your ToDoList part.
todolist.hpp	The C++ header file for your ToDoList part.
todolist.h	The resource header file for your todolist.cpp file.
todolist.rci	The resource file for your todolist.rcx file, which will be created when you generate source code for your <code>main()</code> function.

Generating the source code for your `main()` function

To generate the source code for your `main()` function, select **File→Save and generate→main() for part**. Visual Builder generates the following files in the working directory:

todolist.app	The main function for your application.
	Note: If you start Visual Builder from a WorkFrame project, the name of this file is <code>vbmain.cpp</code> .
todolist.mak	The make file that you specify when you build your application.
todolist.rcx	The main resource file. It includes a resource (.rci) file for each part that you generated part source for. In this example, there is only one, <code>todolist.rci</code> , because you only needed to generate part source for one part.

You have now generated the C++ code for your application. The next step is to build the application.

Building the Application

Building your application consists of compiling and linking it. To build your application, do the following:

1. Open a command prompt window.
2. Change to your Visual Builder working directory.
3. Enter the following command:

```
nmake todolist.mak
```

This command produces the following files:

todolist.exe The executable file for your application.

todolist.map The application configuration map.

todolist.o The object file for your application.

Note: If you start Visual Builder from a WorkFrame project, the name of this file is vbmain.obj.

todolist.rc The preprocessed resource file for your application. Visual Builder concatenates and places all resource files into this file.

todolist.obj The object file for your part. Visual Builder provides a separate object module for your part that is used when compiling this part with other parts.

todolist.res The binary resource file that is bound to todolist.exe.

You have now built your application; the next step is to run your application.

Running the Application

To run your application from the same command prompt from which you entered the nmake command, enter the following:

```
todolist
```

Once your application is running, experiment with it to make sure it works as you designed it.

You can add a finishing touch to your application by creating a program object in OS/2 or a shortcut in Windows. Once you have done this, you can run your application by simply double-clicking on the program object or shortcut you just created.

Getting Started

Exiting the Composition Editor and Visual Builder

To exit either the Composition Editor or Visual Builder, do either of the following.

- Select **File→Exit**.
- Double-click on the system menu icon in the window.

Note: You must exit Visual Builder before you can shut down the operating system. Otherwise, the operating system might not shut down completely, requiring you to turn the computer off.

When you exit Visual Builder, any changes you have made to the selections in the **Options** menu are saved. Therefore, if you want certain options to be selected or deselected the next time you start Visual Builder, be sure to select or deselect them before exiting Visual Builder.

If you try to exit Visual Builder while one or more editor windows is open, Visual Builder displays a message asking if you want to close the editors and Visual Builder. You can select either of the following:

- The **OK** push button to exit the windows
- The **Cancel** push button to cancel the exit request

If you select the **OK** push button with this message displayed or try to exit an editor and the open editor window contains unsaved changes, Visual Builder displays a message asking if you want to save the changes for each open editor before exiting. You can select either of the following:

- The **Yes** push button to save the changes and exit
- The **No** push button to exit without saving the changes
- The **Cancel** push button to cancel the exit request

Chapter 3. Starting Visual Builder

This chapter provides the following topics that show you the different ways you can start Visual Builder.

- From a command prompt
- From the tools folder
- From a WorkFrame project

Starting Visual Builder from a Command Prompt

To start Visual Builder from the C/C++ window in OS/2, do the following:

1. Double-click on the **VisualAge for C++** folder icon. The folder opens.
2. Double-click on the **C/C++ Window** icon. The C/C++ window opens.
3. Enter the following:
`ivb`
4. Press the Enter key. Visual Builder displays the Visual Builder window, as shown in Figure 7.

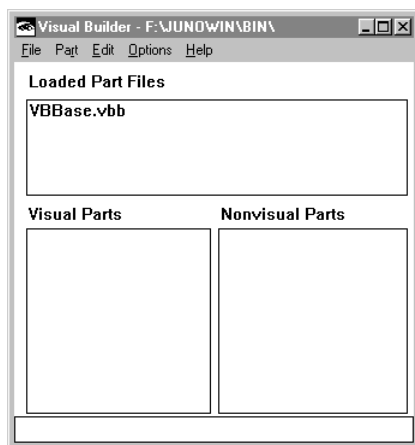


Figure 7. Visual Builder Window

Getting Started

To start Visual Builder from an MS/DOS command window in Windows, do the following:

1. Open an MS/DOS command window.
2. Enter the following:

```
ivb
```
3. Press the Enter key. Visual Builder displays the Visual Builder window, as shown above.

Starting Visual Builder from the Tools Folder

To start Visual Builder from the **VisualAge C++ Tools** folder in OS/2, do the following:

1. Double-click on the **VisualAge for C++** folder icon. The folder opens.
2. Double-click on the **VisualAge C++ Tools** folder icon. The folder opens.
3. Double-click on the **Visual Builder** icon. Visual Builder displays the Visual Builder window, as shown in Figure 7 on page 23.

To start Visual Builder from the **VisualAge C++** folder in Windows 95, do the following:

1. Select **IBM VisualAge C++ for Windows→Visual Builder** from the **Start→Programs** choice on the toolbar. The folder opens.
2. Double-click on the **Visual Builder** icon. Visual Builder displays the Visual Builder window, as shown in Figure 7 on page 23.

To start Visual Builder from the **VisualAge C++** folder in Windows NT, do the following:

1. Double-click on the **IBM VisualAge C++** folder on the desktop. The folder opens.
2. Double-click on the **Visual Builder** icon. Visual Builder displays the Visual Builder window, as shown in Figure 7 on page 23.

Starting Visual Builder from a WorkFrame Project

In OS/2

You can start Visual Builder from a WorkFrame project folder in the following ways.

Note: The following steps assume that your project inherits the settings of a VisualAge C++ project or that it was created using a Project Smarts template.

Getting Started

1. To start Visual Builder and also load and open an existing .vbb file that you want to work with, open the WorkFrame project folder and do either of the following:
 - Double-click on the name of the .vbb file.
 - Click on the name of the .vbb file with mouse button 2 and select **Visual** from the pop-up menu.
2. To start Visual Builder without loading and opening an existing .vbb file, do either of the following:
 - If the WorkFrame project folder is closed, you can click on the folder with mouse button 2 and select **Visual** from the pop-up menu.
 - If the WorkFrame project folder is open, you can do one of the following:
 - Click on the white space in the project folder with mouse button 2 and select **Visual** from the pop-up menu.
 - Select **Project** on the menu bar and then select **Visual** in the pull-down menu.

Visual Builder displays the Visual Builder window, as shown in Figure 7 on page 23. If you double-clicked on a .vbb file to open Visual Builder, that file is loaded and opened.

When you start Visual Builder from a WorkFrame Project folder, the menu bar in the Visual Builder window and in each of the Visual Builder editor windows contains an additional **Project** menu bar choice. Selecting this choice displays a list of the WorkFrame actions that are currently at the project scope for the project with which you are working. Examples of actions you might see in this list are Debug, MakeMake, Build, Run, Database, and Browse.

In Windows

You can start Visual Builder from a WorkFrame project in the following ways.

1. To start Visual Builder and also load and open an existing .vbb file that you want to work with, open the WorkFrame project and do either of the following:
 - Double-click on the name of the .vbb file.
 - Click on the name of the .vbb file with mouse button 2 and select **Visual** from the pop-up menu.
2. To start Visual Builder without loading and opening an existing .vbb file, do either of the following:
 - Click on the white space in the project folder with mouse button 2 and select **Visual** from the pop-up menu.

Getting Started

- Select **Project** on the menu bar and then select **Visual** in the pull-down menu.

Visual Builder displays the Visual Builder window, as shown in Figure 7 on page 23. If you double-clicked on a .vbb file to open Visual Builder, that file is loaded and opened.

When you start Visual Builder from a WorkFrame Project, the menu bar in the Visual Builder window and in each of the Visual Builder editor windows contains an additional **Project** menu bar choice. Selecting this choice displays a list of the WorkFrame actions that are currently at the project scope for the project with which you are working. Examples of actions you might see in this list are Debug, MakeMake, Build normal, Run, Database, and Browse.

Part 2. Touring Visual Builder

This part begins by introducing you to the place where everything starts, Visual Builder's Visual Builder window. It then provides an overview of the Visual Builder editors: the Composition Editor, the Class Editor, and the Part Interface Editor.

Chapter 4. Getting Acquainted with the Visual Builder Window	29
Getting to Know the Visual Builder Window	29
Working with the Files for Storing Parts, .vbb Files	31
Customizing the Information Area	34
Seeing the Base Files	34
Seeing Where Part Files Are Located	34
Seeing the Type List	35
Using File Allocation Table (FAT) File Names	36
Setting Make File Generation	36
Setting the Working Directory	36
Refreshing the Display	37
Chapter 5. Getting to Know the Visual Builder Editors	39
The Editor Symbols	39
The Composition Editor	40
The Class Editor	49
The Part Interface Editor	56

Touring Visual Builder

Chapter 4. Getting Acquainted with the Visual Builder Window

- Getting to know the Visual Builder window
- Working with the files for storing parts, .vbb files
- Customizing the information area
- Seeing the base files
- Seeing where part files are located
- Seeing the type list
- Using File Allocation Table (FAT) file names
- Setting make file generation
- Setting the working directory
- Refreshing the display

Getting to Know the Visual Builder Window

The Visual Builder window is shown in Figure 8.

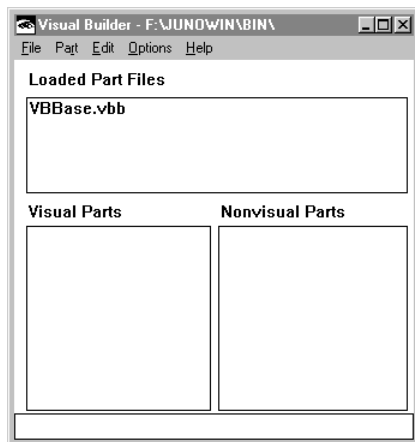


Figure 8. Visual Builder Window

This window contains the following areas:

- The **Loaded part files** list box

Parts that you create are stored in files with an extension of .vbb. These files are called *part files*. You can share part files that you create with other programmers so that they can reuse your parts.

Touring Visual Builder

This list box shows all of the part files that are currently loaded. Visual Builder provides the following part files:

vbbase.vbb

Contains the base parts that Visual Builder provides. This file is always loaded.

vbjmm.vbb

Contains multimedia parts.

vbsample.vbb

Contains miscellaneous sample parts.

Note: The part files that Visual Builder provides are read-only files. Store the parts that you create in your own part files.

Part files must be loaded into Visual Builder for you to use the parts that they contain. Once part files are loaded, you can perform actions on them and on the parts that they contain by using the choices in the Visual Builder's menu bar.

Part files created using Release 3.5 for Windows cannot be used with Release 3.0 for OS/2. You can use part files created using Release 3.0 in Release 3.5. However, once these part files are saved under Release 3.5, you can no longer use them in Release 3.0.



You can access the same choices that are available in the menu bar by moving the mouse pointer over a list box and pressing mouse button 2 to display a pop-up menu. Each pop-up menu contains only the menu choices that pertain to the contents of the list box over which it is displayed. The pop-up menu that Visual Builder displays for the **Loaded part files** list box, for example, contains only the menu choices that pertain to part files.

Also, the pop-up menu for the **Visual parts** list box only affects parts that are selected in that list box, even if parts are also selected in the **Nonvisual parts** list box, and vice versa. For example, to simultaneously open both a visual and a nonvisual part, you must select **Part→Open** from the menu bar. If you select **Open** from the pop-up menu for the **Visual parts** list box, you can only open a visual part. The same is true for the pop-up menu for the **Nonvisual parts** list box.

The choices on the **Part** menu apply to selected items in all of the list boxes in the Visual Builder window. This includes the **Loaded type information** list box, which you display by selecting **Options→Show type list**.

For more information about part files and the actions you can perform on them, see "Working with the Files for Storing Parts, .vbb Files" on page 31.

Touring Visual Builder

- The **Visual parts** list box

This list box displays the names of the visual parts that the selected part file contains. *Visual parts* are parts that the person using your application can see, such as frame windows, push buttons, and sliders.

- The **Nonvisual parts** list box

This list box displays the names of the nonvisual parts and the class interface parts that the selected part file contains. *Nonvisual parts* are parts that your application uses to perform its functions, but the person using your application never sees them. For example, an object factory, which creates new instances of objects, is a nonvisual part. The user sees only the objects that the object factory creates, not the object factory itself.

Class interface parts are nonvisual parts that have no notification ability. Thus, these parts cannot send events to other parts. You can use C++ classes that you have written as class interface parts in Visual Builder.

Working with the Files for Storing Parts, .vbb Files

The topics in this section describe how to perform various actions on part files from the Visual Builder window.

Loading Part Files

To give Visual Builder access to parts, you must load the contents of the part files that contain those parts by doing the following:

1. Select **File→Load** in the Visual Builder window.

Visual Builder displays the window shown in Figure 9.

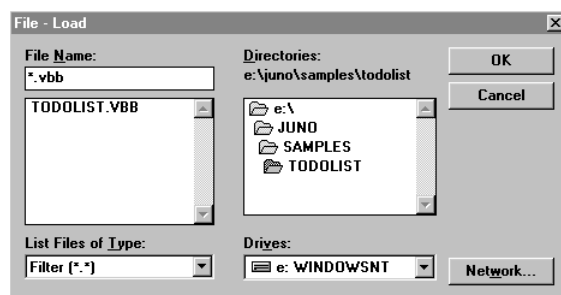


Figure 9. File-Load Window

2. Select the file or files that you want to load.
3. Select the **OK** push button.

Touring Visual Builder



When you are just loading one file, it is quicker to double-click on the file name instead of selecting the file name and the **OK** push button.

The file name or names are displayed in the **Loaded part files** list box in the Visual Builder window. Figure 10 shows the Visual Builder window with multiple part files loaded.

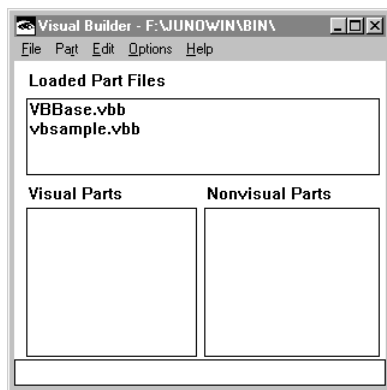


Figure 10. Visual Builder Window with Multiple Part Files Loaded

Unloading Part Files

If a part file appears in the **Loaded part files** list box in the Visual Builder window, Visual Builder has access to the parts that the part file contains. If you do not want Visual Builder to have access to those parts, you can unload the part file, with the exception of `vbbase.vbb`. To unload one or more part files, do the following.

Note: Close all Visual Builder editor windows before unloading part files. In some cases, Visual Builder does not refresh the internal data model to indicate that a part file has been unloaded.

1. Select one or more files in the **Loaded part files** list box.
 - To select multiple files in OS/2, hold down the Ctrl key while clicking on the file names with mouse button 1.
 - To select multiple files in Windows, hold down the Shift key while clicking on the file names with mouse button 1.

To select a block of parts in the list, hold down the Shift key while clicking on the topmost part. Then hold down the Shift key while clicking on the bottommost part. All parts listed between the two selected parts are also selected.

Touring Visual Builder

2. Select **File→Unload**.

The following window is displayed showing the files you selected to unload:

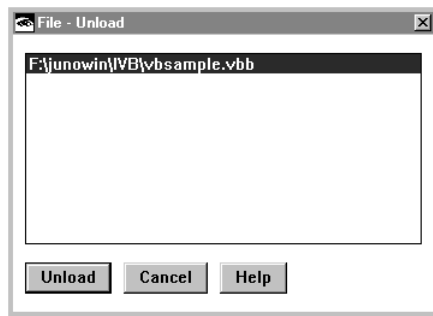


Figure 11. Unload Part Files Window

At this point, you can review the files that you selected and make any changes by deselecting any file or files that you want to remain loaded.

3. Select the **Unload** push button.

The window disappears and the file names are removed from the **Loaded part files** list box.

For information about loading files, see “Loading Part Files” on page 31.

Selecting All Part Files

To select all of the part files, select **Edit→Select all files**. Visual Builder highlights all of the part files listed in the **Loaded part files** list box.

At this point, you can review the list to see if you want to deselect any of the files.

Deselecting All Part Files

To deselect all of the part files, select **Edit→Deselect all files**. Visual Builder removes the highlighting from all of the selected part files listed in the **Loaded part files** list box.

At this point, you can review the list to see if you want to select any of the files.

Touring Visual Builder

Customizing the Information Area

The following options allow you to specify the kind of information that Visual Builder displays in the information area for a selected part in the Visual Builder window. To use these options, select **Options→Information area** and then select one of the following options:

Show base class

Displays the C++ notation for a class and its base class. For example, if you select **IVBContainerControl** when this option is selected, Visual Builder displays the following in the information area to show that **IControl** is **IVBContainerControl**'s base class:

base: IVBContainerControl

Show description

Displays a brief description of the selected part. For example, if you select **IVBContainerControl** when this option is selected, Visual Builder displays the following description in the information area:

IBM VB container control

Show full file names

Displays the name of the part file in which the part is stored. For example, if you select **IVBContainerControl** when this option is selected, Visual Builder displays the following file name in the information area to show that **vbbase.vbb** contains the **IVBContainerControl** part.

from: VBBase.vbb

Seeing the Base Files

Select **Options→Show base files** to display the names of IBM-shipped part files loaded in Visual Builder. If this option is selected, the file names appear in the **Loaded part files** list box.

Seeing Where Part Files Are Located

Select **Options→Show full file names** to see the drive and directory where each of your part files is stored.

Seeing the Type List

The type list shows the types of data, such as enumerations and typedefs, for the parts contained in the part file that is currently selected. If no part file is selected or if the selected part file has no data types defined, this list is empty.

Once a type list is displayed, you can perform the following functions on data types that are selected:

- Delete data types
- Move data types to another part file
- Export data type definitions into part information files

These functions are available in the **Part** pull-down menu.

To display a type list, do the following:

1. Select the part file or files for which you want to see defined types.
2. Select **Options→Show type list**.

A list box titled **Loaded type information** is displayed at the bottom of the Visual Builder window, as shown in Figure 12.

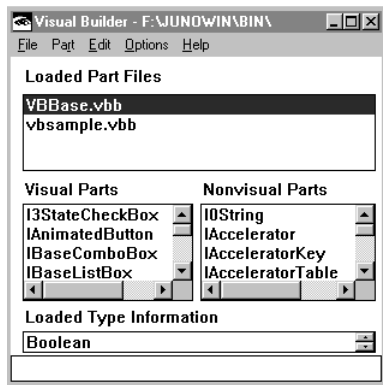


Figure 12. Visual Builder Window with Type Information Displayed

Touring Visual Builder

Using File Allocation Table (FAT) File Names

Select **Options→Default to FAT file names** if your system uses the File Allocation Table (FAT) file system. This option is selected by default when you first install VisualAge for C++. The FAT file system limits file names to a maximum of eight characters and file name extensions to a maximum of three characters.

When you select this option, Visual Builder uses these limits when creating file names and extensions, such as in the Class Editor when it provides a file name and extension for you to use when generating default code. For example, suppose you create a part and name it MyNewPart. This name has nine characters. If you generate code for this part, the default file name that Visual Builder uses for the files it generates will have only eight characters, as will the part file in which the part is saved if you allow Visual Builder to use a default name for that, too.

Attention: Be aware that Visual Builder does not check for existing file names when creating default file names. If you always use the default file name on a FAT system, Visual Builder may use a file name that has already been used, which may cause an existing file to be written over. For example, if you created another part named MyNewPart2, Visual Builder would use the same default file name as it used for MyNewPart.

Visual Builder assigns the file name when you create the part. Deselecting the **Default to FAT file names** option does not change the name of a file that has already been created.

Setting Make File Generation

Select **Options→Generate make files** if you want Visual Builder to generate a make file for you when you generate the source code for the main() function of your application.

Setting the Working Directory

Select **Options→Set working directory** if you want to store files created with Visual Builder in a different working directory. The default working directory is the directory in which you installed Visual Builder.

Note: This option does not appear if you start Visual Builder from a WorkFrame project.

When you select this option, Visual Builder displays the window shown in Figure 13 on page 37:

Touring Visual Builder

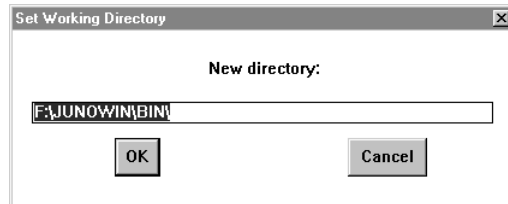


Figure 13. Window for Setting the Working Directory

To change the working directory, do the following:

1. Type the complete path to the directory in which you want to store Visual Builder files that you create.

The path consists of all directories that must be opened to get to the working directory.

2. Select the **OK** push button.

If the path you enter in the Working Directory window is not valid, Visual Builder displays an error message and resets the path to the last valid path that was entered.

Refreshing the Display

You might want to ensure that the information displayed in the Visual Builder window is current, for example, when you have loaded and unloaded several part files or moved parts from one part file to another. If such a situation occurs, you can cause the display to show the latest updates by selecting **Edit→Refresh**.

Touring Visual Builder

Chapter 5. Getting to Know the Visual Builder Editors

This chapter takes you on a tour of the Visual Builder editors. It begins with an overview of the editor symbols and then examines each editor in detail.

The Editor Symbols

The Editor symbols, located at the lower-right corner of the window, provide a fast-path to each of the Visual Builder Editors. They are as follows:



Composition Editor

Use the Composition Editor to create the views for your application, choose the parts that perform the logic you need, and make connections between the parts.

To learn more about the Composition Editor, see “The Composition Editor” on page 40.



Class Editor

Use the Class Editor to specify the names of files and resources associated with the current part.

To learn more about the Class Editor, see “The Class Editor” on page 49.



Part Interface Editor

Use the Part Interface Editor to define the features (attributes, actions, and events) for your parts, along with a list of preferred features for the pop-up connections menu. These features make up the part’s interface. You use them when you make connections between collaborating parts. You can also promote features of subparts from this editor.

To learn more about the Part Interface Editor, see “The Part Interface Editor” on page 56.

Touring Visual Builder

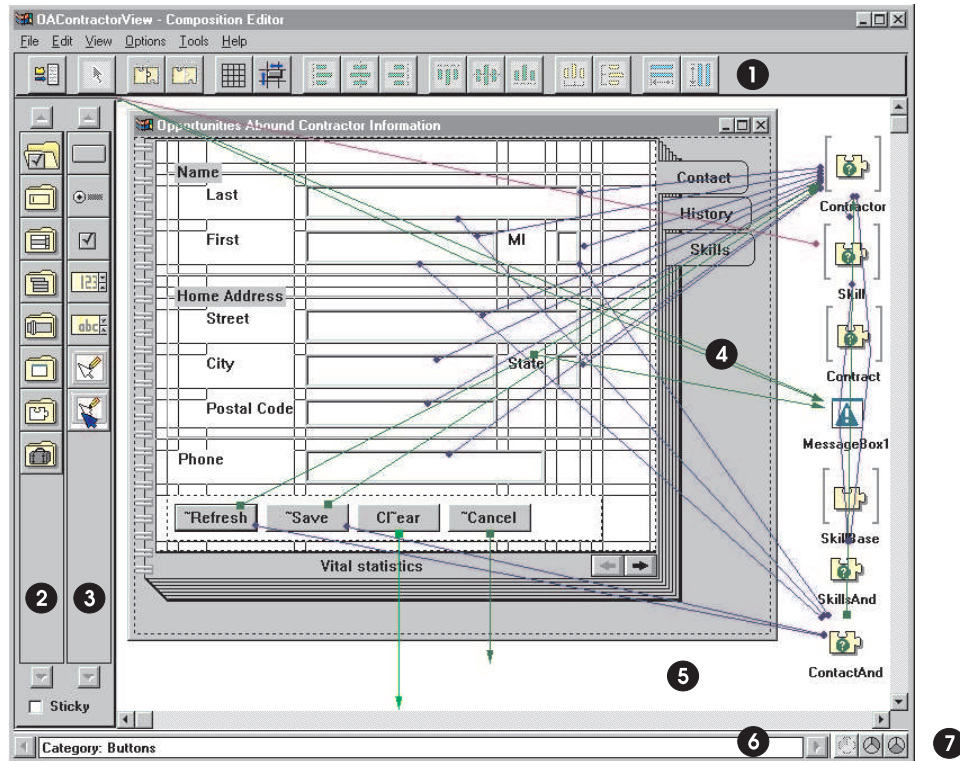
The Composition Editor

Use the Composition Editor to lay out the visual parts that make up your views, choose the parts that perform the logic you need, and make connections between them.

The Composition Editor is the editor you use to visually compose the various parts of your application. This section provides an overview of the Composition Editor's components. See Chapter 8, "Learning to Use Parts" on page 113 for information about visually composing an application.

The Composition Editor is shown in Figure 14 on page 41.

Touring Visual Builder



- | | |
|------------------------------|---------------------|
| 1 Tool bar | 4 Connection lines |
| 2 Parts palette - Categories | 5 Free-form surface |
| 3 Parts palette - Parts | 6 Information area |
| | 7 Editor symbols |

Figure 14. The Composition Editor

The Tool Bar

The tool bar appears below the menu bar of the Composition Editor. It contains icons that provide convenient access to actions that you commonly use when you create composite parts. These tools help you perform such tasks as the following:

- Aligning parts within your composite part
- Managing the connections between parts
- Unloading the mouse pointer
- Generating code for the part you are editing

Touring Visual Builder

All of the tools in the tool bar, except the **Selection tool**, act on the selected objects.



All of the tools available from the tool bar are also available from the **Tools** menu found on the Composition Editor's menu bar, except for the tool used to generate source code for your part. This tool is available in the **File** menu as the **Save and generate→Part source** choice.

The tool bar contains the following tools:



Generate Part Source

Generates C++ source code for the part that you are currently editing. This tool performs the same function as the **File→Save and generate→Part source** menu choice. For information about the source code files that Visual Builder generates, see "Source Files Created during Part Code Generation" on page 275.



Selection tool

Changes the mouse pointer from the crosshairs, which are used when the mouse pointer is loaded with a part, to the arrow that is used to select parts and perform actions on them. If the mouse pointer is not loaded, this tool is not available.

Connection tools



Show Connections

Displays all hidden connections to or from the selected parts. If no parts are selected, all connections are shown.



Hide Connections

Hides all displayed connections to or from the selected parts. If no parts are selected, all connections are hidden.

Touring Visual Builder

Grid tools



Toggle Grid

Toggles the display of the part alignment grid on and off for the selected parts. You can use separate alignment grids for parts in the Composers category and for the free-form surface.



Snap To Grid

Causes the selected parts to be repositioned to the nearest grid coordinate. The grid does not need to be visible for Snap To Grid to work.



Select the **Snap On Drop** and **Snap On Size** choices found in the Composition Editor **Options** menu to automatically align to the grid all parts that you add or size. This allows you to align parts to the grid without having to select the Snap To Grid tool for each part. Use the Snap To Grid tool if you only want to align selected parts to the grid.

Alignment tools



Align Left

Aligns the selected parts to the left edge of the last part selected.



Align Center

Aligns the selected parts along the vertical axis of the last part selected.



Align Right

Aligns the selected parts to the right edge of the last part selected.



Align Top

Aligns the selected parts to the top edge of the last part selected.



Align Middle

Aligns the selected parts along the horizontal axis of the last part selected.

Touring Visual Builder



Align Bottom

Aligns the selected parts to the bottom edge of the last part selected.

Distribution tools



Distribute Horizontally

Spaces the selected parts evenly between the left and right window borders.



Distribute Vertically

Spaces the selected parts evenly between the top and bottom window borders.

For information about the horizontal and vertical distribution of visual parts within a bounding box, see “Spacing Parts within a Bounding Box” on page 137.

Sizing tools



Match Width

Sizes the width of the selected parts to match that of the last part selected.



Match Height

Sizes the height of the selected parts to match that of the last part selected.

The Parts Palette

The parts palette is found on the left side of the Composition Editor. It contains icons for the parts that you use most frequently.

The parts palette organizes parts into categories. The icons in the left column of the parts palette represent the part categories. The right column of the parts palette contains the parts you use to build your application. When you select a category in the left column, the right column shows the parts contained within that category.

Notes:

- The information area at the bottom of the Composition Editor indicates which category and part are currently selected on the palette or which part or connection is currently selected on the free-form surface.

Touring Visual Builder

- You can add categories and parts to the parts palette, as well as delete categories and parts from it. See Chapter 19, “Adding Categories and Parts to the Parts Palette” on page 293 for information about adding parts that you create to the parts palette.
- See “Placing Parts on the Free-Form Surface” on page 124 for information about putting parts on the free-form surface.

The Visual Builder parts palette contains the following categories and parts. Refer to the *Visual Builder Parts Reference* for descriptions of these parts.

The asterisk at the end of the name indicates that the name is actually only a pointer to the part.



Buttons

Contains the following button parts:



IPushButton*



IRadioButton*



ICheckBox*



INumericSpinButton*



ITextSpinButton*



IGraphicPushButton*



IAnimatedButton*

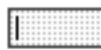


Data entry

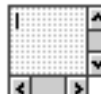
Contains the following data entry parts:



IStaticText*



IEntryField*



IMultiLineEdit*

Touring Visual Builder



Lists



IGroupBox*



IOutlineBox*



IBitmapControl*



IIconControl*

Contains the following list parts:



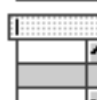
IListBox*



ICollectionViewListBox*



IComboBox*



ICollectionViewComboBox*



IVBContainerControl*



IContainerColumn*



Frame Extensions

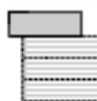
Contains the following parts that you can add to a window frame:



IToolBar*



IToolBarButton*

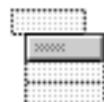


IMenu*

Touring Visual Builder



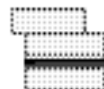
Sliders



IMenuItem*



IMenuCascade*



IMenuSeparator*



IVBInfoArea*



ITitle*

Consists of the following slider parts:



IProgressIndicator*



ISlider*



IScrollBar*



Composers

Consists of the following parts that are used to *contain* other visual parts:



IMultiCellCanvas*



ISetCanvas*



ISplitCanvas*



IViewPort*



INotebook*



ICanvas*



IFrameWindow*

Touring Visual Builder



Models

Contains the following nonvisual parts to help implement the logic of your application:



IVBFactory*



IVBVariable*



IVSequence*



Other

Contains the following miscellaneous parts:



IHelpWindow*



IMessageBox*



IVBFileDialog*



IVBFontDialog*



IVBFlyText*

The Free-Form Surface

The large open area in the Composition Editor (see Figure 14 on page 41) is called the *free-form surface*. This is the working area for visual programming, where you compose the various visual parts of your application and where you make connections to the logic of your application.

You add visual parts, such as static text and push buttons, to either a frame window part, to another part from the Composers category, or to the free-form surface itself. You add nonvisual parts (such as object factories and class interface parts) to your application by placing them on the free-form surface, not on a frame window part or on any other part from the Composers category.

Touring Visual Builder



You can also delete parts from the free-form surface. To do this, select one or more parts and do one of the following:

- Press the Delete key.
- Press mouse button 2 and select **Delete** from the pop-up menu.

The parts are deleted from the free-form surface.

For more information about using the free-form surface, see “Working with Parts on the Free-Form Surface” on page 124.

The Class Editor

The Class Editor enables you to specify the names of files and resources associated with the current part. You can use this editor to do the following:

- Enter a description of the part
- Specify a different .vbb file in which to store the part
- See the name of the part’s base class
- Modify the part’s default constructor
- Enter additional constructor and destructor code
- Specify a .lib file for the part
- Assign an icon resource to the part
- Specify names for the files in which Visual Builder stores generated code
- Specify other files that you want to include when you build your application
- Specify `#ifdef` and `#endif` preprocessor directives

Use your favorite text editor for creating new classes and member functions, writing application logic, and modifying existing member functions.

The Class Editor is shown in Figure 15 on page 50.

Touring Visual Builder

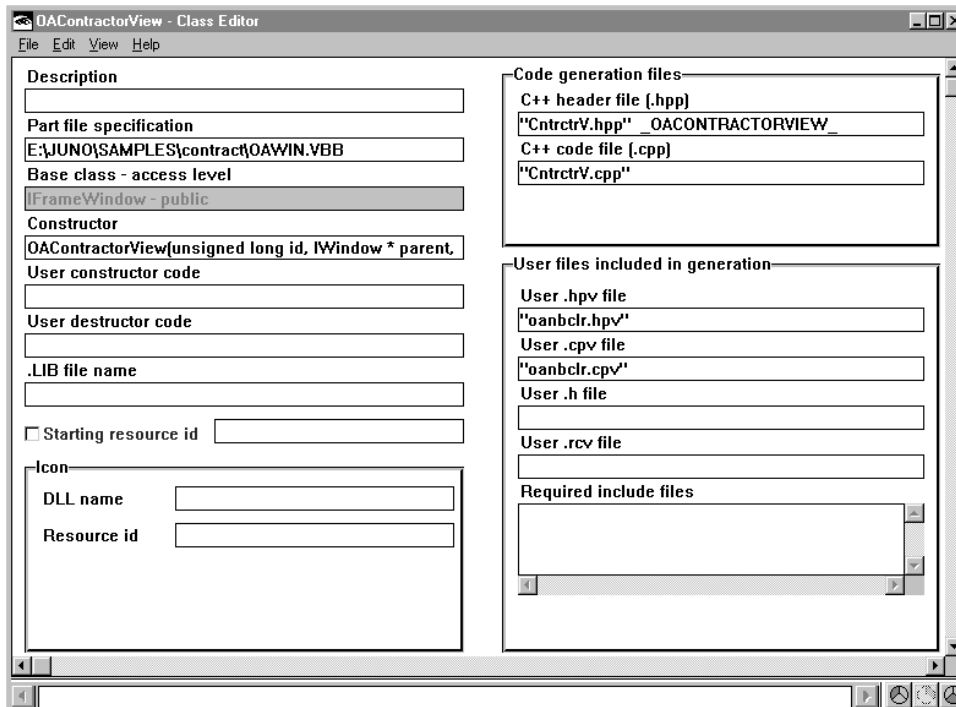


Figure 15. The Class Editor

If you cannot see all of the fields shown in Figure 15, use the scroll bar on the right side of the Class Editor to see the remaining fields.

Entering a Description of a Part

Use the **Description** field in the Class Editor to enter a description of your part. This description is used in the following places:

- If you add your part to the parts palette, the description appears in the information area at the bottom of the Composition Editor when you select the part.
- If you export your part information into a .vbe file, the description is included in the first line.

In the following example, the text shown in quotation marks was taken from the **Description** field for the ToDoList part.

```
//VBBeginPartInfo: ToDoList,"To-Do List sample application"
```

Moving a Part to a Different Part File

The **Part file specification** field in the Class Editor shows the name of the .vbb file that contains this part. If you want to move this part to another part file while using the Class Editor, do the following:

1. Replace the name of the current part file with the name of another part file in which you want to store the part.
2. Select **File**→**Save** to apply the change.

Visual Builder moves the part from the former part file to the one you just specified. If the part file you specified does not exist, Visual Builder creates it for you.

Seeing the Base Class of a Part

The **Base class–access level** field in the Class Editor shows the name of the base class for your part. This is the class name that you specified as the base class when you created the part.

This field also shows you the current access level to the base class: `public`, `protected`, or `private`.

You cannot modify the base class name or the access level.

Modifying a Part's Constructor

The **Constructor** field in the Class Editor initially contains a constructor prototype that Visual Builder inserts for you. If Visual Builder's prototype does not do exactly what you want it to do, you can modify it by typing over the text in this field. If you do modify Visual Builder's prototype, you must be aware of the assumptions made during generation of the constructor code.

When generating the part's base class initializer list, Visual Builder uses the following process to supply input arguments for the base class:

1. Visual Builder tries to match up the names of input arguments in the modified constructor with those of input arguments in the base class' constructor.
2. To supply arguments that remain unmatched, Visual Builder looks for the following special attributes in the primary part that correspond to input arguments in the base class' constructor. (In an `IFrameWindow`-based composite part, the `IFrameWindow*` part is the primary part.)
 - *id*, the part's window ID
 - *parent*, the part's parent window
 - *owner*, the part's owner
 - *rect*, the rectangle that represents the part's size and position

Touring Visual Builder

3. To supply arguments that are still unmatched, Visual Builder looks for default argument values in the header prototype for the base class' constructor.
4. If any arguments are not matched up at this point, Visual Builder displays an error message.

If you want your class to have multiple constructors, put them in the .hvp and .cpv files that contain your feature code and include them when the code is generated. Otherwise, if you modify your code after generating it, your changes will be lost the next time you generate your code.

For information about including files, see “Specifying Files to Include When You Build Your Application” on page 55.

Specifying Your Own Constructor Code

Use the **User constructor code** field in the Class Editor to enter your own constructor code for the part that you are editing. If you enter code in this field, it is added at the end of the default constructor that Visual Builder provides for you. If you have more than one line of code, put your code into a function and put the function name in this field. Put the code for this function in the files that Visual Builder creates when you generate your feature code. These file names are specified in the **User .hvp file** and **User .cpv file** fields.

Specifying Your Own Destructor Code

Use the **User destructor code** field in the Class Editor to enter your own destructor code for the part that you are editing. If you enter code in this field, it is added at the beginning of the default destructor that Visual Builder provides for you. If you have more than one line of code, put your code into a function and put the function name in this field. Put the code for this function in the files that Visual Builder creates when you generate your feature code. These file names are specified in the **User .hvp file** and **User .cpv file** fields.

Specifying a Library File

Use the **.LIB file name** field in the Class Editor to specify the name of a library file (.lib) that you want to be included when you generate the source code for your part. The library file points to a .dll file that contains information about a part in your application that was compiled separately. When you generate the source code for your part, Visual Builder includes a #pragma statement to include the library file.

When Visual Builder generates make files for other parts that use this part, the .lib file is specified in the make file as a dependency.

Specifying a Starting Resource ID

Use the **Starting resource ID** field in the Class Editor to specify the number that Visual Builder is to use as a starting point for generating resource IDs for your part. If this field is enabled, Visual Builder exports all translatable strings into a STRINGTABLE written to the file *partname.rci*. Visual Builder also calculates the corresponding resource IDs for these strings and writes them to a file named *partname.h*. These files are generated with the .cpp and .hpp files for the part.


The check box next to the field enables the starting resource ID. The first time you select this check box, Visual Builder enables the field and inserts a default starting resource ID. You can change this number. For information about how to choose a starting resource ID, see “Guidelines for Specifying Starting Resource IDs” on page 305.

If the check box is deselected later, the number in the field is included in the .h file when you generate the code, but it is not referenced.

Generating resource IDs is useful if the text in your application is being translated into another language. For more information, see “Using Resource Files for Translation” on page 303.

Specifying a Unique Icon for Your Part

Fill in the fields in the **Icon** group box in the Class Editor before you add your part to the parts palette so that you can use an icon other than the default icon provided

by Visual Builder to represent your part. The default icon is .

The **Icon** group box contains the following fields:

DLL name

The resource DLL that contains the icon you want to use. Enter just the file name, not the extension.

Resource ID

The resource ID number of the icon in the DLL whose name you entered in the **DLL name** field.

When you enter both the DLL name and a valid resource ID number, Visual Builder displays the icon that matches the resource ID number in the area below the **Resource ID** field. This occurs when you click on another field. This allows you to verify that you entered the correct resource ID number.

Touring Visual Builder

Note: If you do not specify a DLL, Visual Builder uses the default icon. If you specify a DLL but Visual Builder cannot find it, Visual Builder uses the



question mark icon, .

If the question mark icon appears, make sure the following conditions are met:

- The DLL exists and is in the current directory.
- The DLL file name is correct.
- The resource ID for the icon (in the .rc file) exists in the DLL.

Specifying the Names of Your Code Generation Files

The **Code generation files** group box in the Class Editor contains the following fields:

- A **C++ header file (.hpp)** field
- A **C++ code file (.cpp)** field

The file names displayed in these fields are the files into which your C++ header file code and source code are written. This occurs when you generate code from the Visual Builder window or from any of the editors by selecting **File→Save and generate→Part source**.

For complete information about generating source code, see Chapter 16, “Generating Source Code for Parts and Applications” on page 273.

The fields in the group box initially contain .hpp and .cpp file names that are based on the name of the part you are editing. To change the file names in these fields, select **File→Save** so that Visual Builder uses the new file names.

Attention: If the files already exist, Visual Builder replaces their contents with the code currently being generated.

Visual Builder writes both files to the working directory.

Specifying Files to Include When You Build Your Application

The **User files included in generation** group box in the Class Editor allows you to specify files that you want to be included when you build your application. These fields are typically used to contain the names of the files that hold user code that Visual Builder is to use when you generate feature code for your part's features.

Attention: If you import a nonvisual part and specify a user include source code and header files, do not generate part source for that nonvisual part. If you do, Visual Builder generates its own source code and header files, which overwrite your files.

The .hvp and .cpv file extensions are used because the WorkFrame Build tool tries to compile every .cpp file that it finds into an object module (.obj file). Because these files are not meant to be compiled by themselves, using the different file extension prevents this from happening.

The group box contains the following fields:

User .hvp file

The header file that you want to be included in the header file that Visual Builder generates. You must enter a file name in this field before you can generate feature code.

User .cpv file

The code file that you want to be included in the source code file that Visual Builder generates. You must enter a file name in this field before you can generate feature code.

User .h file

The resource header file that contains resource IDs for your application other than those that Visual Builder generates for you in the *partname.h* file. This file is included in *partname.h*.

User .rcv file

The resource file that contains text strings used in your application other than those that Visual Builder generates for you in *partname.rci* files. The *partname.rcv* file is included in *partname.rci*, which is created when you generate the part source code for the part.

The .rcv file extension is used because the WorkFrame Build tool tries to compile every .rc file that it finds into a binary resource (.res) file. Because partial resource files cannot always be compiled by themselves, using the different file extension prevents this from happening.

Touring Visual Builder

Required include files

The names of other files that you want Visual Builder to include when you generate source code for your application. Some examples of files that you might want to include are the following:

- Standard C++ library files, such as `stream.h`
- IBM Open Class Library header files, such as `IString.hpp`

Visual Builder generates the `#include` statements in the source code.

If you type a label next to the file name, Visual Builder uses that label to generate `#ifndef` and `#endif` statements for you. For example, if you want to include the header file for an address part that you created, you might enter the following in the **Additional include files** field:

```
address.hpp _ADDRESS_
```

This would cause Visual Builder to generate the following in your `.cpp` source code file:

```
#ifndef _ADDRESS_
#include <address.hpp>
#endif
```

If you omit the label, Visual Builder generates only the `#include` statement.

The Part Interface Editor

Each part that Visual Builder provides has a defined *part interface* that allows the part to interact with other parts. The part interface consists of *features*—attributes, events, and actions—that allow you to use the part in constructing your application. An entry field, for example, has a *text* attribute, a push button has a *buttonClickEvent* feature, and a frame window has a *setFocus* action.

The parts that you create must also have a defined part interface so they can be used. Use the Part Interface Editor to define that interface. To learn more about building reusable parts, see Chapter 7, “Creating Nonvisual Parts” on page 101.

Other uses for the Part Interface Editor include the following:

- Viewing the interface of a part
- Modifying or extending the interface of an existing part
- Creating or altering the list of *preferred features*, the features that are displayed in the pop-up connection menu for a part

The Part Interface Editor is a notebook made up of the following visual components:

- The **Attribute** page
- The **Event** page

Touring Visual Builder

- The **Action** page
- The **Promote** page
- The **Preferred** page

The Attribute Page

Use the **Attribute** page of the Part Interface Editor, shown in the following figure, to define the *attributes* for your part.

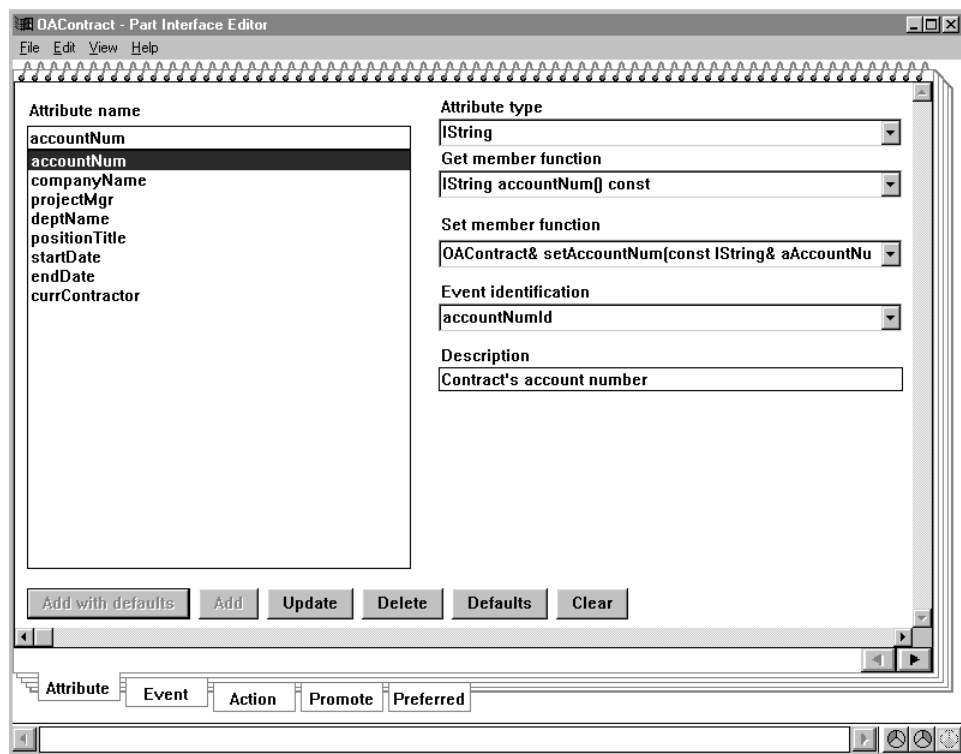


Figure 16. The Attribute page of the Part Interface Editor

You can define many attributes for a part, each with unique characteristics, such as name and data type. An attribute's value is always acquired by using the attribute's get and set member functions. It is up to you whether data for the attribute is stored in a data member, calculated, or acquired from some other location.

In addition, you can define three different kinds of behavior for your attributes, as follows:

Touring Visual Builder

full attribute

Contains all of the characteristics and behaviors that are available for an attribute, as follows:

- A get member function, which is required
- A set member function, which is optional (see no-set attribute below)
- An event identifier, which is optional (see no-event attribute below)

no-set attribute

Has no set member function. It can initialize another attribute if it is the source of an attribute-to-attribute connection, but it cannot set itself to the value of another attribute.

no-event attribute

Has no event identifier. Therefore, if you use a no-event attribute as the source of a connection, it cannot signal another part because of the lack of an event identifier. However, you can still use it as the source of connections so that it can initialize other attributes.

Adding an Attribute

To add an attribute in the Part Interface Editor, do one of the following:

- If you want to add the attribute using the default attribute type, get member function, set member function, and event identification that Visual Builder provides, enter a name in the **Attribute name** field and select the **Add with defaults** push button. Visual Builder adds the attribute to the part interface.
- If you want to see the default attribute type, get member function, set member function, and event identification that Visual Builder provides before you add the attribute, select the **Defaults** push button. Visual Builder displays the default information for the attribute in the fields on the right side of the **Attribute** page. Here are descriptions of those fields:

Attribute name

The name of the attribute. This name appears in the connection menu for the part when you make connections to or from it in the Composition Editor.

Attribute type

The data type of the attribute. The first time you create an attribute for your part, the default data type that Visual Builder uses is int. If you change int to another data type, such as IString, the new data type becomes the default data type for any new attributes that you create until you change it to something else or close the Part Interface Editor window. When you reopen the Part Interface Editor window, the default data type for any new attributes that you create reverts to int. It stays that way unless you change it again when creating another attribute or when modifying an existing attribute.

Touring Visual Builder

You typically define attributes as private data members to protect them from being changed by users of the part. The get and set member functions that you define for an attribute give other programmers access to the attribute's value. You probably do not want to expose all of the data members of your part as attributes, particularly if you intend to share the part with other programmers. You only need to decide which of the data members you want to make available for other programmers to reference. Some of your data members might be used for implementation purposes, so you would not want to identify these as attributes.

Get member function

The public member function used by the part and other parts to query or *get* the value of the attribute.

Set member function

The public member function used by the part and other parts to *set* the value of the attribute. An event identifier is typically signaled from within the implementation of the set member function to indicate that the value of the attribute changed.

If you press either the **Defaults** or **Add with defaults** push button, Visual Builder uses the data type in the **Attribute type** field to determine whether the parameter on the set member function passes the actual value of the attribute being set or whether it simply references the location of that value, as follows:

- If the data type in the **Attribute type** field represents the class name of a part in a loaded .vbb file, Visual Builder references the location of the value that is to be used to set the attribute.

For example, suppose you create a part named MyPart. If you create a *text* attribute for MyPart with an attribute type of *IString* and select either **Defaults** or **Add with defaults**, Visual Builder puts the following in the **Set member function** field:

```
virtual MyPart& setText(const IString& aText)
```

The & in front of the *aText* parameter means that, unless you modify this member function, it passes a reference to the value being used to set the *text* attribute. The & is used because *IString* is defined as a part in *vbbase.vbb*.

- If the data type in the **Attribute type** field is not defined in any loaded .vbb file, or if it is defined in a .vbb file but it does not represent the class name of a part, Visual Builder passes the actual value that is to be set for the attribute.

Using the MyPart example again, if you put an undefined data type in the **Attribute type** field before selecting either **Defaults** or **Add with**

Touring Visual Builder

defaults, Visual Builder puts the following in the **Set member function** field:

```
virtual MyPart & setText(const myType aText)
```

In this example, unless you modify this member function, it passes the actual value being used to set the *text* attribute.

Event identification

The name of the event identifier that is typically signaled from within the implementation of the set member function. You can create one of your own or you can use the default event identifier that Visual Builder supplies when you select either **Defaults** or **Add with defaults**.

Use this identifier to notify this part and other parts that the attribute's value has changed. This is typically done when the attribute is used as the source of a connection. The connection types that use an attribute as the source of a connection are as follows:

- Attribute-to-attribute
- Attribute-to-action
- Attribute-to-member function
- Custom logic, which can use either an attribute or an event as the source of a connection

You are not required to specify an event identification for any attribute because you are not required to notify other parts when the value of the attribute changes. However, failing to do so could prevent your application from passing necessary information from one part to another when it is needed.

The event identifier is typically signaled from within the implementation of the attribute's set member function, causing the attribute to behave as an event. Therefore, you do not need to specify another event with this event identifier on the **Event** page of the Part Interface Editor.

Description

A description of the attribute. This entry field is blank unless you enter a description, or you select either the **Defaults** or **Add with defaults** push button, in which case Visual Builder supplies a default description consisting of the attribute's name.

- If you want to add the attribute after seeing or modifying its default information or after entering your own information, select the **Add** push button. Visual Builder adds the attribute to the part interface.

Touring Visual Builder

Changing an Attribute

To change, or update, an attribute in the Part Interface Editor, do the following.

Note: You can change anything about an attribute except its name. To change an attribute's name, you must create a new attribute with the name you want to use.

1. Select the attribute that you want to change or type its name in the **Attribute name** field.
2. Make the changes that you want to make in the fields on the right side of the **Attribute** page.
3. Select the **Update** push button. Visual Builder updates the changes you made in its internal data model. To save the changes, select **File→Save**.

If you select **Update** and try to close the part without selecting **File→Save**, Visual Builder displays a message asking if you want to save the file.

Deleting an Attribute

To delete an attribute in the Part Interface Editor, do the following:

1. Select the attribute that you want to delete or type its name in the **Attribute name** field.
2. Select the **Delete** push button. Visual Builder deletes the attribute.

Setting Defaults for an Attribute

To set defaults for an attribute in the Part Interface Editor, do the following:

1. Select the attribute that you want to set defaults for or type its name in the **Attribute name** field.
2. Change the attribute type in the **Attribute type** field.
3. Select the **Defaults** push button.

Visual Builder changes all occurrences of the former attribute type to the new attribute type in the fields on the right side of the **Attribute** page.

Clearing the Attribute Page Fields

To clear the fields on the **Attribute** page, select the **Clear** push button.

Visual Builder clears all of the fields on the **Attribute** page.

Touring Visual Builder

An Attribute Example

Suppose you create a nonvisual Customer part that inherits from Visual Builder's sample ICustomer part. You also create an *age* attribute for which you specify the following:

- An age get member function
- A setAge set member function
- An ageId event notifier

You create this attribute so other parts can access its value or so it can be passed as a parameter value.

Note: Visual Builder does not use the value that is passed on the attribute's event. It uses the attribute's get member function instead.

The feature source code that Visual Builder would generate for the *age* attribute would be similar to these code segments, which show the following.

Note: You must enter the names of the .hvp and .cpv files in the **User .hvp file** and **User .cpv file** fields, respectively, in the Class Editor before generating the feature source code.

- The declarations of the get member function, set member function, event identifier, and data member in the user header (.hvp) file.

```
// Feature source code generation begins here...
public:
    virtual IString age() const;           /* Get member function */
    virtual Customer& setAge(
        const IString& aAge);             /* Set member function */

    static INotificationId ageId;          /* Event notifier    */

private:
    IString iAge;                          /* Data member        */
// Feature source code generation begins here...
```

- The event notifier initialized as a text string, followed by the get and set member functions for the *age* attribute, as defined in the user source code (.cpv) file.

Touring Visual Builder

```
// Feature source code generation begins here...
INotificationId Customer::ageId = "Customer::age";

int Customer::age() const
{
    return iAge;
}

Customer& Customer::setAge(const IString& aAge)
{
    if (!(iAge == aAge))
    {
        iAge = aAge;
        notifyObservers(INotificationEvent(Customer::ageId, *this));
    } // endif
    return *this;
}

// Feature source code generation ends here.
```

The Event Page

Use the **Event** page of the Part Interface Editor, shown in the following figure, to define the *events* you use to notify this part or other parts about changes you decide are significant. For example, you might notify other parts when an attribute is set to a certain value or when important processing is finished. In this way, someone using your reusable part can link to one of your part's events and receive automatic notification of the event whenever it is triggered.

Touring Visual Builder

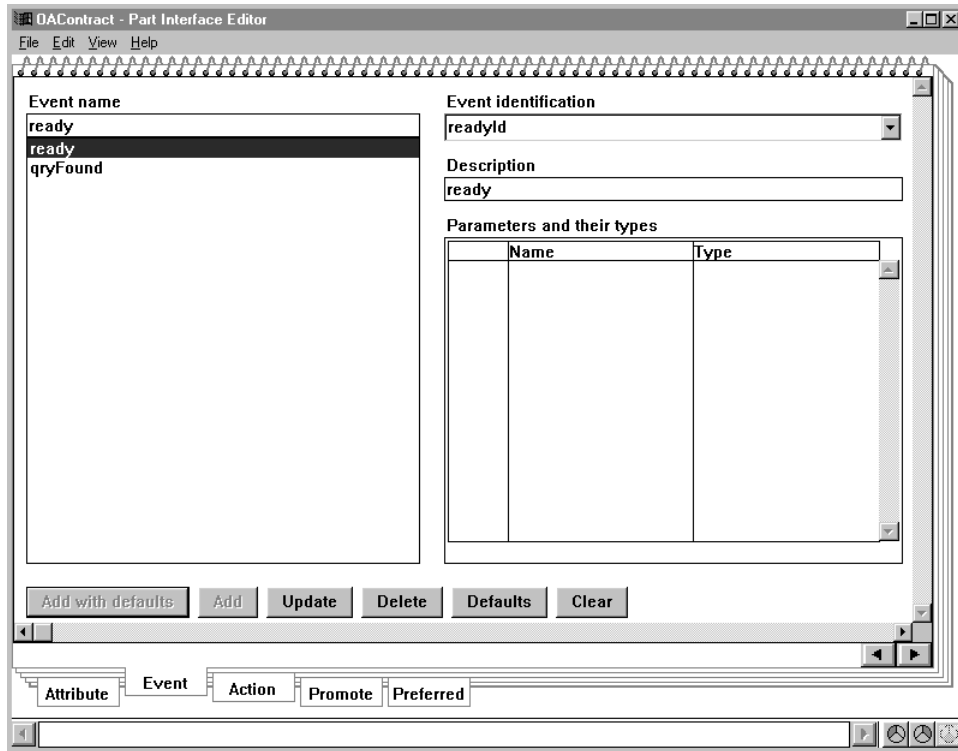


Figure 17. The Event Page of the Part Interface Editor

If you cannot see all of the fields shown in the preceding figure, use the scroll bar on the right side of the Part Interface Editor to see the remaining fields.

The names of the part's events are displayed in the list box named **Event name**. When you create a new part, the first time you open the Part Interface Editor and turn to the **Event** page, you see that Visual Builder has provided an event for you: the *ready* event.

The Ready Event

Visual Builder adds the *ready* event to every new part that you create using the **Part→New** menu choice in the Visual Builder window. However, this event is not added if you import part information from a part information file (.vbe file).

By connecting the *ready* event to one of your part's subparts, you can cause an action or member function to be invoked when your application is executed. The *ready* event is sent to the part when both of the following occur:

- All subparts of the part have been constructed.

Touring Visual Builder

- All connections have been made and initialized.

For example, suppose you have a part named MyApp and this part is an application that has a window that contains an animated push button. By connecting the MyApp part's *ready* event to the animated push button's *startAnimation* action, you cause the push button to become animated as soon as the application starts running.

The *ready* event is not a preferred feature by default, but you can add it to your part's preferred features list. For information on how to do this, see "The Preferred Page" on page 75.

Adding an Event

To add an event in the Part Interface Editor, do one of the following:

- If you want to add the event using the default event identification that Visual Builder provides, enter a name in the **Event name** field and select the **Add with defaults** push button.

Visual Builder adds the event to the part interface.

- If you want to see the default event identification that Visual Builder provides before you add the event, select the **Defaults** push button.

Visual Builder displays the default event identification in the **Event identification** field on the right side of the **Event** page. Here are descriptions of that field and the other fields on the page:

Event name

The name of the event. If you add this event name to the preferred features list, it appears in the pop-up connections menu for the part.

Event identification

The name of the event identifier that is actually used in a member function signalling this event.

Description

A description of the event. This entry field is blank unless you enter a description or unless you select either the **Defaults** or **Add with defaults** push button. In the latter case, Visual Builder inserts a default description consisting of the event's name.

Parameters and their types

A table used to define a parameter and data type sent as part of an event.

Name

The name of the event parameter.

Touring Visual Builder

Type

The data type of the event parameter, such as IString, integer, or unsigned long.

The parameter that you specify in this table is used to specify event data, or a reference to that data's location, that you want to send along with the event identifier as part of the event. Doing this prevents you from having to perform a separate query for the data.

Adding a parameter and its type

To add a parameter and its type to the **Parameters and their types** table, do the following.

Note: You can add only one parameter and type for each event.

1. Move the mouse pointer over the table and click mouse button 2.
Visual Builder displays a pop-up menu.
2. Select either **Add before** or **Add after**.
3. Select the **Update** push button. Visual Builder adds a row to the table with a default parameter name and type.

Changing parameter names and types

To change the parameter names and types in the table, do the following:

1. Click on the parameter name with mouse button 1.
2. Type the parameter name you want to use.
3. Click on the parameter type with mouse button 1 or press the Tab key.
4. Type the parameter type you want to use.

Note: To exit the edit mode, press Shift+Enter.

5. Select the **Update** push button.

Deleting parameter names and types

To delete parameter names and types from this table, do the following:

1. Select the row by moving the mouse pointer to the number in the left-hand column and clicking mouse button 1.
2. With the mouse pointer still over the row, click mouse button 2.
3. Select **Delete selected row** from the pop-up menu. The row is deleted.

Changing an Event

To change, or update, an event, do the following.

Note: You can change anything about an event except its name. To change an event's name, you must create a new event with the name you want to use.

1. Select the event that you want to change or type its name in the **Event name** field.
2. Make the changes that you want to make in the fields on the right side of the **Event** page.
3. Select the **Update** push button. Visual Builder saves the changes you made.

Deleting an Event

To delete an event, do the following:

1. Select the event that you want to delete or type its name in the **Event name** field.
2. Select the **Delete** push button. Visual Builder deletes the event.

Note: If you added the event that you just deleted to the preferred features list, you must go to the **Preferred** page and delete it there, too.

Setting Defaults for an Event

To set defaults for an event, do the following:

1. Select the event that you want to set defaults for or type its name in the **Event name** field.
2. Select the **Defaults** push button. Visual Builder changes the former event identification to the default event identification in the **Event identification** field.

Clearing the Event Page Fields

To clear the fields on the **Event** page, select the **Clear** push button.

Visual Builder clears all of the fields on the **Event** page.

An Event Example

Using the same example shown for the **Attribute** page, suppose you also create an *invalidDataEntered* event for which you specify the following:

- An *invalidDataEnteredId* event notifier
- An *invalidData* parameter with a data type of *IStrng*

You create this event because you want to show an error message for the Customer part whenever invalid data is entered for any of the Customer part's attributes, such as the customer's age being outside a valid range. Then, in your source code for the

Touring Visual Builder

age attribute's set member function, you can call the `notifyObservers` member function to display a message asking for a valid age if the check fails.

The default feature code that Visual Builder would generate for the `invalidDataEnteredId` event notifier would be similar to these code segments, which show the following:

- The declarations of the get member function, set member function, event identifiers, and data member in the user header (.hvp) file.

Note: Sample code highlighted in bold shows the code that was added to modify the previous example.

```
public:
    IString age() const;           /* Get member function */
    Customer & setAge(
        const IString &aAge);     /* Set member function */

    static INotificationId ageId; /* Event notifiers */
    static INotificationId invalidDataEnteredId;

private:
    IString iAge;                 /* Data member */

    INotificationId Customer::ageId = "Customer::ageId";
    INotificationId Customer::invalidDataEnteredId =
        "Customer::invalidDataEnteredId";

    IString Customer::age() const /* The age attribute's get */
    {                             /* member function returns */
        return iAge;             /* the value of the data */
    }                             /* member. */

    Customer & Customer::setAge( /* The age attribute's set */
        const IString &aAge)     /* member function sends */
    {                             /* the event notification */
        if (aAge > 99)
        {
            notifyObservers(
                INotificationEvent(
                    Customer::invalidDataEnteredId,
                    *this,
                    true,
                    (void *) IString("Enter a value between 1 and 99."));
            }
        }
        else
        {
            if (iAge != aAge)
            {
```


Touring Visual Builder

```
iAge = aAge;
IString eventData(iAge);
notifyObservers(INotificationEvent(Customer::ageId, *this,
                                true, (void *)&eventData));
} /* endif */
}
return *this;
}
```

The Action Page

Use the **Action** page of the Part Interface Editor, shown in the following figure, to define the *actions* your reusable part uses to perform specific tasks. Often, you will want to perform some task when a specific event occurs.

For example, you might want to update a *balance* attribute each time a push button's *buttonClickEvent* feature is triggered. You might create an action named *updateBalance* to perform this task and connect it to the push button's *buttonClickEvent* feature.

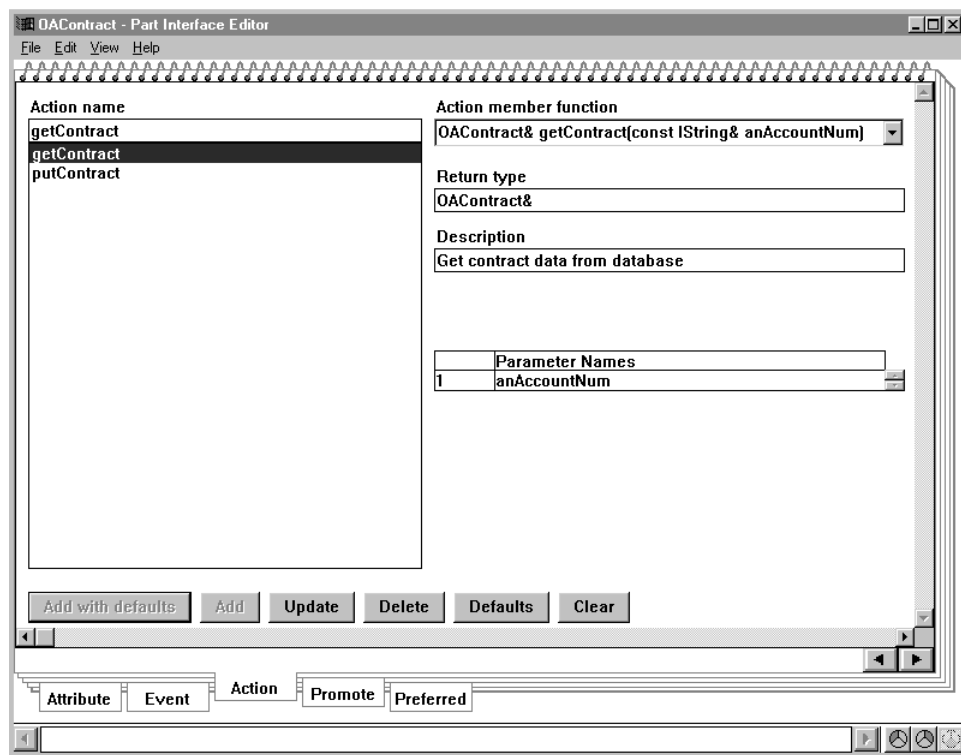


Figure 18. The Action Page of the Part Interface Editor

Touring Visual Builder

The names of the part's actions are displayed in the list box below the **Action name** field.

Adding an Action

To add an action in the Part Interface Editor, do one of the following:

- If you want to add the action using `int` as the default return type for both the action member function and for the value that the member function returns, enter a name in the **Action name** field and select the **Add with defaults** push button. Visual Builder adds the action to the part interface. The action name is used as the action member function name with a return type of `int` in the **Action member function** field. In addition, `int` is inserted in the **Return type** field.
- If you want to see the default return type (`int`) before you add the action, select the **Defaults** push button. Visual Builder displays the default return type in the **Return type** field, as well as in the **Action member function** field, on the right side of the **Action** page. Here are descriptions of those fields and the other fields on the page:

Action name

The name of the action. This name appears in the pop-up menu for the part when you make connections to or from it in the Composition Editor if you add it to the preferred features list. If you add this action name to the preferred features list, it appears in the pop-up connections menu for the part.

Action member function

The name of the public member function, defined within your reusable part, that performs the action.

Return type

The return type of the action member function. The default is `int`. Leaving this field blank is the same as entering a return type of `void`. In either of these cases, the *actionResult* feature is not available for connections because the action member function does not return anything.

The default feature code that is generated depends on the return type specified in this field. You can, and in most cases should, edit the default code to specify what you want to be returned. The following list shows what is generated:

int

Generates default code to return 0.

Reference to same part

Generates default code to return a pointer to *this*.

Touring Visual Builder

Anything else

Generates a comment line delimited with question marks, which requires you to specify what to return. Otherwise, the compiler generates an error.

Description

A description of the action.

Parameter names

Parameters of the selected action. Initially, the names that are inserted in the **Parameter names** table are the same as the parameter names that are specified in the **Action member function** field.

The parameter names in the table are linked to the action name and can appear in a pop-up connection menu. The purpose of the table is to allow you to change the parameter names that appear in the part interface without affecting the member function definition itself. That is, if you change the parameter names in the table, the parameter names in the **Action member function** field do not change.

Here are some reasons for using the **Parameter names** table:

- To give parameters descriptive names. Parameter names used in member function definitions are usually not very descriptive. The **Parameter names** table lets you enter descriptive names for your parameters. The parameter names that you enter in this table appear in pop-up connection menus. By entering descriptive names for your parameters, you can make the source or target of the connection easier to determine than if the less descriptive member function parameter names were used.
- To associate specific parameter names with actions. You might have more than one action name for the same public member function. If so, to make your part interface easier to use, you might not want to have the same parameter name associated with each action. The **Parameter names** table allows you associate separate, descriptive parameter names with each action, even though those names represent the same parameter in the same member function.
- To change a member function without changing its action name in the part interface. You might decide to change a member function. For example, you might rename a parameter or substitute another member function entirely. The **Parameter names** table allows you to do this without having to change your part interface. You can modify and then update the information in the **Action member function** field without changing the parameter names in the table. Therefore, if other programmers are using your part, they continue to see the same part interface. They would only need to regenerate their source code.

Touring Visual Builder

Changing an Action

To change, or update, an action in the Part Interface Editor, do the following.

Note: You can change anything about an action except its name. To change an action's name, you must create a new action with the name you want to use.

1. Select the action that you want to change or type its name in the **Action name** field.
2. Make the changes that you want to make in the fields on the right side of the **Action** page.
3. Select the **Update** push button. Visual Builder saves the changes you made.

Deleting an Action

To delete an action in the Part Interface Editor, do the following:

1. Select the action that you want to delete or type its name in the **Action name** field.
2. Select the **Delete** push button. Visual Builder deletes the action.

Note: If you added the action that you just deleted to the **Preferred Features** list, you must go to the **Preferred** page and delete it there, too.

Setting Defaults for an Action

To set defaults for an action in the Part Interface Editor, do the following:

1. Select the action that you want to set defaults for or type its name in the **Action name** field.
2. Select the **Defaults** push button. Visual Builder changes the former return type to the default return type in both the **Action member function** and the **Return type** fields.

Changing Parameter Names

To change the names in the **Parameter names** table on the **Action** page in the Part Interface Editor, do the following:

1. Place the mouse pointer over a parameter name and click mouse button 1.
2. Edit the text in the cell.
3. To exit the edit mode, press **Shift-Enter**.
4. Select the **Update** push button.

Touring Visual Builder

Clearing the Action Page Fields

To clear the fields on the **Action** page, select the **Clear** push button.

Visual Builder clears all of the fields on the **Action** page.

The Promote Page

Use the **Promote** page of the Part Interface Editor to specify features that you want to connect to another part when this part is embedded as a subpart within another part. The features (attributes, events, and actions) that you specify appear in the window that is displayed when you select **More** from this part's pop-up connection menu. For a complete description of promoting a part's features, see "Promoting a Part's Features" on page 153.

Figure 19 shows the **Promote** page:

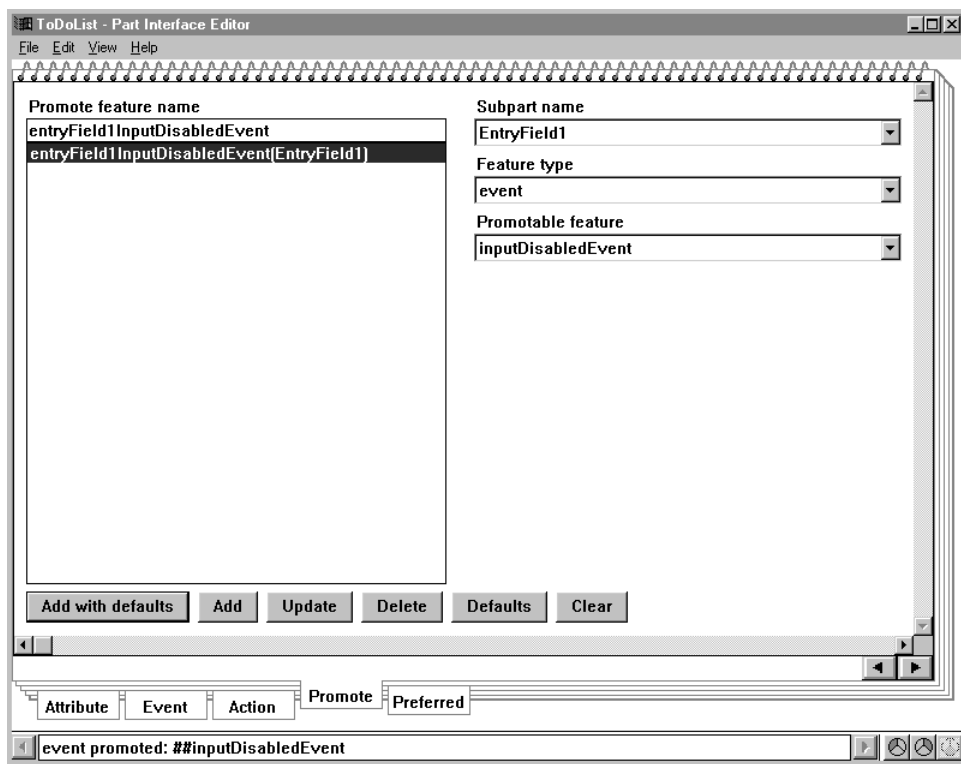


Figure 19. The Promote Page of the Part Interface Editor

Touring Visual Builder

Promoting a Feature

To promote a feature in the Part Interface Editor for a subpart, do the following:

1. Select a subpart name from the list box beneath the **Subpart name** field or type the name in the field. Visual Builder displays the name of the subpart that you select in the **Subpart name** field.
2. Select a feature type from the list box beneath the **Feature type** field or type the name in the field. Visual Builder displays the type that you select (attribute, event, or action) in the the **Feature type** field.
3. Select the feature that you want to promote from the list box beneath the **Promotable feature** field or type the name in the field. Visual Builder displays the feature that you select in the **Promotable feature** field.
4. Do one of the following:
 - If you want to promote the feature using the default name that Visual Builder provides, select the **Add with defaults** push button. Visual Builder promotes the feature and displays the feature name in both the **Promote feature name** field and in the list box below this field.
 - If you want to see the default feature name that Visual Builder provides before you promote the feature, select the **Defaults** push button. Visual Builder displays the default name for the feature in the **Promote feature name** field.
 - If you want to promote a feature after seeing its default name or typing another name that you prefer, select the **Add** push button. Visual Builder promotes the feature using the name in the **Promote feature name** field and displays the name in the list box below this field.

Changing a Promoted Feature

To update a feature that you have already promoted, do the following:

1. Select the promoted feature that you want to update.
2. Select those aspects of the promoted feature that you want to update in the fields on the right side of the **Promote** page. You can select any or all of the following:
 - Subpart name
 - Feature type
 - Promotable feature
3. Select the **Update** push button. Visual Builder updates those aspects of the feature that you selected.

The only noticeable change is the subpart name if you selected a different one. The subpart name shown in parentheses behind the promoted feature name changes if you

Touring Visual Builder

selected a different subpart. For example, suppose you promoted the *buttonClickEvent* feature of `PushButton1` and then realized you should have promoted the *buttonClickEvent* feature of `PushButton2`. Instead of deleting the promoted feature, you can update it by changing only its subpart.

Deleting a Promoted Feature

To delete a promoted feature, do the following:

1. Select the promoted feature that you want to delete.
2. Select the **Delete** push button. Visual Builder deletes the promoted feature from the list box beneath the **Promote feature name** field.

Note: If you added the promoted feature that you just deleted to the preferred features list, you must go to the **Preferred** page and delete it there, too.

Clearing the Promote Page Fields

To clear the fields on the **Promote** page, select the **Clear** push button. Visual Builder removes the information from all of the fields on the page.

The Preferred Page

Use the **Preferred** page of the Part Interface Editor, shown in the following figure, to specify the *preferred features* for your part—the features that you use most often when connecting this part to another part. The features (attributes, events, and actions) that you specify appear in the pop-up menu that is displayed when you begin a connection on this part. You can include any features that exist for your part, as well as any features that your part inherits from other parts.

Touring Visual Builder

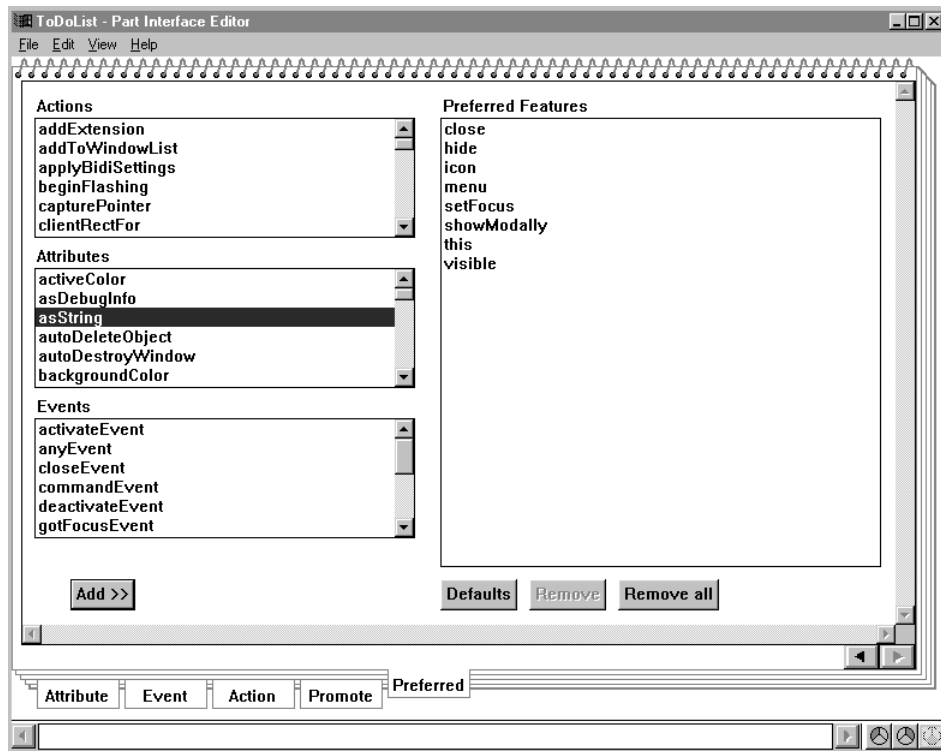


Figure 20. The Preferred Page of the Part Interface Editor

In addition to these features, the pop-up connection menu contains the **More→** selection. This selection allows you to display a window that contains all of the features for this part, as well as the features it inherits. Visual Builder provides this window in case you need a feature that is not in the preferred features list.

The names of the part's features are displayed in the list boxes named **Actions**, **Attributes**, and **Events** on the left side of the page. The preferred features are displayed in the **Preferred features** list box on the right side of the page.

Adding a Preferred Feature

To add a preferred feature to the connection menu for a part, do the following:

1. Select a feature name from one of the list boxes on the left side of the page.
2. Do either of the following:
 - Select the **Add** push button at the bottom of the page.
 - With the mouse pointer still over the list box in which you selected the feature name, do the following:

Touring Visual Builder

- a. Click mouse button 2. A pop-up menu with the **Add** choice appears.
- b. Select **Add** to add the feature.

The feature name that you selected is inserted into the **Preferred features** list box in alphabetical order.

Removing a Preferred Feature

You can remove a preferred feature name from the connection menu for a part. Doing this removes the feature from the menu only; it does not delete the feature.

To remove a preferred feature from the connection menu for a part, do the following:

1. Select the name that you want to remove from the **Preferred Features** list box.
2. Do either of the following:
 - Select the **Remove** push button at the bottom of the page.
 - With the mouse pointer still over the feature name in the **Preferred features** list box, do the following:
 - a. Click mouse button 2. A pop-up menu appears.
 - b. Select **Remove** to remove the selected feature.

A message box is displayed to make sure you want to remove the name of this preferred feature.

3. Select **Yes** to remove the feature name from the list. The feature name that you selected is removed from the list.

Removing All Preferred Features

You can remove all of the feature names from the connection menu for a part. Doing this removes the features from the menu only; it does not delete the features. Once you remove all preferred features from the connection menu, you must select **More** to use the features in a connection.

To remove all of the preferred features from the connection menu for a part, do either of the following:

- Select the **Remove all** push button at the bottom of the page.
- With the mouse pointer over the **Preferred Features** list box, do the following:
 1. Click mouse button 2. A pop-up menu appears.
 2. Select **Remove all** to remove all of the selected features.

A message box is displayed to make sure you want to remove all of the preferred features.

Touring Visual Builder

Select **Yes** to remove all of the feature names from the list. All of the feature names are removed from the list.

Showing Inherited Preferred Features Only

To show only the preferred features that your part inherits from other parts, select the **Default** push button.

A message box is displayed. Select **Yes** to display only the inherited preferred features.

Part 3. Developing Visual Builder Applications

This part provides the information you need to create a basic Visual Builder application.

Chapter 6. Designing Effective Applications	83
Designing Parts for a Single Purpose	83
Taking the Object-Oriented Approach	84
Identifying Reusable Parts	85
Using Inheritance	86
Using Abstract Classes	88
Keeping Your Components Small	89
Designing Parts for the OASearch Application	90
Chapter 7. Creating Nonvisual Parts	101
Defining the Part Interface	101
Adding Code to Your Part	105
Modifying the Generated Feature Code	109
Adding Code Created Outside Visual Builder	112
Chapter 8. Learning to Use Parts	113
Working with Parts in the Visual Builder Window	113
Working with Parts on the Free-Form Surface	124
Constructing a GUI: the OASearch Application	157
Chapter 9. Learning to Use Connections	163
Connection Type Summary	170
Making the Connections	171
Connecting Features to Member Function Connections	179
Connecting Features to Custom Logic	184
Connecting Exception Events to Actions and Member Functions	188
Manipulating Connections	189
Rearranging Connections	196
Making Connections for the OASearch Application	199
Chapter 10. Displaying Objects in a List Box	207
Copying the ToDoList Part	208
Creating the ToDoItem Nonvisual Part	208
Replacing and Modifying the List Box	211
Placing and Modifying an Object Factory Part	212
Placing and Modifying an IVSequence* Part	212
Making the New Connections	213

Developing Applications

Generating the Source Code for Your Visual Part and main() Procedure	214
Building and Running the Modified Application	215
Chapter 11. Creating Resizable Windows	217
What Are the Benefits of Using Multicell Canvases?	217
Adding a Multicell Canvas	219
Adding Parts to the Multicell Canvas	219
Changing the Multicell Canvas Grid	222
Extending a Part to Span More than One Cell	223
Adding a Group Box	225
Changing the Settings for a Multicell Canvas	227
Chapter 12. Constructing Containers and Notebooks	231
Adding Container Parts	231
Adding Notebook Parts	237
Chapter 13. Adding Menus to Your Application	243
Types of Menus and Menu Items	243
Adding a Menu Bar	244
Connecting the Menu Bar to the Window	245
Adding Menu Choices	246
Adding Menu Separators	246
Connecting Menu Choices to Actions	247
Chapter 14. Adding Help to Visual Builder Applications	249
Creating the Help File	250
Providing Context-Sensitive Help	252
Providing General Help	253
Providing the Application Help Window	254
Providing Help for Factory-Generated Frame Windows	256
Providing a Help Push Button	257
Displaying Fly-Over Help When the Mouse Pointer Is Over a Part	258
Displaying Help in an Information Area	259
Chapter 15. Integrating Visual Parts into a Single Application	263
Adding Nonvisual Support Parts to the Primary Part	263
Adding Static Parts	264
Adding Visual Parts as Dynamic Instances	264
Completing the Menu Bar	269
Resetting Your Application's Main Resource Library	270
Chapter 16. Generating Source Code for Parts and Applications	273
Preparing for Source Code Generation	273

Developing Applications

Generating C++ Source Code for Individual Parts	274
Generating Source Code for Your Application's main() Function	276
Preparing Generated Files for Compilation	277
Packaging Runtime DLLs with Your Application	279
Compiling and Linking Your Application	279
Chapter 17. Sharing Parts with Others	283
Providing part files (.vbb)	283
Providing Part Information Files (.vbe)	284

Developing Applications



Chapter 6. Designing Effective Applications

When designing your applications, remember that Visual Builder implements parts as C++ classes. Therefore, keep in mind good design practices and techniques that apply to both classes and parts.

At the heart of Visual Builder lies visual programming and construction from parts. The predefined palette of reusable parts coupled with the Composition Editor provide a new level of power in developing applications. Of course, increased power does not guarantee good design and improved productivity, so the following sections are provided to assist you in designing more effective Visual Builder applications.

Designing Parts for a Single Purpose

Much like designing the user interface of any application, you need to design each part in your Visual Builder application for a single purpose. For example, consider an application that queries a simulated database for information about contractors, their skills, and the contracts to which they are assigned. These queries clearly identify the three actions that can be performed using the application. Therefore, this application needs a separate part that shows the results of each query.

These parts could include a notebook that displays information about individual contractors, a window with entry fields that contain information about a specific contract, and a container that shows which contractors have a certain skill. Each of these specific queries becomes the single purpose around which you design your parts. Of course, supporting parts such as a main window for starting the queries, nonvisual parts that supply the results of a search, and dialogs for specifying the search targets are also required.

You can then combine your parts into a single business-wide application that can be shared by multiple users. Each user has direct access to the parts needed to complete a specific query without interference from any parts of the application that are conducting other queries.

In “Designing Parts for the OASearch Application” on page 90, we introduce a sample application that can perform the queries described in this section and explain the design of each of the application’s parts.

Developing Applications

Taking the Object-Oriented Approach

Object-oriented design seeks to find parallels between computer entities and real-world entities. By not blurring the lines between the roles and tasks of the business, and by providing a part for each, you effectively provide a computer application that parallels the way your business operates in the real world.

Thinking more closely in terms of Visual Builder and the Composition Editor, parts that are too large can have a negative effect on usability, performance, and maintenance. As a general rule, for each part, you want to use the minimum number of visual parts, nonvisual parts, connections, and parameters necessary to achieve a single purpose. *Building VisualAge for C++ Parts for Fun and Profit* contains implementation checklists that you might find helpful in keeping the number of parts to a minimum.

In addition, try to use attributes whenever possible instead of actions. Attribute-to-attribute connections, for example, tend to minimize the number of required connections because usually you do not need any additional connections to provide a parameter as you do with actions.

Steps for creating an OO application

Here is a sample list of steps you might follow to create an OO application using Visual Builder:

1. Design the application.
 - Decide which tasks and functions you want the application to perform.
 - Determine how you will represent those tasks and functions on the interface.
 - Design the user interface, deciding which windows the application displays.
 - If more than one window is involved, decide which one is to be the main window. The *main window* normally consists of the following:
 - The window that users first see when they start the application
 - Any other visual parts that are included in the window, such as list boxes, entry fields, and push buttons
 - Any nonvisual parts that are needed, such as object factories
 - Decide which parts the application needs in order to provide those windows and to support the task a user wants to accomplish.
 - Decide which member functions the parts need by determining what messages need to be sent from one object to another.

Developing Applications

- Decide which data members the parts need by determining the various types of data the part is to provide.
 - Decide which features (attributes, events, and actions) the parts need.
 - Decide which features of each part need to be connected to features of another part. For example, a **Close** push button can close a window only if its *buttonClickEvent* feature is connected to the window's *close* action.
2. Create any parts you need that Visual Builder does not already provide or that someone else has not already created.
 3. Create the main window and any other required windows using the parts and connections predetermined in the previous steps.
 4. Generate the C++ code for the application.
 5. Make any necessary modifications to the generated C++ feature code.
 6. Build and run the application.

Developing an OO application is a process in which these (and perhaps other) steps are usually repeated many times. For that reason, the preceding steps are provided merely as an example. You might not follow them exactly in the order given when creating your applications. For example, you might not realize that you need to create a certain part until you have already started creating a composite part.

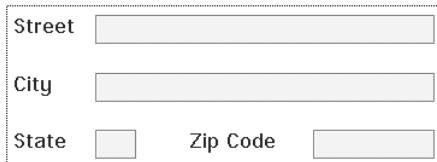
The next section, “Identifying Reusable Parts,” suggests ways to get the most from parts, once you have settled upon their high-level design.

Identifying Reusable Parts

Once you have the high-level design of your application's parts, identify common elements. These common elements are perfect opportunities for reuse. Make them into reusable parts and add them to the parts palette.

For example, go back to the application that queries for contractor, skill, and contract information. Consider the visual part that displays information about an individual contractor. Along with the information supplied for each contractor is the contractor's name and address. You can make this common address element into a reusable, composite visual part that looks like this:

Developing Applications



A screenshot of a reusable address form. It contains four input fields: 'Street' (a single line), 'City' (a single line), 'State' (a small dropdown menu), and 'Zip Code' (a single line). The fields are arranged in a compact, rectangular layout.

Figure 21. Reusable Address Part

You can then reuse the same address part in visual parts for other applications you create. By reusing parts, you increase productivity. For example, you only have to lay out the form once. In addition, because the **State** and **Zip code** field formats differ from the default formats used by the other fields, you only have to set the formats for these fields once, as well.

You also get the potentially more important benefit of improved maintenance. For example, if an error is discovered, you only have to fix it in one place. Or later, if your business decides to start using an **Auxiliary address** field, you would only have to make the change once to your visual address part.

Likewise, if the business grows into worldwide sales, then you could change **State** to **State/Region**, and add a **Country** field (provided you are using a database that can accommodate such changes). Having to make these changes only once saves time.

The next section, “Using Inheritance,” suggests more ways to reuse parts.

Using Inheritance

As you are scanning your application looking for common elements, do not overlook the possibility of reusing an entire part. You can reuse both nonvisual and class interface parts.

Many times, you need parts that are similar to ones you have already built. Inheritance gives you an easy way to make new parts from the ones you already have. Reusing existing parts through inheritance saves most of the work that would usually be required to create a new part.

Using inheritance to build a derived part

Suppose you created a new nonvisual part named **Person**. You could use inheritance to build a derived **Contractor** part that inherits the **Person** part’s features and has other features of its own. The following example assumes you have already created a nonvisual **Person** part.

1. Begin in the Visual Builder window by selecting **Part→New**.

Developing Applications

The following window is displayed:

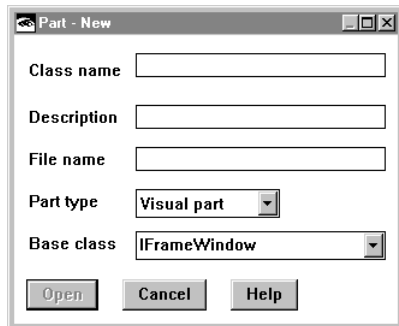


Figure 22. Part-New Window

2. Type Contractor in the **Class name** field.
3. Change the part type to Nonvisual part in the **Part type** field.
4. Enter Person in the **Base class** field.

Your new Contractor part now inherits from your Person part. Your window looks like the following:

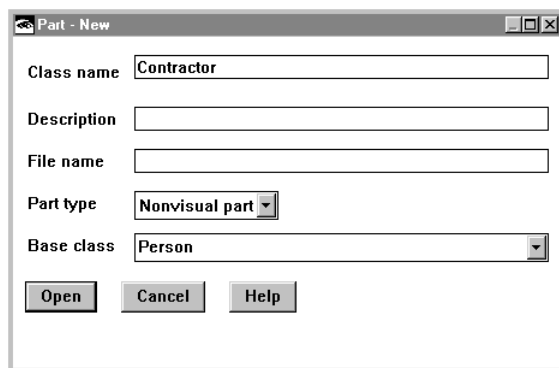


Figure 23. Part-New Window with Person Part as Base Class

5. Select **Open** to create the part and open the Part Interface Editor.
When the Part Interface Editor opens, it displays the **Attribute** page.
6. Add new attributes that a contractor might have, such as *startDate*, *endDate*, and *activeStatus*.

Developing Applications

The next section, “Using Abstract Classes” on page 88, suggests ways to factor out common classes and member functions so you can better reuse the code that you write.

Using Abstract Classes

The technique of scanning your application for common elements applies as much to code as it does to visual parts.

As you are writing member functions that enhance your visual parts, look for common member functions. These common member functions indicate perfect opportunities for reuse and can possibly be placed in an abstract class.

Abstract classes are used for factoring out common behavior. They are not intended to define any particular real part. Therefore, never represent an abstract class as a part on the free-form surface of the Composition Editor. Instead, use the abstract class as the base class, or as an ancestor of the base class, of nonvisual or class interface parts that you create and place on the free-form surface.

Abstract classes are repositories for common member functions. The similarities between your various parts determine which member functions are candidates to be grouped together under a common abstract class.



If you move a member function from a part to an abstract class, remove that member function from the interface of the part from which it was removed. You are not required to do this, but maintaining your application is easier if each part properly reflects only the member functions it contains in its part interface.

All of the benefits of reuse are achieved, namely increased productivity and reduced maintenance, from only having to write and maintain the member functions in one place.

The next section, “Keeping Your Components Small” on page 89, emphasizes a common theme in designing effective Visual Builder applications.

Keeping Your Components Small

As the topics in this chapter have probably made clear by now, a primary goal of any object-oriented environment, including Visual Builder, is reuse, for which keeping your components small is fundamental. The smaller you make the components of your applications, the more opportunity you have to take advantage of reuse.

The following table provides the numbers, based on C++ research, that you should aim for in creating your classes, member functions, and data members.

Criteria	Optimum number	Reason
Average lines per member function	Fewer than 24	A higher average generally indicates poor object-oriented design and a tendency toward function-oriented programming.
Average member functions per class	No more than 20	Higher averages indicate you can distribute your member functions among more classes.
Average data members per class	No more than six	Higher averages indicate you can distribute your data members among more classes.
Average number of base classes per class	No more than six	Inheriting from too many classes indicates too few member functions in each class. This can cause performance and size problems because the more base classes from which your classes inherit, the slower your compiled code runs and the larger it becomes.

These numbers generally depend on the complexity of the class. Less complex classes that take full advantage of prewritten reusable parts typically have fewer member functions per class. Also, the number of data members depends on how data-intensive the classes are. However, many of the member functions and data members you need can be inherited by default from classes provided by Visual Builder.

Applying the design strategies covered in these sections and gaining experience should help you achieve these averages.

Developing Applications

Designing Parts for the OASearch Application

A sample application is used in this part of the book to show you how to build a simple GUI application. You can find the nonvisual parts in `oanonvis.vbb`. All visual parts are in `oawin.vbb`. For in-depth information about designing your own parts, refer to *Building VisualAge for C++ Parts for Fun and Profit*.

This application, called OASearch, serves recruiters for a fictitious technical contracting services firm, called Opportunities Abound (OA). Customers call OA to request temporary help; recruiters use the application to match skill requirements with available contractors. When matches are made, the recruiters enter contract information into the application.

Using a Simulated Database

This sample application does not require the installation of a database program. Instead, the application uses the `IProfile` class to create and maintain a simple database. For more information about the `IProfile` class, refer to the *IBM Open Class Library Reference*.

Designing Nonvisual Parts

When designing your own application, take extra time to think through the nonvisual parts. One way to minimize rework during design is to use index cards, putting the data and tasks of each part or class on a separate card. This method enables you to throw away one design in favor of another, rearranging the cards until you are confident of your design.

The OASearch application uses the following nonvisual parts to hold or manage data:

- `OACContractor`, which represents OA employees
- `OACContract`, which represents OA contracts
- `OASkill`, which represents marketable skills held by OA's employees
- `OASkillBase`, which controls traffic between the `OASkillView` window and the skill database. This is necessary because of the one-to-many relationship that exists between `OACContractor` and `OASkill` instances. That is, many contractors could hold the same skill and a single contractor could hold many skills.
- `SkillList`, a part instance of type `IVSequence*`. For more information about sequences, refer to the *Visual Builder Parts Reference*.

The OASearch part found in `oanonvis.vbb` is not used in the OASearch sample as it is shipped, but you can use it as an example when packaging your application.

The OACContractor Part

When you design a Visual Builder part, you are designing a C++ class. The part interface corresponds to the part interface of the underlying class. The class can also have member functions (public, protected, or private) that do not exist in the part interface.

OACContractor Attributes

The OACContractor part has the following attributes:

- *lastName*, the contractor's last name
- *firstName*, the contractor's first name
- *middleInitial*, the contractor's middle initial
- *homeStreet*, the contractor's home street address
- *homeCity*, the contractor's home city
- *homeState*, the contractor's home state or province
- *homeZip*, the contractor's home postal code
- *phoneNumber*, the contractor's daytime phone number
- *startDate*, the date on which the contractor started working at Opportunities Abound
- *endDate*, the date on which the contractor stopped working at Opportunities Abound
- *activeStatus*, whether the contractor is interested in a new position
- *currentContract*, the contractor's current assignment (if any)
- *contractorID*, the contractor's database identifier
- *IDvalid*, a validation flag for the *contractorID* attribute

All attributes are of type *IString*, except for the *activeStatus* and *IDvalid* attributes, which are of type *Boolean*. The *contractorID* attribute is derived from the contractor's name attributes.

To enable you to use these attributes, the contractor part must also contain public get and set member functions. By convention, the names of these member functions follow a pattern determined by the name of the attribute. For the *IString* attribute *lastName*, the get member function is called *lastName*, and the set member function is called *setLastName*. For the *Boolean* attribute *activeStatus*, the get member function is called *isActiveStatus*, and the set member function is called *enableActiveStatus*.

To retrieve or update contractor information from the database, the OACContractor part contains the following actions:

- *getContractor*
`OACContractor & getContractor();`
- *parseName*

Developing Applications

```
OACContractor & parseName(const IString & aName);
```

- *putContractor*
OACContractor & putContractor();
- *refreshID*
OACContractor & refreshID();

The *parseName* action parses an input string to set the *firstName*, *middleInitial*, and *lastName* attributes. The *getContractor* and *putContractor* actions use the derived attribute *contractorID* to retrieve and store contractor information in the database. The *refreshID* action checks for all name attributes and resets the *contractorID* attribute. The feature code for this part appears in *contrctr.cpv*.

For a look at the visual part that OACContractor supports, see “The Opportunities About Contractor Information Window” on page 96.

The OACContract Part

The OACContract part contains the following attributes:

- *accountNum*, the contract’s account number
- *companyName*, the name of the client company
- *projectMgr*, the client company’s representative
- *deptName*, the department within the client company that needs a position filled
- *positionTitle*, a description of the position to be filled
- *startDate*, the first day of work at the client company
- *endDate*, the last day of work at the client company
- *currContractor*, the identifier of the contractor filling the position

It also contains the following actions:

- *getContract*
OACContract & getContract(const IString & anAccountNum);
- *putContract*
OACContract & putContract(const IString & anAccountNum);

All attributes are of type IString. The feature code for this part appears in *contract.cpv*.

For a look at the visual part that OACContract supports, see “The Opportunities About Contract Information Window” on page 98.

The OASkill Part

The OASkill part contains the following attributes:

- *skillName*, a description of the skill required
- *yearsExp*, the number of years of experience needed
- *contractorID*, the identifier of the contractor holding this skill
- *key*, an index used to construct the database key

All attributes are of type IString. The feature code for this part appears in skill.cpv.

For a look at the visual part that OASkill supports, see “The Opportunities Abound Skill Information Window” on page 98.

The OASkillBase Part

The OASkillBase part contains no attributes but has the following actions:

- *getSkills*

```
OASkill & getSkills(const IString & aSkillName, IVSequence <OASkill*> * aList);
```

This action retrieves skill information from the database, instantiates OASkill objects from retrieved information, and adds pointers for the new objects to the specified list of skills.

- *putSkill*

```
OASkill & putSkill(OASkill* aSkill);
```

This action stores skill information in the database.

The feature code for this part appears in skillb.cpv.

For a look at the visual part that OASkillBase supports, see “The Opportunities Abound Skill Information Window” on page 98.

The OASearch Part

The OASearch part contains no local features; its only purpose is to set a resource library for the OASearch application. To distribute your finished application to customers that do not have VisualAge for C++ installed, you need a part similar to this one. You do not need this part to build the OASearch application as it is shipped by IBM; this part is provided only as an example. For more information about how to use this part, see “Resetting Your Application's Main Resource Library” on page 270

Developing Applications

Designing Visual Parts

Although the OASearch application was designed to illustrate variety more than efficiency, consider the following design decisions:

- The Opportunities Abound Databases window (part OAMain) is the integration point for the application. All variables found in the other composite parts are resolved in OAMain. All parts except the OASkill search results (SkillList) are instantiated here.
- The connections between visual parts are minimized with the use of promoted variables to pass data. Inside each visual part, attribute-to-attribute connections load data from the variables into the windows.
- All views use multicell canvases to hold the primitive visual parts. They are as follows:
 - The Opportunities Abound Databases window
 - The Opportunities Abound General Information window
 - The query windows
 - The Opportunities Abound Contractor Information window
 - The Opportunities Abound Contract Information window
 - The Opportunities Abound Skill Information window

The Opportunities Abound Databases Window

The welcome window for OASearch appears in Figure 24.

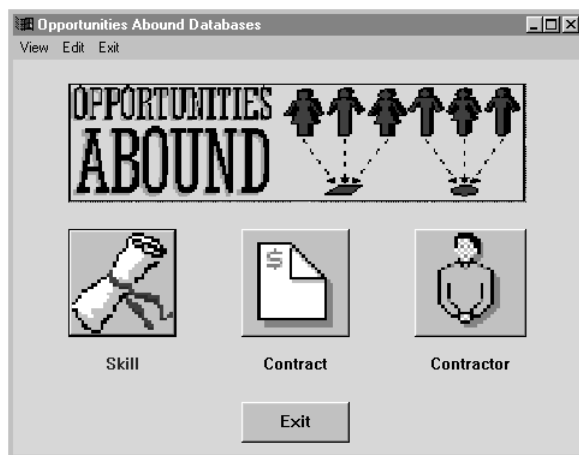


Figure 24. The Opportunities Abound Databases Window (OAMain)

This first window to appear is the *primary view*; the part containing this window, OAMain, is the *main part*. To view information, recruiters can use the **View**

Developing Applications

submenu or press one of the three graphic push buttons. To enter information about new contracts or contractors, they can use the **Edit** submenu.

This window shows use of the following:

- Menu bar
- Embedded multicell canvas
- Icons as resources
- Object factory parts
- Help support
- Variables used with object factory parts

The Opportunities Abound General Information Window

When a user selects **General information** from the **View** submenu, the Opportunities Abound General Information window appears, as shown in Figure 25.



Figure 25. The Opportunities Abound General Information Window (OAGenInfo)

This window shows use of the following:

- IViewport* with multicell canvas

This window uses an IViewport* part as the client. An IMultiCellCanvas* part within the IViewport* holds the IStaticText* parts. In the compiled application, the IMultiCellCanvas* is resized whenever the window is resized.

Developing Applications

The Query Windows

When recruiters select a graphic push button or **View** menu item from the Opportunities Abound Databases window, a query window appears, as shown in Figure 26.

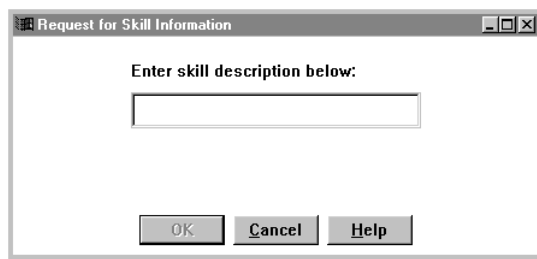


Figure 26. The Request for Skill Information Window (OAQuerySkill)

The OAQuerySkill part contains this window. Two other parts, OAQueryContract and OAQueryContractor, hold other query windows.

When a user enters a value in the query window and selects the **OK** push button, the query window closes and the corresponding information window appears. If the requested value is not found, a message box appears. Users can also get help or return to the Opportunities Abound Databases window.

The query windows show use of the following:

- Modal window display
- Help enablement
- Promoted variables and features

The Opportunities Abound Contractor Information Window

When the requested contractor is found in the contractor database, the Opportunities Abound Contractor Information window appears, as shown in Figure 27 on page 97.

Developing Applications

The screenshot shows a software window titled "Opportunities Abound Contractor Information". On the right side, there is a vertical sidebar with three buttons labeled "Contact", "History", and "Skills". The main content area is divided into three sections. The "Name" section has two text input fields for "Last" and "First", followed by a small dropdown menu labeled "MI". The "Home Address" section has three text input fields for "Street", "City", and "Postal Code", followed by a small dropdown menu labeled "State". Below these is a single text input field for "Phone". At the bottom of the main area are four buttons: "Refresh", "Save", "Clear", and "Cancel". At the very bottom of the window, there is a label "Vital statistics" with two small arrow buttons (left and right) next to it.

Figure 27. The Opportunities Abound Contractor Information Window (OAContractorView)

The application retrieves the records matching the requested contractor identifier, sets an OAContractor* part, and displays the values in this window. The **Contact** page contains information about how to get in touch with the contractor. The **History** page contains information about the contractor's work history. The **Skills** page is provided for users of the sample application to update skills that a hypothetical contractor has.

This window shows use of the following:

- INotebook* part
- IVBLogicalAnd sample part
- Different multicell canvas layouts
- Event-to-member function connection
- Event-to-custom logic connection
- Variables used as placeholders
- Promoted features
- Torn-off attribute

Developing Applications

The Opportunities Abound Contract Information Window

When the requested contract is found in the contract database, the Opportunities Abound Contract Information window appears, as shown in Figure 28.



Figure 28. The Opportunities Abound Contract Information Window (OAContractView)

The application retrieves the records matching the requested contract account number, sets an OAContract* object, and displays the values in the window. This window shows basic multicell canvas layout.

A variant of the OAContractView part, ContractView, appears in contract.vbb. This version of the Opportunities Abound Contract Information window uses an ICanvas* part instead of a multicell canvas. It is formatted for an XGA/8514 display.

The Opportunities Abound Skill Information Window

When the requested skill is found in the skill database, the Opportunities Abound Skill Information window appears, as shown in Figure 29 on page 99.

Developing Applications

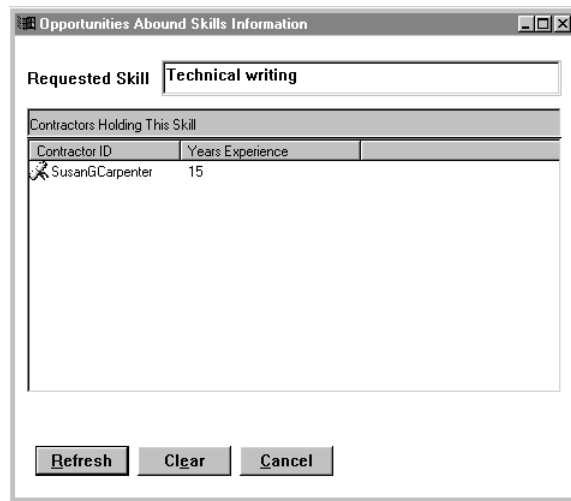


Figure 29. The Opportunities Abound Skill Information Window (OASkillView)

The application retrieves all records matching the requested search description, instantiating an OASkill object for each pair of records (contractor and years of experience) found. Each new OASkill object is put into a list and, from there, into the container shown in the window. Recruiters can search on other skills without leaving this window by typing the new skill description into the **Requested skill** field and selecting the **Refresh** push button.

This window shows use of the following:

- IVBContainerControl*
- IContainerColumn*
- IVSequence*

How the Sample Application Was Built

The OASearch application was developed in the following order:

1. The major nonvisual parts were designed.
2. The major visual parts were designed.
3. The nonvisual parts were built.
4. The view parts were built.
5. The query parts were built.
6. The OAMain and OAGenInfo parts were built.
7. Code for all parts was generated at the same time from the Visual Builder window.

Developing Applications



Chapter 7. Creating Nonvisual Parts

This chapter describes how to define your own parts. For this example, we will create a nonvisual part, OACContractor, because it illustrates the process better than a visual part.

For information about using the Composition Editor to create visual parts, see Chapter 8, “Learning to Use Parts” on page 113. For more information about using the Part Interface Editor to define part interfaces, see “The Part Interface Editor” on page 56.

Creating a part

You create a part by doing the following:

1. Design the part. (See “Designing Parts for the OASearch Application” on page 90.)
2. Define the part interface, either through the Part Interface Editor or by importing a part information file. (See “Defining the Part Interface.”)
3. Add code to your part. You can use C++ code written outside of Visual Builder, or you can generate feature code in Visual Builder and modify it. (See “Adding Code to Your Part” on page 105.)

Using existing C++ code

If you have previously existing C++ code that you would like to use in your part, the following process is probably more efficient:

1. Design the part.
2. Define the part interface using part information files.

Defining the Part Interface

When you are satisfied with your part’s design, you are ready to define the part interface to Visual Builder, as follows:

1. Define the attributes of your part.

A part’s attributes typically correlate to the class’ data members and can additionally include *derived attributes*, or attributes that are determined based on the value of other attributes.

Developing Applications

2. Define the member functions that get or set the value of those attributes.
3. Define any actions that you want the part to be able to do.
A part's actions correlate to the class' public member functions.
4. Specify the event notification identifier used to signal a change in the value of each attribute.

You can define the part interface in either of the following ways:

- Use the Part Interface Editor. In this way, you create the part from the Visual Builder window and then enter each feature of the part interface individually using the pages of the Part Interface Editor.
- Use a part information file. In this way, you encode part information in a file and then create the part and its interface by importing the information into Visual Builder. This method might be more efficient if C++ code already exists for your part. See Chapter 18, "Using Existing C and C++ Code with Visual Builder" on page 287 and refer to *Building VisualAge for C++ Parts for Fun and Profit* for information about creating part information files.

Defining the Part Interface Using the Part Interface Editor

To define the part interface using the Part Interface Editor, do the following:

1. Open a new nonvisual part called OACContractor.
The Part Interface Editor appears.
2. On the **Attribute** page, do the following:
 - a. Type `lastName` in the **Attribute name** text-entry field and select the **Add with defaults** push button.

Your Part Interface Editor looks like Figure 30 on page 103.

Developing Applications

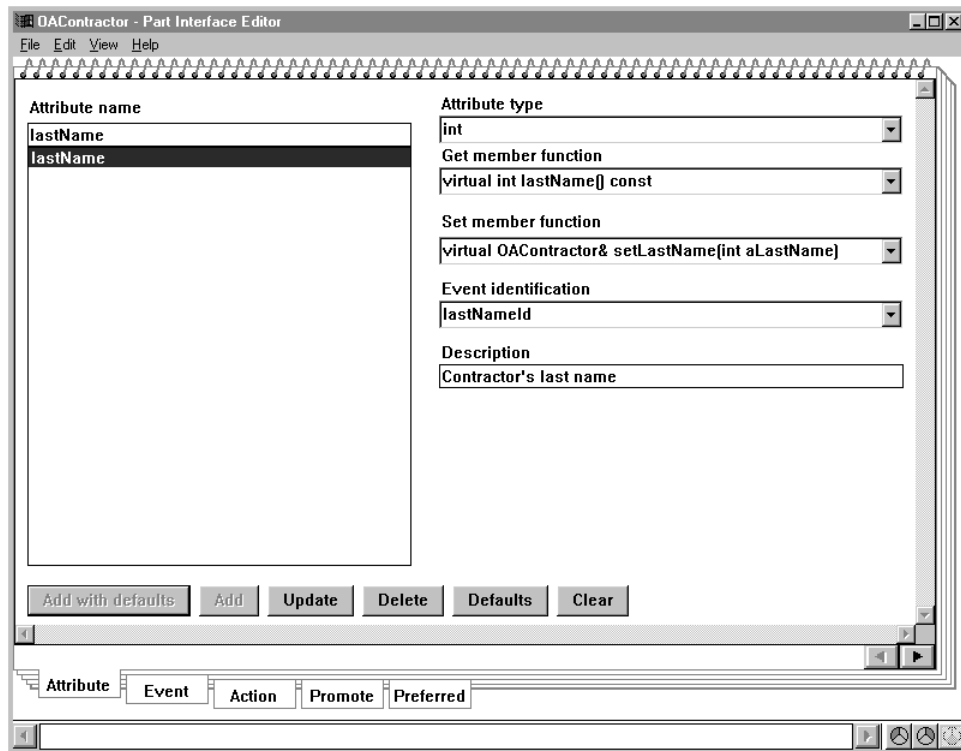


Figure 30. Creating the *lastName* Attribute

Notice that several default values are displayed:

- The default **Attribute type** is `int`. This means that the *lastName* attribute has a data type of `int`.
- The default **Get member function**, `lastName`, matches the *lastName* member function of your design. When your *OACContractor* part is queried for the value of its *lastName* attribute, it executes the *lastName* member function to retrieve the contractor's last name.
- The default **Set member function** is `setLastName`. When your *OACContractor* part is passed the name that has been assigned to the contractor, it uses the `setLastName` member function to store the value in its *lastName* attribute.
- The default **Event notification** is `lastNameId`. When a value is stored in the *lastName* attribute using the `setLastName` member function, the *OACContractor* part signals this event to let other parts know that the value has changed. For example, if there is an attribute-to-attribute

Developing Applications

connection between *lastName* and a static text control's string value, the static text control part is informed so that it displays the latest value.

- If needed, you can explain more about the attribute in the **Description** field.
- b. Change any of the default values, if needed.
For this example, do the following:
 - 1) Change the default attribute type to `IString`.
You can do this by doing either of the following:
 - Type the value directly into the entry field.
 - Select the drop-down list box arrow to the right of the entry field and select `IString` from the list.
 - 2) Edit the **Get member function** and **Set member function** fields, changing `int` to `IString`. This causes these member functions to accept `IString` as a parameter instead of `int`.
 - 3) Select the **Update** push button to save the changes you just made.
- c. Select the **Clear** push button to clear the fields before adding the next attribute.
- d. Repeat the previous steps for the contractor's *firstName*, *middleInitial*, *homeStreet*, *homeCity*, *homeState*, *homeZip*, *startDate*, *endDate*, *currentContract*, and *phoneNumber* attributes. For these attributes, specify the `IString` attribute type before selecting the **Add with defaults** push button in the first step. This prevents you from having to change `int` to `IString` as you did previously.
- e. Define the *contractorID* attribute as `IString`, with the following get member function:

```
OACContractor & contractorID();
```

This attribute is derived from other attributes; we will add the code for this later. You do not want users to set this attribute directly, so do not include a `setContractorID` member function.
- f. Define the *activeStatus* and *IDvalid* attributes as `Boolean`. Note that the default get member functions and set member functions follow a different format than the ones for `IString` attributes.
- g. To define the actions for `OACContractor`, select the **Action** page.
- h. Type `putContractor` in the **Action name** field and the following into the **Action member function** field:

```
OACContractor & putContractor();
```

Developing Applications

- i. Select the **Add** push button; then select **Clear**.
- j. Repeat for the *getContractor*, *parseName*, and *refreshID* actions. At this point, the **Action** page appears as shown in Figure 31.

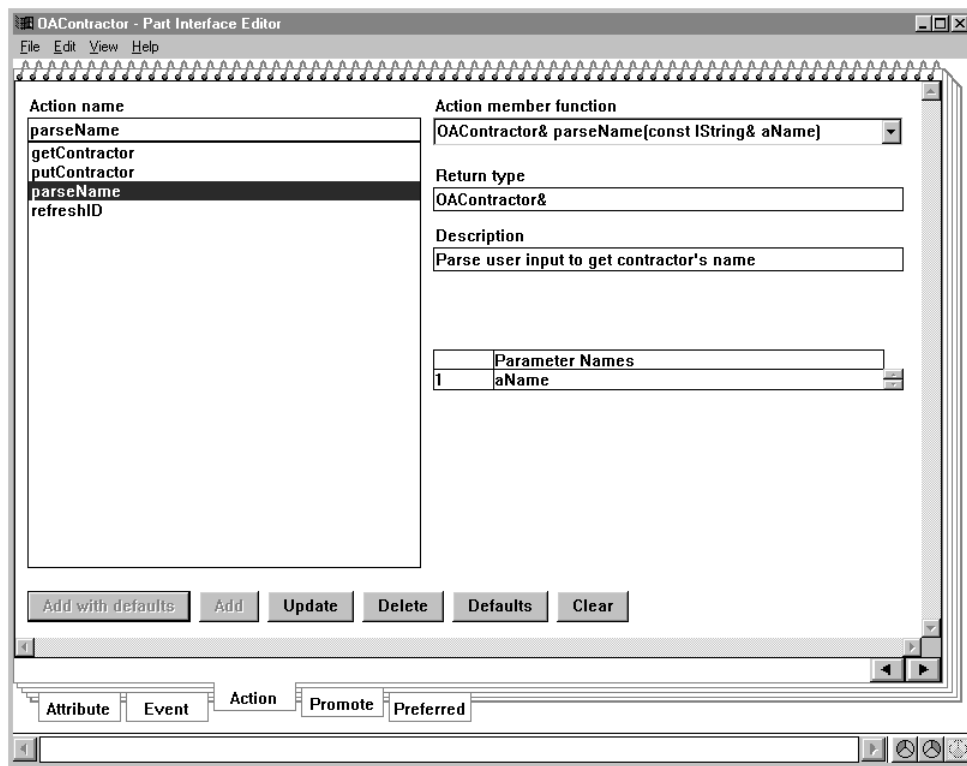


Figure 31. Adding Actions to the Part Interface Editor

- k. Select **Save** from the **File** pull-down menu to save your work.

The next step is described in “Adding Code to Your Part.”

Adding Code to Your Part

Once you have specified the part interface for your part, you are ready to add the code to make the part work. Adding code involves the following tasks:

1. Generating feature code. If you already have C++ code for your part and have imported the part information, this step is not necessary.
2. Modifying the feature code.
3. Adding code created outside Visual Builder if it already exists.

Developing Applications

Generating Feature Code

If code does not already exist for your part, you can use Visual Builder to generate feature code. This feature code is based on the part interface you defined earlier. For most attributes, the generated feature code is sufficient to define get member functions, set member functions, and event notifications. For actions, you have to modify the feature code to add the function or logic you want your part to perform.

If you also use Visual Builder to generate the source code for your parts, you will find it helpful to generate feature code separately. Each time you generate source code, Visual Builder replaces the existing files with new ones because there is no need for you to modify these files.

The files that you usually need to modify are the feature code files. These are the files in which you write the code to tell your application how to perform the actions that you create in the Part Interface Editor. Each time you generate feature code, Visual Builder appends the newly generated code to the end of each existing feature code file. This is done so that you will not lose any code that you have written.


In addition, you do not have to generate new code for all of your features each time you need code for a feature. You can create a new feature in the Part Interface Editor and then generate feature code just for the new feature. Visual Builder appends the code for the new feature to the end of the existing files so that you can modify it.

Note: If you regenerate code for a feature, be sure to remove the previous code for that feature from the file to prevent compilation errors or unwanted results.

Generating feature code for the OACContractor part

To generate feature code for the OACContractor part, follow these steps.

Note: This example is a continuation of the example in the preceding section using the OACContractor part, which should be open in the Part Interface Editor.

1. Switch to the Class Editor by selecting the  icon in the lower-right corner of the window.
2. Specify the .hvp and .cpv files for Visual Builder to use for the feature code for the OACContractor part's attributes and actions by filling in the following fields:

User .hvp file contrctr.hvp

User .cpv file contrctr.cpv

The Class Editor with the appropriate fields filled in is shown in Figure 32 on page 107.

Developing Applications

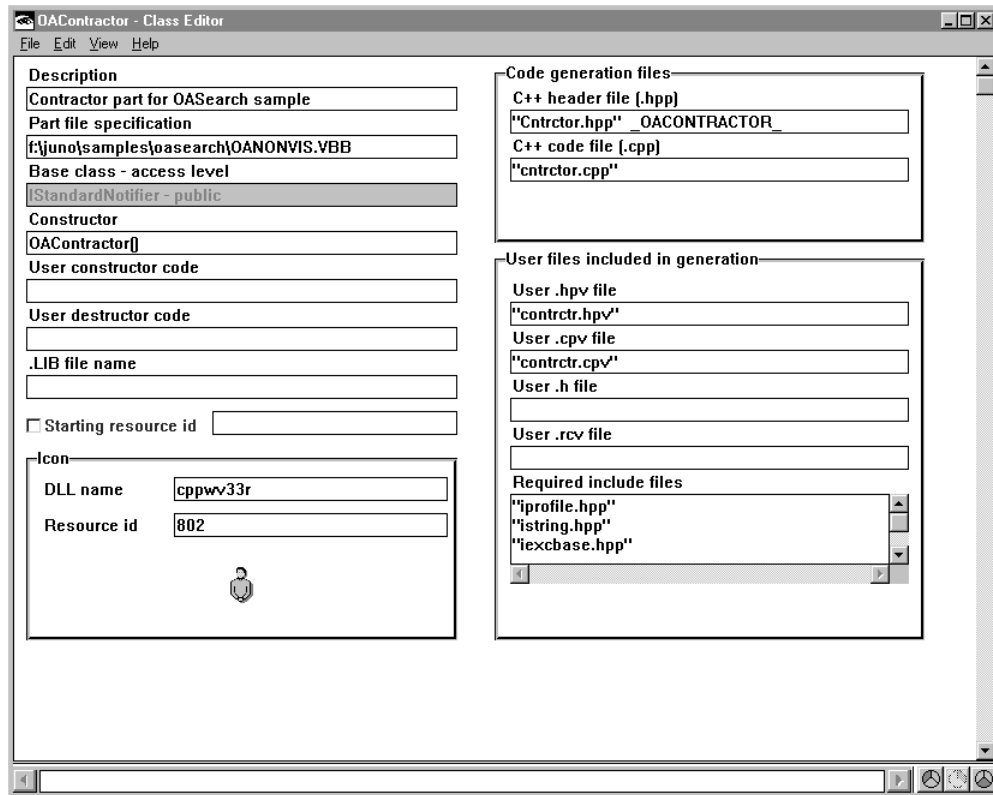


Figure 32. Specifying File Names in the Class Editor

3. Select **File**→**Save and generate**→**Feature source**.

The Generate Feature Source Code window appears, as shown in Figure 33 on page 108.

Developing Applications

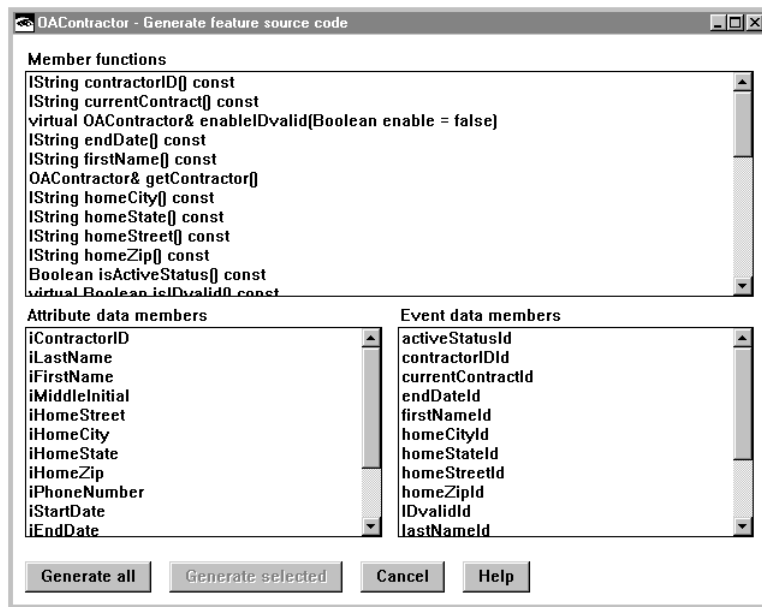


Figure 33. The Generate Feature Source Code Window

4. Generate the feature code using one of the following methods:
 - Select the **Generate all** push button to generate feature code for member functions and data members.
 - Select the appropriate member functions and data members from the **Member functions**, **Attribute data members**, or **Event data members** list boxes. Then, select the **Generate selected** push button.

The first time you will probably want to generate code for all of your features. Then, you can generate code for other features as you add them.

For this example, select the **Generate all** push button.

The generated feature code is stored in the files you specified in the Class Editor, `contrctr.hpv` and `contrctr.cpv`. If these files already exist, the code you just generated is appended to the ends of these files.

For most parts, you must modify the code to make your part fully functional.

Modifying the Generated Feature Code

After you have Visual Builder generate the feature code, review it. Does it do what you need it to do? If not, modify the code. The `OACContractor` part needs the following changes:

- Modified code for the *getContractor* action
- Modified code for the *putContractor* action
- Modified code for the *parseName* action
- Modified code for the *refreshID* action
- Added code to set the *activeStatus* attribute
- Added code to set the *contractorID* attribute

Modifying Code for the *getContractor* Action

This version of the contractor application uses an IProfile-based database to store information about the contractors. Before the `OACContractor` part can use this database, we must complete the *getContractor* and *putContractor* actions and modify the *activeStatus* attribute. The following example shows the code generated for the two actions:

```
OACContractor & OACContractor::getContractor()
{
    return *this;
}

OACContractor & OACContractor::putContractor()
{
    return *this;
}
```

To modify feature code, edit the code files using the syntax text editor provided by VisualAge for C++ or your favorite editor. Code for the *getContractor* action is found in `contrctr.cpv`.

The *getContractor* action checks for one possible error condition. If it occurs, the action throws an exception. Otherwise, the action retrieves data by contractor identifier from the `contrctr.ini` file.

In response to these exceptions, the `OACContractorView` part shows a message box. More about the contractor view part appears in “Constructing a GUI: the `OASearch` Application” on page 157. For information about how to handle exceptions visually, see “Passing Exceptions to Message Boxes” on page 203.

Developing Applications

Modifying Code for the putContractor Action

Code for the *putContractor* action is found in *contrctr.cpv*. The *putContractor* action checks for one possible error condition. If it occurs, the action throws an exception. Otherwise, the action stores data by contractor identifier in the *contrctr.ini* file.

In response to this exception, the *OACContractorView* part shows a message box. More about this part appears in Chapter 12, “Constructing Containers and Notebooks” on page 231. For information about how to handle exceptions visually, see “Passing Exceptions to Message Boxes” on page 203.

Modifying Code for the parseName Action

The *parseName* action parses input text from the Request for Contractor Information window and sets the contractor’s *firstName*, *middleInitial*, and *lastName* attributes. The code for this action is found in *contrctr.cpv*.

Modifying Code for the refreshID Action

The *refreshID* action checks the current values of the *firstName*, *middleInitial*, and *lastName* attributes and sets the contractor’s *contractorID* attribute. The code for this action is found in *contrctr.cpv*.

Adding Code to Set the activeStatus Attribute

You must add a member function to pass string data to the one Boolean attribute, *activeStatus*. All data in the simple database is stored as strings, but the set member function for the *activeStatus* attribute takes a Boolean as a parameter. Because only *OACContractor*’s two actions use this member function, it is not necessary to add this to the part interface for *OACContractor*. However, you must add the member function prototype to the *contrctr.hpv* file.

The following example shows both the generated feature code and the added code in *contrctr.cpv*:

```
Boolean OACContractor::isActiveStatus() const
{
    return iActiveStatus;
}

OACContractor & OACContractor::enableActiveStatus(const Boolean enable)
{
    if (iActiveStatus != enable)
    {
        iActiveStatus = enable;
        notifyObservers(INotificationEvent(OACContractor::activeStatusId, *this));
    } // endif
    return *this;
}

// Start Boolean string enabler
```

Developing Applications

```
OACContractor & OACContractor::enableActiveStatus(const IString & status)
{
    Boolean tempBoolean = iActiveStatus;
    if (status = "yes") iActiveStatus = true;
    if (status = "no") iActiveStatus = false;
    if (tempBoolean != iActiveStatus)
    {
        notifyObservers(INotificationEvent(OACContractor::activeStatusId, *this));
    }
    return *this;
}

// End Boolean string enabler
```

`iActiveStatus` is the private data member that corresponds to the *activeStatus* attribute.

The `notifyObservers` function signals that the value of the *activeStatus* attribute has changed. For more information about notification, see *Building VisualAge for C++ Parts for Fun and Profit*.

Adding Code to Set the *contractorID* Attribute

Now, add code to derive the value of the *contractorID* attribute. The following example shows the code as modified in `contrctr.cpv`:

```
OACContractor & OACContractor::setContractorID()
{
    IString tempString = iFirstName+iMiddleInitial+iLastName;
    if (iContractorID != tempString)
    {
        iContractorID = tempString;
        notifyObservers(INotificationEvent(OACContractor::contractorIDId, *this));
    } // endif

    // Test for string with nontrivial content ; needed for clear() call
    if (iContractorID.length() > 5)
        enableIDvalid(true);

    return *this;
}
```

This action sets the *IDvalid* attribute to true if the resulting value of *contractorID* is valid.

Developing Applications

Adding Code Created Outside Visual Builder

To include previously existing member function code with generated class declarations, do the following:

1. Change the file extension of .cpp files to .cpv.
2. Change the file extension of .hpp files to .hvp.
3. Change the file extension of any .rc files to .rcv.
4. Add these files to the Class Editor for the appropriate part.

Developing code outside Visual Builder

If you prefer not to use Visual Builder to develop the code for a nonvisual part but you want to be able to use it in the Composition Editor, you can do the following:

1. Write the code.
2. Compile it into a DLL.
3. Define the part interface using a part information file.
4. Import the part information file.

For more information on part information files, see “Defining the Part Interface Using Part Information Files” on page 287.



Chapter 8. Learning to Use Parts

This chapter provides general information on using parts, and then shows you how some of this information can be applied to an actual application—the OASearch sample application that was introduced in “Designing Parts for the OASearch Application” on page 90.

With the Composition Editor you can visually construct an application by placing parts on the free-form surface and making connections between them.

With connections, you can construct the interactions between the parts of the application. For information about connecting parts to each other, see Chapter 9, “Learning to Use Connections” on page 163.

In Visual Builder you use the following kinds of parts:

- Visual parts, such as `IPushButton*`, `IListBox*`, and `IEntryField*`, to construct the graphical user interface (GUI) of the application.
- Nonvisual parts, such as the sample `ICustomer` and `ICompany` parts, to represent the data or objects manipulated by the application.
- Class interface parts, such as `IDate*` and `ITime*`, to represent data or objects, also. The difference between nonvisual parts and class interface parts is that nonvisual parts can notify other parts when an event occurs or an attribute value changes; class interface parts, however, do not have this notification capability.

Working with Parts in the Visual Builder Window

The topics in this section describe how to perform various actions on part classes from the Visual Builder window.

Displaying Part Names

To display the names of the parts in a part file, in the **Loaded files** list box in the Visual Builder window, select the part file whose parts you want to see. The names of the parts contained in the part file that you selected are displayed in the Visual Builder window. Visual parts are displayed in the **Visual parts** list box; nonvisual parts and class interface parts are displayed in the **Nonvisual parts** list box.

Once part names are displayed, you can perform actions on them, such as opening or deleting them. If you need information about loading part files, see “Loading Part Files” on page 31.

Developing Applications

Figure 34 on page 114 shows the Visual Builder window with the names of the parts in the vbbase.vbb file displayed:

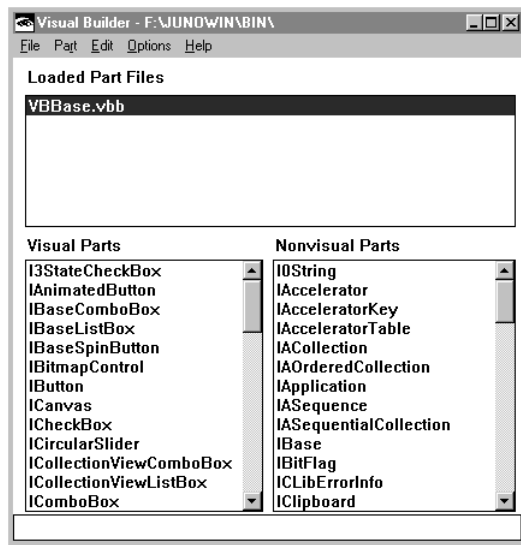


Figure 34. Visual Builder Window with vbbase.vbb Parts Displayed

Selecting All Parts

To select all of the parts in the selected part files, select **Edit→Select all parts**.

At this point, you can review the list to see if you want to deselect any of the parts. You can do so by pressing the Ctrl key and clicking on the part name with mouse button 1.

Deselecting All Parts

To deselect all of the parts in the selected part files, select **Edit→Deselect all parts**.

Importing Part Information

Using any text editor, you can create files called *part information files*, which are used to import existing C++ classes into Visual Builder as parts. Part information files are normally recognizable by their .vbe extension.

The **Import part information** function loads part information files so that you can use the parts specified in those files in Visual Builder.

To import part information, do the following:

1. Select **File→Import part information**. The following window is displayed:

Developing Applications

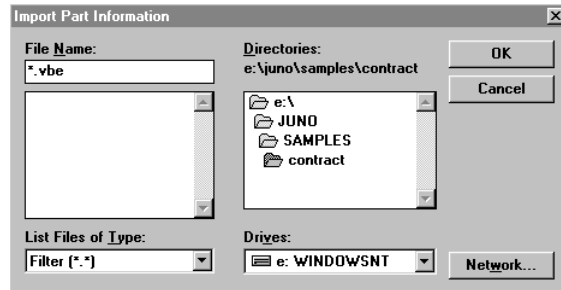


Figure 35. Window for Importing Part Information Files

2. Select the part information file that you want to import.
3. Select the **OK** push button. The part information in the part information file is imported. Any visual parts that the part information file contains are displayed in the **Visual parts** list box, and any nonvisual parts and class interface parts it contains are displayed in the **Nonvisual parts** list box. In addition, one or more part files might be created.

For more information about using and creating part information files, see the following:

- “Defining the Part Interface Using Part Information Files” on page 287
- *Building VisualAge for C++ Parts for Fun and Profit*

In addition, Visual Builder provides the following .vbe files that contain sample parts. You can look at these files in an editor and import them into Visual Builder using the instructions provided in this section.

vbcc.vbe

Contains sample parts based on the IBM Collection Class Library. You must import this file to use the parts.

vbsom.vbe

Contains Direct-to-SOM parts that you can use in Visual Builder. You must import this file to use the parts.

Developing Applications

Exporting Part Information

Just as you can import part information from an existing part information file, you can also export part information for any Visual Builder part into a .vbe file. This lets you share nonvisual and class interface parts with other programmers. Part information files contain usage information that programmers who do not have access to the part file (.vbb) can use.

To export part information, do the following:

1. Select the part or parts whose information you want to export in either the **Visual parts** list box, the **Nonvisual parts** list box, or both.
2. Select **Part→Export interface**. The following window is displayed:

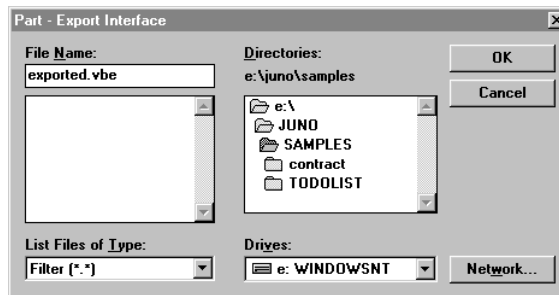


Figure 36. Window for Exporting Part Information Files

3. Type the name of the .vbe file in which you want the part information to be stored in the **Open file name** field. If you do not enter a file name, Visual Builder uses exported.vbe as the default file name.
4. Select the **OK** push button. The part information maintained by Visual Builder is exported to the file name you specified in the **Open file name** field.

For more information about using and creating part information files, see the following:

- “Defining the Part Interface Using Part Information Files” on page 287
- *Building VisualAge for C++ Parts for Fun and Profit*

Creating a New Part

This section provides only the basic steps for creating a new part. For a description of how to define a part once it has been created, see Chapter 7, “Creating Nonvisual Parts” on page 101.

To create a new part, do the following:

1. Select **Part→New**. A Part–New window is displayed, as follows:

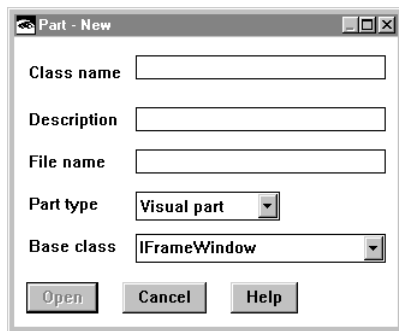


Figure 37. Part–New Window

2. Enter the name that you want to give to your part in the **Class name** field.
3. Enter a description of your part in the **Description** field. Visual Builder uses the description that you enter here in the following ways:
 - If you add the part that you create to the parts palette, Visual Builder displays the part’s description in the information area at the bottom of the Composition Editor when the part is selected.
 - If you export the information about the part to a .vbe file, the description is included with the other information about the part.
4. Enter the name of the part file in which you want Visual Builder to store the part in the **File name** field. If the file does not already exist, Visual Builder creates it for you. If you leave this field blank, Visual Builder creates a part file as follows:
 - If you are using the File Allocation Table (FAT) file system and have selected **Options→Default to FAT file names**, Visual Builder creates a part file whose name has no more than eight characters. Without this selection, Visual Builder attempts to create a part file whose name is the same as the name of your part, which causes an error if your part name has more than eight characters.

Developing Applications

Note: If you are using the FAT file system, we recommend that you always use part names and file names that have eight characters or fewer, even if you have selected the **Default to FAT file names** option. Otherwise, Visual Builder might use a file name for a .vbb file that is the same as one that already exists and write over the existing file.

- If you do not select **Options→Default to FAT file names**, the name of the part file is the same as the name of your part.
5. Select the type of part that you want to create in the **Part type** field. You can select one of the following:
 - Visual part
 - Nonvisual part
 - Class interface part
 6. Either keep the default class name provided by Visual Builder in the **Base class** field, change it, or delete it.


Note the following:

- A nonvisual part must have the `IStandardNotifier` class in its inheritance so it can exhibit the behavior required for all parts—a part interface (attributes, events, and actions). It must inherit this behavior from `IStandardNotifier`. Therefore, you cannot leave this field blank when creating a nonvisual part. The default base class for a nonvisual part is `IStandardNotifier`.
 - A visual part must have the `IWindow` class in its inheritance so it can inherit the visual behavior common to all windows, as well as part interface behavior, which `IWindow` inherits from `IStandardNotifier`. Therefore, you cannot leave this field blank when creating a visual part. The default base class for a visual part is `IFrameWindow`, which inherits from `IWindow`.
 - No inheritance is required for a class interface part. Therefore, you can leave the **Base class** field blank when creating a class interface part. The default base class for a class interface part is `IVBase*`.
7. Select **Open**. One of the following occurs:
 - If you are creating a visual part, the Composition Editor is displayed.
 - If you are creating a nonvisual part or a class interface part, the Part Interface Editor is displayed.
 8. Use the displayed editor to create your part.

Opening Parts

Use **Part→Open** to open parts that are already created. You must load the part file that contains a part before you can open the part.



Visual Builder uses the question mark icon, , to represent the unloaded parts on the free-form surface. If you open a part that contains other parts and the part files that contain the other parts are not loaded, Visual Builder displays this icon.

The question mark folder icon indicates that most of the information about the unloaded part is not available to Visual Builder. You can select connections between unloaded parts and other parts to see which features are connected, but the features are not available in the unloaded part's connection menu.

You should not make any changes to an unloaded part or generate any code when a part is not loaded.

If you open a part and see a question mark folder icon, do the following:

1. Close the part you just opened.
2. Load the part file that contains the unloaded part.
3. Reopen the part you previously opened. The question mark folder icon is replaced by the part's icon.

After loading additional part files, close and reopen editor windows. Otherwise, any question mark folder icons that appear in the Composition Editor are not changed to reflect the newly loaded part data.

If you want to add a bitmap to the folder, see “Specifying a Unique Icon for Your Part” on page 53.

The following instructions tell you how to open one part at a time or multiple parts simultaneously.

Opening one part

To open one part, do the following:

1. Find the name of the part that you want to open by scrolling through the appropriate list box in the Visual Builder window.

Note: If the list boxes in the Visual Builder window are empty or if you cannot find the part, the part file that contains the part you want to open is not selected or not loaded. See “Loading Part Files” on page 31 if you need help loading part files.

Developing Applications

The Visual Builder window with parts loaded from the part file is shown in Figure 34 on page 114.

2. Select the part you want to open.
3. Select **Part** on the menu bar.
4. Select **Open** in the pull-down menu. One of the following occurs:
 - If you are opening a visual part, Visual Builder displays the Composition Editor.
 - If you are opening a nonvisual part, Visual Builder displays the Part Interface Editor.



A quicker way to open an existing part is to double-click on the part name within the **Visual parts** or **Nonvisual parts** list box.

Opening multiple parts

To open multiple parts, do the following:

1. Find the name of the first part that you want to open by scrolling through the **Nonvisual parts** and **Visual parts** list boxes shown in the Visual Builder window.

Note: If the list boxes in the Visual Builder window are empty, see “Loading Part Files” on page 31 if you need help loading part files.
- The Visual Builder window with parts loaded from the part file is shown in Figure 34 on page 114.
2. Select the first part you want to open.
3. Do one of the following, depending on how the other parts appear in the list:
 - If the other parts are adjacent in the list to the part previously selected, hold down the Shift key and click on the last part in the group you want to select. All parts between the first and last parts selected are now highlighted.
 - If the parts are not adjacent in the list, hold down the Ctrl key while selecting each part.
4. Select **Part** on the menu bar.
5. Select **Open** in the pull-down menu. Visual Builder displays a separate window for each part that you selected. The window displayed is the Composition Editor for visual parts or the Part Interface Editor for nonvisual parts.

Copying Parts from One Part File to Another

To copy a part, do the following:

1. Select the part that you want to copy in the **Visual Parts** or **Nonvisual parts** list box. If you select more than one part or if you do not select a part, the **Copy** function is not available.
2. Select **Part→Copy**. The following window is displayed:

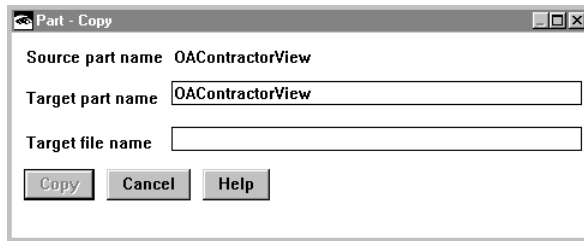


Figure 38. Copy Part Window

- The **Source part name** field shows the name of the part that you selected to copy.
3. In the **Target part name** field, enter the name you want the part to have when you copy it.
 4. In the **Target file name** field, enter the name of the part file to which you want to copy the part. If you leave this field blank, the part's current file name is used.
 5. Select the **Copy** push button. The part is copied under the new name and stored in the designated part file.

Moving Parts to a Different Part File

Here is what happens to the part file into which the part or parts are being moved:

- If this part file does not exist, Visual Builder creates and loads it for you.
- If this part file already exists and is loaded, the part or parts are moved into it.
- If this part file already exists but is not loaded, Visual Builder displays a message to warn you that the unloaded part file will be overwritten by the part or parts that you are moving into it.

To move one or more parts from one part file to another, do the following:

1. Select the part or parts that you want to move. If you do not select at least one part, the **Move** function is not available.

Developing Applications

2. Select **Part→Move**.
3. Use the following instructions for moving one part or multiple parts:

Moving one part

If you selected one part, the following window is displayed:

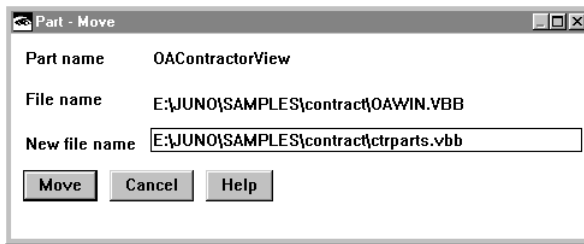


Figure 39. Move Part Window for Moving One Part

The **Part name** field of this window shows the name of the part that you selected to move. The **File name** field displays the complete path of the part file that contains the part you want to move.

Do the following:

- a. In the **New file name** field, enter the path and name of the part file to which you want to move the part.
- b. Select the **Move** push button.

The part is moved to the part file specified in the **New file name** field.

Moving multiple parts

If you selected more than one part, the following window is displayed:

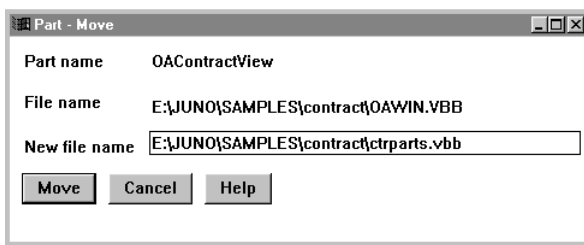


Figure 40. Move Part Window for Moving More Than One Part

The text in the window specifies the names of the parts you selected. Do the following:

Developing Applications

- a. In the entry field, enter the name of the part file to which you want to move the parts. If the part file is not in your current directory, specify the complete path for the part file.
- b. Select the **OK** push button. The parts are moved to the part file specified in the entry field.



An alternative method of moving a part is to change the name of the part file specified in the Class Editor. For more information, see “Moving a Part to a Different Part File” on page 51.

Deleting Parts from a Part File

To delete a part, do the following:

1. Select the part or parts that you want to delete in the **Visual parts** list box, **Nonvisual parts** list box, or both.

If you do not select at least one part, the **Delete** function is not available.

2. Select **Part→Delete**.

The following window is displayed:



Figure 41. Delete Parts Window

Deselect any parts that you do not want to delete. Once you delete a part from a part file, you cannot recover it unless you have another copy stored in another part file.

3. Select the **Delete** push button.

The selected parts are deleted.

Developing Applications

Renaming Parts in Part Files

The **Part→Rename** menu choice lets you change the name that a part is stored under in a part file.



Use care when renaming parts because the name changes only in the part file in which the part is stored. The name of the part does not change in any other part in which this part is embedded, that is, used as a subpart. Therefore, the next time you open the part in which you embedded the renamed part, Visual Builder will not be able to find the renamed part.

To rename a part in a part file, do the following:

1. Select the part that you want to rename in the **Visual parts** or **Nonvisual parts** list box. If you select more than one part or if you do not select a part, the **Rename** function is not available.
2. Select **Part→Rename**. The following window is displayed:

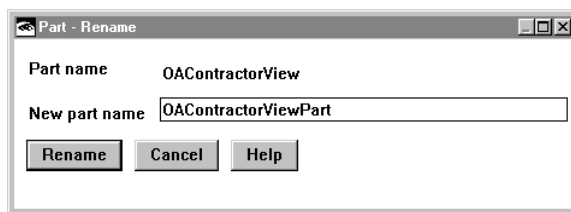


Figure 42. Rename Part Window

The **Part name** field shows the name of the part that you selected to rename.

3. In the **New part name** field, enter the new name that you want to give the part.
4. Select the **Rename** push button. The part is renamed under the new name.

Working with Parts on the Free-Form Surface

Placing Parts on the Free-Form Surface

In the Composition Editor, you place visual, nonvisual, and class interface parts on the free-form surface. This section explains how to place parts there that appear on the parts palette, as well as parts that do not appear on the parts palette.

Placing a part that appears on the parts palette

1. From the left column of the parts palette, select the appropriate category. Then, from the right column, select the part you want to add. When the mouse pointer


Developing Applications

is moved over a place where the part can be placed, it changes to a crosshair, indicating that it is loaded with the part.

2. Move the mouse pointer to where you want to add the part.
3. Click mouse button 1. If you hold down mouse button 1 instead of clicking it, an outline of the part is displayed under the pointer to help you position the part. After the part is in position, release mouse button 1.

See “The Parts Palette” on page 44 and “The Free-Form Surface” on page 48 for more information.

To unload the mouse pointer at any time, do either of the following:

- Select , the Selection tool, on the tool bar.
- Select **Tools**→**Selection tool** on the menu bar.



To add several copies of the same part, select **Sticky** on the parts palette. When **Sticky** is selected, the mouse pointer remains loaded with the part you last selected. When **Sticky** is not selected, the mouse pointer becomes unloaded after you add a part.

Placing a part that is not on the parts palette

You can place on the free-form surface any part whose .vbb file is loaded by doing the following:

1. Select **Add part** from the **Options** pull-down menu. The Add Part window appears, which resembles the window shown in Figure 43:

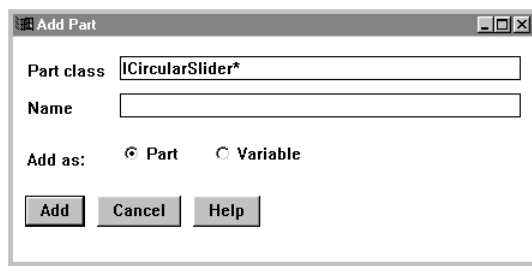


Figure 43. The Add Part Window

Developing Applications

2. Type the part's class name in the **Part class** field. This is the class name that was specified when the part was created. When you begin typing, you replace the highlighted prompt, `class_name*`, with the name of the part you want to add.

The asterisk at the end of the name is a reminder that you are actually entering a pointer to the part. When you enter a valid class name for the part without deleting the asterisk, Visual Builder automatically selects the **Part** radio button. You can change this by selecting the **Variable** radio button if you want to place a pointer to a variable on the free-form surface. Figure 44 shows how you could place a notebook part on the free-form surface using the Add Part window:

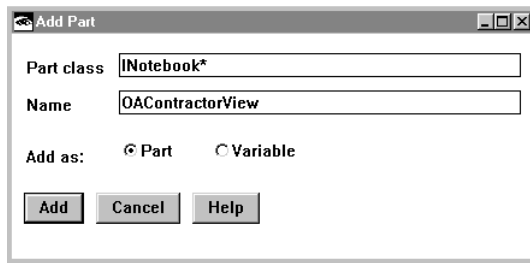


Figure 44. Placing a Notebook Part Using the Add Part Window

If you delete the asterisk, the **Part** radio button is not available. This means that you can only place a variable on the free-form surface; Visual Builder automatically selects the **Variable** radio button indicating that you will be adding a variable whose type is the class name you entered in the **Part class** field, as shown in Figure 45:

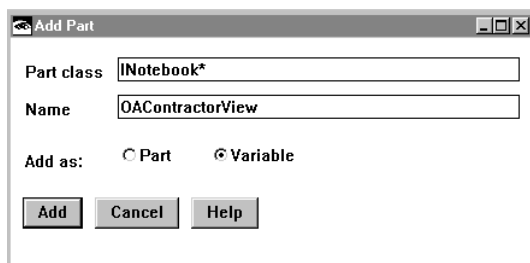


Figure 45. Placing a Notebook Variable Using the Add Part Window

Note: You cannot add abstract parts or template parts using the Add Part window. For example, `IButton*` is the abstract part that `IPushButton*` inherits from. You can add `IPushButton*`, but not `IButton*`.

Developing Applications

Similarly, you cannot add an object factory part using the Add Part window because it is a template. You must add it by selecting it on the parts palette.

3. Type a name for the part in the **Name** field. This name will appear in the information area at the bottom of the Composition Editor when you select the part after it is placed; it is also used for the part when you generate your part code.

The **Name** field is optional. If you leave it blank, the part's class name is used.

4. Select the **Add** push button to add the part. The Add Part window disappears and the mouse pointer turns into the same crosshairs used for placing a part on the free-form surface.

Note: If you did not enter the name of a part that is known to Visual Builder in the **Part class** field, the **Add** push button is not enabled.

5. Move the crosshairs to the place where you want to add the part and click mouse button 1.

Guidelines for Placing Parts on the Free-Form Surface

Following are guidelines for placing parts on the free-form surface:

- Avoid overlaying parts

You can overlay visual parts. Generally speaking, however, it is not good interface design for one part to overlay another part, such as one push button either completely or partially covering another push button. Be aware that completely overlaying a part can cause focus problems, meaning that runtime users might be able to see, but not select, the part.

Partially overlaying a part can cause problems, too, because the runtime users might not be able to see where the overlaying occurs. When they try to select the part that is partially overlaid, they might be lucky and select the right spot, or they might select the part that is overlaying the part they are trying to select. If you overlay parts, be sure to do it in a way in which the user can understand why the parts are overlaid and how to select them.

You cannot overlay, or cover up, nonvisual parts. You can overlay visual parts, but you should avoid doing this if possible unless you are placing parts on top of a part in the Composers category (see the next guideline).

- Place other parts on top of parts in the Composers category

Parts included in the Composers category have a special behavior; these parts can contain any other visual parts that are placed on top of them. The parts that the Composers part contains automatically become subparts of the Composers part. For example, if you place an entry field, a list box, and two push buttons in a

Developing Applications

frame window, the frame window contains the other parts and they in turn become the frame window’s subparts.

The following table lists each of the Visual Builder categories and specifies how you can use the parts in each category.

Figure 46. Categories and How You Can Use Their Parts

Category	Use Parts to Contain Other Parts?	Use Parts as Subparts?
Buttons	No	Yes
Data entry	No	Yes
Lists	No	Yes
Frame Extensions	No	No
Sliders	No	Yes
Composers	Yes	Yes
Models	No	No
Other	No	No

- Use supplementary composite parts as subparts

Suppose you create a visual composite part that consists of a canvas on which you have placed other visual parts, such as radio buttons and check boxes, with each radio button and check box connected to a variable, as shown in

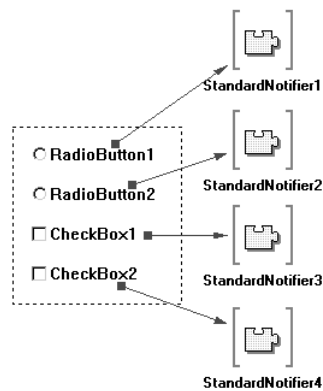


Figure 47. Parts Connected to Variables

Assume this part is not your main composite part but is instead a supplementary part that you want to use in your application’s user interface. When you place this part in your main composite part, such as in a frame window, as shown in

Developing Applications

Figure 48 on page 129. You place it and work with it as one part, not as a canvas and separate radio buttons and check boxes. The frame window contains the entire supplementary part, which becomes a subpart of the frame window.

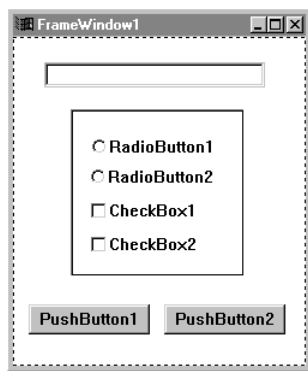


Figure 48. Composite Part Placed in Frame Window as Subpart

One of the first things you have probably noticed is that the connections for a supplementary composite part are not displayed when that part is added to another part. The connections and variables are still there; you just cannot see them because you cannot edit them directly from the Composition Editor window that contains the main composite part. Also, you cannot select the individual radio buttons, check boxes, or their connections in the supplementary part that you placed in your main part.

To change the connections or the default text on the radio buttons and check boxes, or to do anything else to alter this part, you must edit the part indirectly, as described in “Editing Parts Placed on the Free-Form Surface” on page 151.

Selecting and Deselecting Parts

Before you can perform an action on a part that you have placed on the free-form surface, such as sizing it, you must first select the part. The name of the part currently selected is displayed in the information area at the bottom of the Composition Editor. If more than one part is selected, then **Multiple selection** is displayed.

You cannot select parts and connections together. They are mutually exclusive. However, if you delete a part that is connected to other parts, Visual Builder deletes the connections in addition to the part.

When a part is selected, small boxes, called selection handles, are displayed on its corners. If more than one part is selected, the one you selected last has solid

Developing Applications

selection handles, indicating that it is the *anchor part*. The other selected parts have hollow selection handles as shown in the following figure:

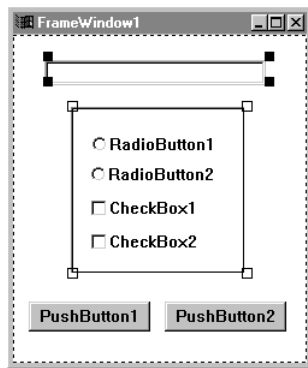


Figure 49. Multiple Parts Selected with the Entry Field as the Anchor Part

Some parts are not sizable and, therefore, do not have any selection handles. These parts have their background reverse colored. Parts with this behavior include variables, menus, and tear-off attributes, among others.

The following sections describe how to select and deselect a single part and multiple parts.

Selecting a single part

To select a part that you have placed on the free-form surface, click on the part with mouse button 1. If other parts are already selected, they are deselected automatically.

Selecting multiple parts

Selecting multiple parts lets you perform the same operation on several parts at once. To select multiple parts, do one of the following:

- Hold down the Ctrl key in OS/2 or the Shift key in Windows and click mouse button 1 on each additional part you want to select.
- Hold down mouse button 1 instead of clicking it. Then move the mouse pointer over each additional part you want to select. After you select the parts, release mouse button 1. (This method works only in OS/2.)

Note: Depending on the operation you want to perform, remember to consider which part you want to be the anchor part because that is the part you want to select last. For example, if you select two parts because you want to match the

Developing Applications

width of one part to the width of the other, the part you select last is the anchor part, the part whose width is used for the operation.

Deselecting parts

To deselect a part after you have selected it, do one of the following:

- Hold down the Ctrl key in OS/2 or the Shift key in Windows and click on the selected part with mouse button 1.
- Click mouse button 1 in a clear spot on the free-form surface.

When the selection handles disappear, you know that the part is no longer selected.

To deselect multiple parts, do the following:

1. Hold down the Ctrl key in OS/2 or the Shift key in Windows.
2. Click and release mouse button 1 one or more selected parts.
3. Repeat the previous step until all parts you want to deselect have been deselected.

Manipulating Parts

Once a part is added to the free-form surface, you can manipulate it in a number of different ways. The following sections explain each of those ways.

Displaying Pop-Up Menus

To display the pop-up menu of a part, click on the part with mouse button 2. The pop-up menu displays the operations you can perform on that part.

A part does not have to be selected for you to display its pop-up menu. The pop-up menu that is displayed is for the part the mouse pointer is over when mouse button 2 is clicked, even if another part is selected.

Copying Parts

To copy parts by dragging them, do the following:

1. Select all the parts you want to copy. If you only want to copy one part, you do not have to select it.
2. Move the mouse pointer over the part you want to copy or one of the selected parts.
3. Hold down the Ctrl key and mouse button 2 in OS/2 or the Ctrl key and mouse button 1 in Windows.
4. Drag a copy of the part or parts by moving the mouse pointer to a new position. An outline of the part or parts is displayed to help you with positioning. When

Developing Applications

you are copying multiple parts, the outlines of each part move together as a group.

5. Release the Ctrl key and mouse button when the part or parts are where you want them to be. A copy of the part or parts appears where you positioned the outline or outlines.

Copying parts using the clipboard

To copy parts by using the clipboard, do the following:

1. Select all the parts you want to copy.
2. From the **Edit** pull-down menu, select **Copy**. A copy of each selected part is placed on the clipboard.
3. Select **Paste** from the **Edit** pull-down menu when you are ready to use the parts. The mouse pointer turns to crosshairs to show that it is loaded with the copied parts.
4. Position the mouse pointer where you want the parts to be copied.
5. Click mouse button 1. Copies of the parts are pasted at the position of the mouse pointer.



Parts that you copy remain on the clipboard until you copy something else. Therefore, you can continue to paste copies of those parts by selecting **Paste**, positioning the mouse pointer, and clicking mouse button 1.

If you select **Paste** and then decide against pasting the parts, you can unload the mouse pointer by either selecting the Selection tool on the tool bar or by selecting **Tools**→**Selection tool** on the menu bar.

Deleting Parts

To delete one or more parts, do the following:

1. Select all of the parts you want to delete. If you are deleting just one part, you do not have to select it.
2. Position the mouse pointer over the part you want to delete or one of the selected parts.
3. Click mouse button 2.
4. From the part pop-up menu, select **Delete**. The part or parts are deleted.

You can also delete a part by pressing the Delete key after selecting the part.



Any connections between the part that you are deleting and other parts are also deleted. Visual Builder displays a message to alert you of this. However, the **Edit→Undo** function also restores any connections that were removed when you deleted the part.

Editing Text Strings

Some visual parts, such as push buttons and menus, have text strings. To directly edit a part's text string, do the following:

1. Hold down the Alt key.
2. Click mouse button 1 on the text string.
3. Edit the text string.
4. When you have finished, do either of the following:
 - Click mouse button 1 anywhere outside of the text string.
 - Press Shift+Enter.



You can also use this *direct editing* technique to edit the names of nonvisual parts. The name of a nonvisual part is displayed directly below its icon.

Renaming Parts on the Free-Form Surface

When you use parts in the Composition Editor, Visual Builder gives those parts a name based on the names given to the parts on the parts palette or the names you specify when you place parts on the free-form surface. For example, the first push button part that you use is named `PushButton1`. When you select this part, the information area at the bottom of the Composition Editor shows the message “`PushButton1` selected.” The second push button you use is named `PushButton2`, the third is named `PushButton3`, and so forth. These default names are assigned to help Visual Builder distinguish one part from another, as well as the connections between parts, when you generate the code to build your application.

If you want to give parts names that are more descriptive or meaningful to your application, you can do so as follows:

1. Move the mouse pointer over the part whose name you want to change.
2. Click mouse button 2 to display the pop-up menu for the part.
3. Select **Change name**. A Name Change Request window is displayed. Figure 50 on page 134 shows a Name Change Request window for a push button part.

Developing Applications

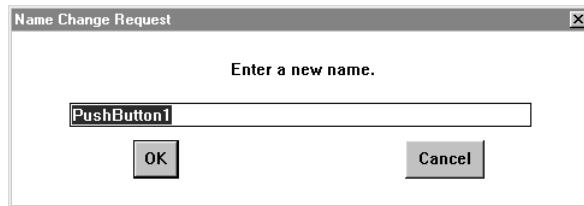


Figure 50. Name Change Request Window for a Push Button Part

4. Type a new name in the entry field.
5. Select **OK**. Visual Builder changes the name of the part to the name that you typed in the entry field.

You can also change a part's name by opening the part's settings notebook and changing the name in the **Subpart name** field.

Arranging Parts

You can arrange parts on the free-form surface in a number of different ways. The following sections explain each of those ways.

Moving Parts

To move a part, move the mouse pointer over the part, hold down mouse button 2 in OS/2 or mouse button 1 in Windows, and move the mouse pointer to drag the part to the new position.



You can move several parts at once by first selecting all the parts you want to move and then dragging one of the parts as described. All of the selected parts will move together, maintaining their position relative to each other.


Positioning Parts on the Grid

The free-form surface has a grid that you can use to position parts. In addition, parts that can contain other parts (for example, any Composers part, such as a frame window) have a grid associated with them. You can use this grid to align and evenly space subparts that Composers parts contain.

To position the upper-left corner of parts to the nearest grid coordinate, do the following:

1. Select all the parts you want to position to the grid.

Note: If the parts you select are subparts, they are positioned to the grid set up inside the Composers part, not the grid for the free-form surface.

2. Select , the **Snap To Grid** tool.



You can automatically position a part to the nearest grid coordinate when it is added to the free-form surface or a Composers part by selecting **Snap on drop** from the **Options** pull-down menu.


Specifying Grid Spacing

To specify the grid spacing, do the following:

1. From the pop-up menu of a Composers part or the free-form surface, select **Set grid spacing**.
2. Specify the horizontal and vertical distance between the lines of the grid in pixels.


Showing and Hiding the Grid

To toggle between showing and hiding the grid for the free-form surface, do one of the following:

- If no parts are selected, you can select , the **Toggle Grid** tool to toggle the grid for the free-form surface.
- If a Composers part is selected, selecting the **Toggle Grid** tool toggles the grid for the Composers part instead of the free-form surface.

Toggle between showing and hiding the grid for a Composers part

To toggle between showing and hiding the grid for a Composers part, do one of the following:

- Select the Composers part and the select , the **Toggle Grid** tool.
- From the Composers part's pop-up menu, select **Toggle Grid**.

Developing Applications

Sizing Parts

To change the size of a part, select it and use mouse button 1 to drag one of the selection handles to the new position. An outline of the part is displayed under the mouse pointer to show you the new size of the part.



You can size several parts at once by first selecting all the parts you want to size.

To size a part in only one direction, press and hold the Shift key while using mouse button 1 to size the part. Holding down the Shift key prevents one dimension of the part from changing while you resize the other dimension. For example, to change the width of a part but prevent its height from changing, hold down the Shift key while changing the width.

You can also size a part to the grid coordinates by selecting **Snap on size** from the **Options** pull-down menu.

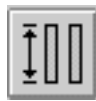
Matching Part Sizes

To size parts to the same width or height of another part, do the following:

1. Select all the parts you want to size, making sure the last part you select is the part whose size you want the others to match.
2. Select one of the following sizing tools from the tool bar:



Match Width



Match Height

The size of all the parts you selected, with the exception of the last part, changes to match the size of the last part selected.

Aligning Parts

To align parts to the same position as another part, do the following:

1. Select all the parts you want to align, and then select the part you want the others to be aligned with.
2. Select one of the following alignment tools from the tool bar:



Align Left



Align Top



Align Center



Align Middle



Align Right



Align Bottom

Spacing Subparts within Composers Parts

To evenly space subparts within their Composers part, do the following:

1. Select all the parts you want to evenly space.
2. Select one the following spacing tools from the tool bar:



Distribute Horizontally



Distribute Vertically

Spacing Parts within a Bounding Box

To evenly space parts within the unseen bounding box that contains the selected parts, do the following:

1. Select all the parts you want to evenly space. You must select a minimum of three parts.
2. From the pop-up menu of one of the selected parts, select **Layout→Distribute**, and then one of the following:

Horizontally in bounding box

Evenly distributes the selected parts within the region bounded by the leftmost edge and rightmost edge of the selected parts.

Vertically in bounding box

Evenly distributes the selected parts within the region bounded by the topmost edge and bottommost edge of the selected parts.

For more information on tool bar tools, see “The Tool Bar” on page 41.

Developing Applications

Changing Settings for a Part

The settings notebook of a part provides a way to display and set attributes and options for the part.

The settings notebooks for the Visual Builder parts do not necessarily reflect the current default values for nonvisual parts and connection parameters. To ensure that initial values are set the way you expect, always explicitly set them. For example, suppose the default value of a Boolean is true and you want to set it to false. When you open the settings notebook for the part, the check box for the Boolean attribute is not selected. The deselected check box indicates only that the value has never been set, not that the initial value is false. To set it to false, select and then deselect the check box.

For some part instances, certain current settings appear on the **Styles** page as “not set,” even if you select a radio button to set the styles. A “not set” label indicates only that the style setting is not reflected in the Composition Editor display; the correct setting appears in the generated code for the part. If you do not set such styles explicitly, default values appear in the generated code.

For settings in numeric entry fields, Visual Builder reflects only those values entered in decimal notation. You can enter numbers with bases other than 10 by using the number sign (#, as in #0xBF), but the expression is not evaluated until compile time.

Visual Builder does not evaluate settings that you enter as string expressions (text strings preceded by a number sign). User interface controls whose size depends on the length of the setting value, such as push buttons whose size is determined by the length of their text strings, are not displayed in the Composition Editor as they would appear in the compiled application. However, Visual Builder passes the expression correctly into the generated code so it compiles successfully. For example, suppose you enter #pbText as the value of a text attribute of an IPushButton* part. Because #pbText is not a literal value, the length of the text string is unknown and the push button appears small in the Composition Editor. Assuming you have defined the value of #pbText elsewhere, the push button appears with the correct size and text string in the compiled application.

Opening the settings notebook for one part

To open the settings notebook for a part, move the mouse pointer over the part and do one of the following:

- Double-click mouse button 1.
- Click mouse button 2 and select **Open settings** from the part’s pop-up menu.

Developing Applications

Opening settings notebooks for multiple parts

You can open the settings notebooks for multiple parts by doing the following:

1. Select the parts whose settings you want to change.
2. Move the mouse pointer over one of the selected parts.
3. Click mouse button 2.
4. Select **Open settings** from the pop-up menu.

Visual Builder opens a settings notebook for each of the selected parts.

Navigating through a settings notebook

You can navigate through the notebook pages in the following ways:

- To turn the pages of a notebook, use the small left- and right-arrow push buttons at the lower-right corner of each page.
- To move to a different settings category, select one of the tabs to the right of the pages.

Note: When a category has more than one page, the page number and total number of pages within the category are displayed at the bottom of the page.

- If all of the category tabs do not appear on the pages of the notebook, small left- and right-arrow push buttons are displayed to the left of the category tabs, and small up- and down-arrow push buttons are displayed above and below the category tabs. Use these buttons to move through the available category tabs.

About the settings pages

The following list contains a description of each of the pages a settings notebook might contain:

General

A page for setting the name of the part, any static text that might appear on the part, and other part-specific settings. For example, the **General** page for an `IMenuItem*` part contains a group box for setting the keyboard accelerator for the menu item. Refer to the *Visual Builder Parts Reference* for descriptions of specific settings for parts.

Control

A page that allows you to specify information for the part in its role as a control part, such as fly-over text, a window ID, and whether the part is available for the user to select.

Developing Applications

Styles

A page that provides style settings from the IBM Open Class Library. The style settings generally correspond to those of the class on which the part is based. Refer to the *IBM Open Class Library Reference* for descriptions of the style settings.

Selecting the **defaultStyle** check box means you want Visual Builder to use the default style provided for the class by the IBM Open Class Library. You can select this check box and then modify the default settings by selecting the **On** and **Off** radio buttons on the page beside each style setting. Doing this means you want Visual Builder to use the default style except for any exceptions that you have made by turning style settings on or off.

The **Default** column shows which style settings are the defaults as specified for the class in the IBM Open Class Library. Visual Builder has not turned these style bits on or off. They are set to whatever the default settings are supposed to be.



If you deselect the **defaultStyle** check box and also turn off one or more of the required settings, you will get errors when you generate and compile your code. However, selecting this check box ensures that no required settings will be omitted. Therefore, we recommend that you keep the **defaultStyle** check box selected so that you always have the required settings.

Selecting one style setting does not cause another to be automatically deselected. For example, selecting minor tabs for a notebook page does not cause major tabs to be deselected. In this case, the major tab style setting overrides the minor tab setting. Be sure to deselect the styles that you do not want to use.

Handlers

A page that allows you to list handlers that you want to attach to this part. You can use handlers instead of event connections, such as event-to-action connections. List the handlers in the order in which they should be called.

Adding a handler

To add a handler, do the following:

1. Enter the name of the handler class in the **Handler name** field, along with the list of parameters that you want to send to the handler's constructor. If you use any part names as parameters, be sure to use the default name that Visual Builder has assigned unless you have changed the name. Also, put a lowercase "i" before those parameters because Visual Builder prefixes the part name with an "i" when it generates the code files. For example, if you are using the first entry field that you placed in a frame window as a parameter

Developing Applications

and have not changed the default name that Visual Builder assigns, the parameter name would be `iEntryField1`.

2. If the **Handler list** list box contains other handlers, select the handler that you want your new handler to either precede or follow.
3. Select either the **Add after** or **Add before** push button. If you did not import the handler class from a `.vbe` file, Visual Builder displays a message saying that the name you entered is not a valid part and asks if you want to continue.
4. Select the **Yes** push button. The message disappears and the handler is added either after or before the handler that is selected in the **Handler list** list box, depending on which push button you select.
5. Select the **OK** push button to save the new handler in the handler list and close the settings notebook.



We recommend that you put your handler class declaration and code in separate `.hvp` and `.cpv` files rather than modifying the files that Visual Builder generates. This way, if you need to regenerate the files, you do not have to recreate your handler code.

Be sure to include the names of the files that contain the handler code in the **User .hvp file** and **User .cpv file** fields in the Class Editor.

For information about implementing handlers, refer to the *IBM Open Class Library User's Guide* and the *IBM Open Class Library Reference*.

Moving a handler

To move a handler to a different position in the list, do the following:

1. Select the handler that you want to move.
2. Select the **Move** push button.
3. In the window that is displayed, select the handler that is to precede or follow the handler being moved.
4. Select the **Move after** push button to move the handler after the selected handler, or select the **Move before** push button to move the handler before the selected handler. The window disappears and the handler is moved.

Removing a handler

To remove a handler from the list, do the following:

1. Select the handler you want to remove from the list.
2. Select the **Remove** push button. The selected handler is removed from the list.

Developing Applications

Color

A page that allows you to change the color of the part.

Changing the color

To change the color, do the following:

1. In the **Color area** group box, select the area, such as foreground or background, whose color you want to change.
2. Do one of the following:
 - If you want to specify red-green-blue values, select the **RGB** check box and specify values in the fields in the **RGB values** group box.
 - If you want to select a color by its name, deselect the **RGB** check box and select a color from the **Colors** drop-down list box.
3. Select either the **Apply** push button to see how this color looks for your part without saving the change or the **OK** push button to close the settings notebook and save the color change.

Size/Position

A page that allows you to specify the size and position of a part.

Specifying the size and position of a part

To specify the size and position of a part, do the following:

1. In the **x** and **y** fields, specify the initial X and Y coordinates for the part. These coordinates determine the position of the part's upper-left corner.
2. In the **width** and **height** fields, specify the number of pixels for the width and height of the part.
3. Optionally, you can also specify the smallest size the part can have by using the **Minimum size** group box to do either of the following:
 - If you want the minimum size to be calculated for you, select the **Calculate at execution time** radio button.
 - If you want to specify the minimum size for the part, select the **Set value here** radio button and then enter the width and height in pixels in the corresponding fields.

Font

A page that allows you to specify the font that is to be used for the part.

Changing the font for a part

To change the font for a part, do one of the following:

- If you know the name and size of the font you want to use, you can enter them in their respective fields.

Developing Applications

- If you do not know the name and size of the font you want to use or if you want to change the style or emphasis, select the **Edit** push button. Visual Builder displays a standard font dialog from which you can select the name, size, style, and emphasis you want to use for the part's font.

Using code strings to supply initial field values

Many settings pages provide fields in which you can specify initial values for part settings. For example, the **General** page of the IEntryField* settings notebook contains a **Text** field and a **Limit** field. In the **Text** field, you can enter a text string that you want Visual Builder to initially display in the entry field. The **Limit** field contains a default value of 32, which represents the maximum number of characters a user can type in the entry field, and you can change this number.

To facilitate national language support (NLS) translation and code changes in providing settings values, such as the ones just described, you can enter a code string to provide those values. You must precede the code string with a number sign (#). If the first character of your code string is a #, be sure to enter two #s—the first one to signify that a code string follows and the second one to begin your code string.

For example, suppose you want the initial text in an entry field to be Enter a name here. Further, suppose that you want the limit for this entry field to be 18 characters. In a user header file (.hvp or .h), you could insert the following #define statements:

```
#define NAME_PROMPT "Enter a name here"
#define NAME_LENGTH 18
```



Be sure to enter the name of the file that contains these #define statements in the **Required include files** field in the Class Editor. Otherwise, this file is not included when you generate the code for this part.

Then, on the **General** page of the IEntryField* settings notebook you could enter the following in the **Text** and **Limit** fields, respectively:

```
#NAME_PROMPT
#NAME_LENGTH
```

By doing this, the values that you defined for NAME_PROMPT and NAME_LENGTH are used when you generate the source code for the part being edited.

For an example that uses a code string to specify an icon to represent items in a container, see the subsection titled “Specifying the container type and layout” in “Adding Container Parts” on page 231.

Developing Applications

Activating settings changes

After you make changes to the settings, you can activate them in the following ways:

- Select the **OK** push button to immediately activate and save the settings changes you made and to close the settings notebook.
- Select the **Apply** push button to apply the settings changes you have made and keep the settings notebook open.

This allows you to see whether you need to modify any of the changes you have made. The changes remain applied until you change them again.

Select the **Cancel** push button to close the settings notebook. If you made changes and selected the **Apply** push button, the changes are saved.

Select the **Help** push button for more help with using the settings notebook.

For information about the settings of a particular part, refer to the settings section of the part in the *Visual Builder Parts Reference*.

Using the Generic Settings Notebook

When you create a part, Visual Builder provides a settings notebook for that part. The settings notebook for your part has one page, which contains the following types of settings:

- An entry field for each attribute that has a set member function
The #define statements are supported for the generic settings notebook, just as they are for a regular settings notebook. See “Changing Settings for a Part” on page 138 for information about using #define statements in a settings notebook.
- A check box for each Boolean attribute

If your part has no attributes, the page displays a message saying that there are no values to set.

To view the settings notebook for your part, do one of the following:

- To set values for an instance of your part that you are using in a composite part, do the following:
 1. Place the part in the Composition Editor.
 2. Move the mouse pointer over the part and click mouse button 2.
 3. Select **Open settings** from the pop-up menu.

Visual Builder displays the settings notebook for the part.

Developing Applications

- To set attribute values for your part that will be available each time your part is used, do the following:
 1. Open the part.
 2. If the part is a nonvisual or class interface part, switch to the Composition Editor. You will already be in the Composition Editor if your part is a visual part.
 3. Move the mouse pointer over the free-form surface and click mouse button 2.
 4. Select **Open settings** from the pop-up menu.



A quicker way to open the generic settings notebook is to double-click on the part.

Visual Builder displays the settings notebook for the part.

Figure 51 shows the generic settings notebook for the OAContract part:

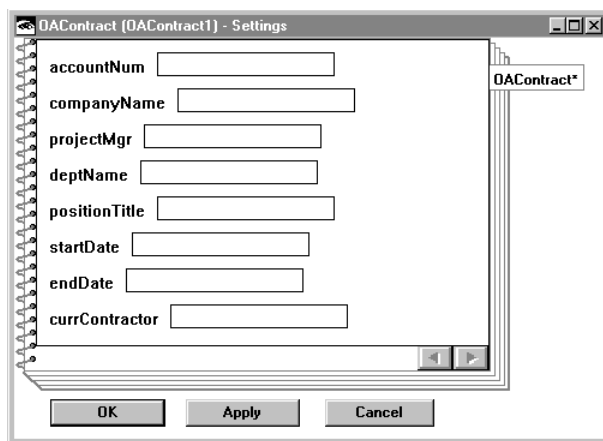


Figure 51. Generic Settings Notebook for OAContract Part

The notebook page includes settings for attributes that your part inherits from other parts in addition to attributes for the part you created. For example, the **enabledForNotification** check box is present because the OAContract part inherits this Boolean attribute from IStandardNotifier.

Developing Applications


Listing Parts within a Composite Part

The Parts List window provides a way to display an ordered list of the parts dropped on a given part. At first, parts are listed in the order in which they were dropped. If you then change the tabbing order of parts that have tabbing set, Visual Builder rearranges the list to reflect the updated tabbing order. For more information on tabbing, see “Setting the Tabbing Order” on page 148.

Note: In Windows, make sure your system palette is set to 256 colors or fewer before trying to list parts.

To list parts within a composite part, do the following:

1. Open the composite part.
2. Click on the free-form surface with mouse button 2. The part’s contextual menu appears.
3. Select **View parts list**. The Parts List window opens. At first, the Parts List

window displays only the immediate subparts of the selected part. An  expansion icon appears next to each part that contains subparts of its own. To see those parts, select the expansion icon.

The parts list for the OAContractView part is shown in Figure 52.

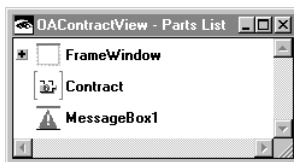


Figure 52. Parts List for OAContractView

Changing Depth Order within a Composite Part

Depth order is the order in which parts are stacked on the application desktop. Parts lower in the depth order overlay at least a portion of parts higher up. An example of this is a push button on a canvas. The canvas appears higher (or first) in the depth order; the push button, which lies on top of the canvas, appears lower (or later).

Visual Builder assigns the depth order as parts are dropped. Depth order is not linear, but hierarchical, depending on the arrangement of Composers parts. In the OAContractView part, the two Composers parts are fully nested. The parts list for OAContractView appears in Figure 52.

Developing Applications

In the OAMain part, a IMultiCellCanvas* part holds two parts: an IBitmapControl* and another IMultiCellCanvas*. In turn, the embedded IMultiCellCanvas* part holds several push button parts and some IStaticText* parts. The parts list for OAMain appears in Figure 53 on page 147.

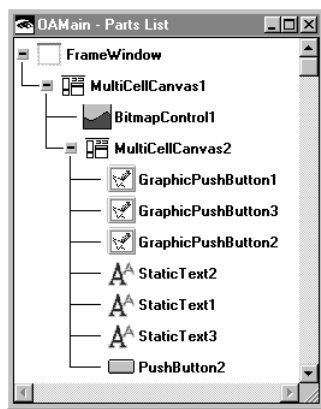


Figure 53. Parts List Window for OAMain

You can change the depth order within a single parent part by dragging items in the parts list. To change the depth order, do the following:

1. Open a parts list in the composite part by selecting **View parts list** from the part's contextual menu.
2. To move more than one part, do one of the following, depending on how the other parts appear in the list:
 - If the parts are adjacent in the list, select the first part in the group to be moved. Then hold down the Shift key and click on the last part in the group to be moved. All parts between the first and last parts selected are now highlighted.
 - If the parts are not adjacent in the list, select the first part. Then hold down the Ctrl key while selecting the other parts.

If you want to move only one part, you do not need to select it first.

3. Using mouse button 2, drag the selected parts to their new location in the depth order.

If tabbing has been set for any of the parts moved, this also changes the tabbing order. To find out more about tabbing order, see “Setting the Tabbing Order” on page 148.

Developing Applications

Performing Other Operations on Parts in the Parts List Window

You can perform some of the same operations on parts in the parts list that you can perform on the parts on the free-form surface. Visual Builder provides pop-up menus that contain the enabled operations for each part in the list.

To perform an operation on a part in the parts list, do the following:

1. Move the mouse pointer over the part.
2. Click mouse button 2 to open the part's contextual menu.
3. Select the operation you want to perform.

Setting the Tabbing Order

The tabbing order is the order in which the input focus moves from part to part as the user presses the Tab key. The tabbing order can also indicate the order in which the input focus moves among parts within a tab group as the user presses the arrow keys. Tabbing order is related to depth order, as discussed in “Changing Depth Order within a Composite Part” on page 146.

The tabbing order can only be set or displayed for parts that are placed within a Composers part. For example, if you place a row of push buttons in a frame window, you can set the tabbing order for the push buttons. Consider the part shown in Figure 54.

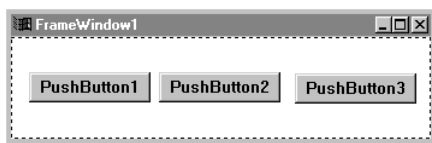


Figure 54. Frame Window with Push Buttons

The initial tabbing order is determined by the order in which you place the parts on the Composers part. Also, the first part in the tabbing order receives the initial input focus. For example, if the first part in the tabbing order is a push button, that push button receives the initial input focus when the application starts.

To display the tabbing order, open a parts list for the Composers part that contains the push buttons. Within the parts list, you can change the positions of parts in the tabbing order.

Changing the Tabbing Order

Because the order in which parts are placed on a Composers part determines the tabbing order, you will probably need to change the order as you add or rearrange parts. For example, suppose you decide to rearrange the three push buttons from the example in the preceding section so that `PushButton3` is between `PushButton1` and `PushButton2`, as shown in Figure 55.

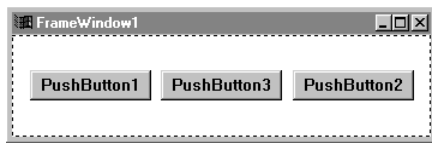


Figure 55. Rearranged Push Buttons

The tabbing order of these push buttons is `PushButton1`, `PushButton2`, `PushButton3`, even though `PushButton3` is now between `PushButton1` and `PushButton2`.

To change the position of a part within the tabbing order, do the following:

1. Open a parts list for the `ICanvas*` part that contains the push buttons.
2. Move the mouse pointer to the part in the list whose position you want to change.
3. Press and hold mouse button 2.
4. Drag the part icon to its new position.
5. Release mouse button 2.

The changed tabbing order is shown in Figure 56.

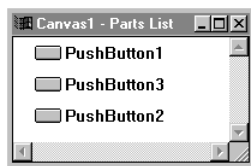


Figure 56. Resequenced Tabbing Order



You cannot move a subpart to a new Composers part by changing the tabbing order. You must do this by moving the parts themselves in the Composition Editor.

Developing Applications

Setting Tab Stops and Groups

If you want the user to be able to move the input focus to a part using the Tab and backtab keys, do the following:

1. Select the part.
2. Open the part's contextual menu.
3. Select **Set tabbing→Tab stop**.

If you want the user to be able to move the input focus to a part with the keyboard arrow keys, do the following:

1. Select the part.
2. Open the part's contextual menu.
3. Select **Set tabbing→Group**.

All parts in the tabbing order below the part that has **Group** selected are included in the group.

To start another group, select **Set tabbing→Group** for the part that you want to be the first part in that group. If a part has both **Group** and **Tab stop** selected, a user can tab to the first part in the group and then use the arrow keys to move to the other parts in the group.

Special considerations for radio buttons and entry fields

When you put radio buttons in groups, they become mutually exclusive within their group. For example, suppose you have four consecutive radio buttons in your list and you select **Group** for RadioButton1 and RadioButton3. In this case, RadioButton1 and RadioButton2 become mutually exclusive in their group, with RadioButton3 and RadioButton4 mutually exclusive in their group, as well. Tab stops are also set so a user can tab between the two groups.

Consider setting a tab stop on each entry field that a user can type in to allow the user to move the input focus from one entry field to another. Read-only entry fields do not need a tab stop, and arrow keys only move the cursor within an entry field; only the Tab key, backtab key, and mouse can change the input focus from one entry field to another.

Style guidelines for setting groups and tab stops

The following are some typical style guidelines for setting groups and tab stops:

- The position of the parts in the tabbing order should be the same as the order in which they are displayed in the window, from left to right and then top to bottom.

Developing Applications

- Parts that are not in groups, such as entry fields and list boxes, should have **Group** and **Tab stop** selected.
- Each group of related parts, such as check boxes and radio buttons, should be put within an outline box or a group box. If there is only one group of related parts, such as push buttons, you do not need to put them within an outline box or group box. Select only **Tab stop** for these parts.
- Parts that should not receive input focus, such as static text parts, should not have either **Group** or **Tab stop** selected.

Editing Parts Placed on the Free-Form Surface

Suppose you create a composite part, add it to another composite part that you are creating, and then realize that you need to change the first composite part. With Visual Builder you do not have to start over. It provides a way for you to edit the part that needs to be changed right from the free-form surface.

The only exception is the base parts that Visual Builder provides. Visual Builder does not allow you to modify these parts. This includes all of the parts that the part file contains. If you place one of these base parts on either a Composers part or the free-form surface, you can modify the subpart by doing any of the following:

- If you want to add an action to the subpart, consider connecting to a member function or custom logic that belongs to the composite part, instead. Write a member function or provide custom logic if you need to perform an action of limited use—that is, one that you do not anticipate using very often and that you do not want derived parts to inherit. For information about using member functions and custom logic in connections, see the following:
 - “Adding an Event-to-Member Function Connection” on page 180
 - “Connecting Features to Custom Logic” on page 184
- If you want to add a new feature that you plan to use often, create a new part that is derived from the base part. For example, to add a new feature to an IEntryField* part, create a new visual part whose base part is the IEntryField* part and replace the IEntryField* part that you were using with your new part. You can then add as many new features to your new part as you need.
- If you derive a new part from a base part, you also have the option of adding handlers to the new part. One reason for adding handlers is that you might want to monitor Presentation Manager or Windows messages to see when certain events occur and then notify observers. Another reason is that you might want to add special behavior to your part. You can add handlers to parts that Visual Builder provides on the **Handlers** page of the part’s settings notebook. For more information about handlers, refer to the *IBM Open Class Library User’s Guide*.

Developing Applications

If you need to edit a part that was added to the part you are currently editing, do the following:

1. If you have not already done so, load the part file that contains the part you want to edit.

Note: See “Loading Part Files” on page 31 if you need information about loading part files.

2. Move the mouse pointer over the part you want to edit.
3. Click mouse button 2. The part’s pop-up menu appears.
4. Select **Edit part**.

Visual Builder displays the appropriate editor for the part, as follows:

- If you are editing a visual part, Visual Builder displays the Composition Editor
- If you are editing a nonvisual or class interface part, Visual Builder displays the Part Interface Editor.

5. Edit the part.



If you want to promote any of the features of the parts used to create the composite part you are editing, doing so now keeps you from having to edit this part again later. See “Promoting a Part’s Features” on page 153 if you need more information about doing this.

6. Select **File→Save** to save the part.
7. Close the editor by doing one of the following:
 - Double-click on the system menu icon.
 - Select **File→Exit**.

The editor you are using disappears and you are returned to the Composition Editor you were using previously. However, Visual Builder has not applied the changes you made to the part you just edited, so those changes are not visible yet.

8. Select **File→Save** to save the original part.
9. Close the Composition Editor for the original part that you were editing, as described previously.
10. Reopen the original part you were editing by double-clicking on the part’s name in the Visual Builder window.

You should now be able to see the changes you made to the part that you edited.

Promoting a Part's Features

“Guidelines for Placing Parts on the Free-Form Surface” on page 127 discusses the relationship between parts in the Composers category and parts that are placed on top of them, called subparts. When you create a visual part that consists of a part from the Composers category that contains subparts, you can then place that visual part on top of another part from the Composers category. However, if you do this, the features of the subparts in the visual part that you created are not automatically available. You must promote these features to use them in connections.

For example, suppose you create a visual part called Buttons whose base class is ICanvas*. This part consists of a canvas part that contains three push button parts. Here is what the Buttons part looks like:



Figure 57. Buttons Part

Suppose you create another visual part whose base class is IFrameWindow* and then add the Buttons part to the frame window part. Here is what the frame window part with the Buttons part looks like:

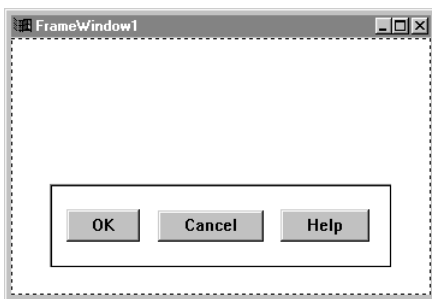


Figure 58. Frame Window with Buttons Part

Now, suppose that you want to connect the *buttonClickEvent* feature of the **Cancel** push button to the close feature of the frame window so that the window closes whenever the **Cancel** push button is selected. However, features of the push button parts are not automatically available for connections because the push buttons are subparts of the canvas.

When you use a part such as Buttons as a subpart, only the features of the Buttons part's base classes (ICanvas, IWindow, and so forth) are available in the connections menu for the Buttons part. To use the features of the push button parts, you must

Developing Applications

promote them to the Buttons part. You can do this either before or after you add the Buttons part to the frame window.

Promoting a part's features from the Composition Editor



To promote features of several subparts, we recommend using the **Promote** page of the Part Interface Editor. For information about promoting a part's features from the Part Interface Editor, see "The Promote Page" on page 73.

1. If you have not already done so, load the part file that contains the composite part whose features you want to promote.

Note: See "Loading Part Files" on page 31 if you need information about loading part files.

2. If you have not already done so, open the composite part or edit the subpart, whichever is necessary.

Note: See the following if you need more information:

- "Editing Parts Placed on the Free-Form Surface" on page 151 for information about editing subparts after you have placed them on the free-form surface
- "Opening Parts" on page 119 for information about opening parts

Visual Builder displays the Composition Editor with the part that you are opening or the subpart that you are editing on the free-form surface.

3. Move the mouse pointer over a part whose features you want to promote.
4. Click mouse button 2 to display the part's pop-up menu.
5. Select **Promote Part Feature**. Visual Builder displays a window with three columns of features: one each for actions, attributes, and events. Figure 59 on page 155 shows the window that is displayed for a push button part.

Developing Applications

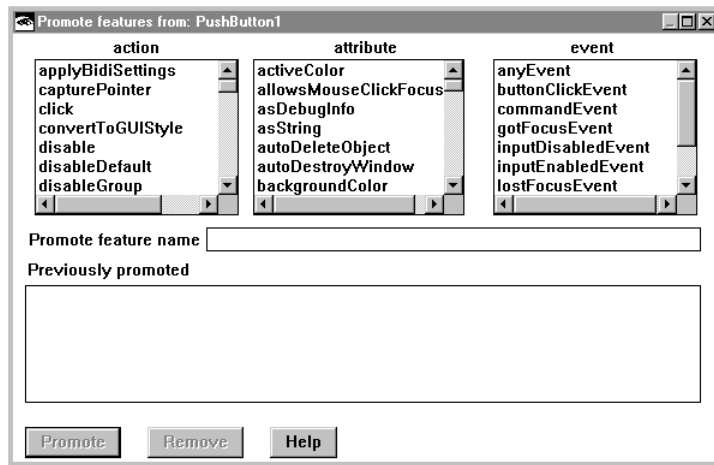


Figure 59. Promoting Features for a Push Button Part

6. Select a feature. The name of the feature you selected is displayed in the **Promote feature name** entry field prefixed with the name of the part. This is done so that when you make the connection, you can tell which push button part the feature belongs to. For example, if you selected the *buttonClickEvent* feature for the PushButton2 part, the feature name displayed in the entry field would be *pushButton2ButtonClickEvent*, as shown in Figure 60. This is how the feature name will appear in the connections menu unless you change it before performing the next step.

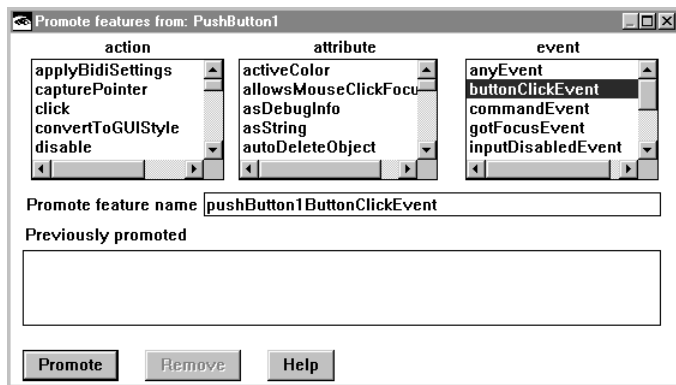


Figure 60. Part Name Prefixed to Feature Name for Promotion

7. Select the **Promote** push button. The feature name is added to the **Previously promoted** list box, as shown in Figure 61 on page 156.

Developing Applications

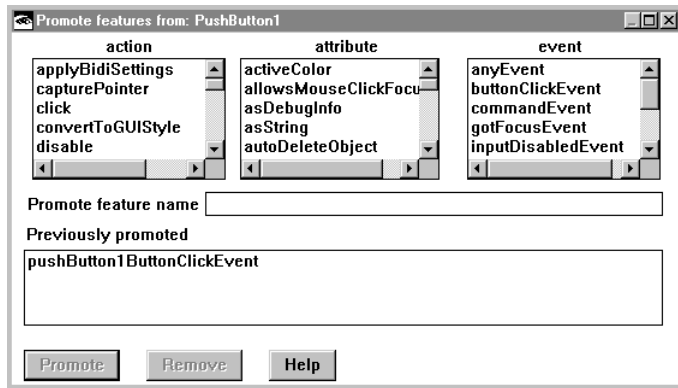


Figure 61. Promoted Feature Added to Previously Promoted List

8. Repeat the previous two steps until you have promoted all of the features you need to make the necessary connections.
9. Close the window for promoting features.
10. Select **File**→**Save** to save the features you just promoted.
11. Close the Composition Editor.

You can now use the feature or features that you promoted to make connections.

Tearing Off an Attribute

Select **Tear-off attribute** from a part's pop-up menu to work with an attribute as if it were a stand-alone part. The torn-off attribute is not actually a separate part but a variable that either represents the attribute itself or points to it.

When you select **Tear-off attribute**, Visual Builder displays the list of attributes for the part that you are tearing from. After you select an attribute from the list, you can drop the torn-off attribute on the free-form surface. Visual Builder creates an attribute-to-attribute connection between the original part and the torn-off attribute. You can then make other connections to or from the torn-off attribute. See Chapter 9, "Learning to Use Connections" on page 163 if you need information about attribute-to-attribute connections.

You might want to tear off an attribute to do the following:

- Allow direct access from one that is nested inside of another
- Enable direct access to an attribute's events and actions

For example, in an address book application you might tear off attributes as follows:

Developing Applications

- You might have a Person part that contains both *homeAddress* and *workAddress* attributes, both of which, in turn, could contain *street*, *city*, and *state* attributes.
- By tearing off either the *homeAddress* or *workAddress* attribute, you can create a new part that contains *street*, *city*, and *state* attributes.
- Tearing off a *homeAddress* or *workAddress* attribute makes the nested *street*, *city*, and *state* attributes directly accessible. Now that the nested attributes are directly accessible, you can make connections to and from them, as well as to their associated events and actions.

Undoing and Redoing Changes in the Composition Editor

If you change something in the Composition Editor and then decide that you should have left things as they were, select **Undo** from the **Edit** pull-down menu to restore the part to its previous state. You can undo as many operations as you want, up to when you opened the Composition Editor.

If you undo an operation and then decide that you did the right thing in the first place, select **Redo** from the **Edit** pull-down menu. **Redo** restores the part to the state it was in before the last **Undo**, including any connections that were deleted.

If you are not sure which operations you want to undo or redo, select **Undo/Redo list** from the **Edit** pull-down menu to display two lists of operations, one for undoing and one for redoing. From these lists, you can select an operation and then select the **Undo** or **Redo** push button. The operation that you select and all of the operations listed below it are undone or redone.

Note: **Undo**, **Redo**, and **Undo/Redo list** only affect operations you perform on the free-form surface and parts palette in the Composition Editor. They have no affect on any of the functions in the **File** pull-down menu, such as **Save**, **Save as**, and **Save and generate**, which you cannot undo.

Constructing a GUI: the OASearch Application

The remainder of this chapter describes how to visually construct the graphical user interface (GUI) of an application using the OASearch application as an example. You construct a GUI by adding visual parts, such as push buttons, lists, and menus, to a window part. For a list of the visual parts available on the Composition Editor, see “The Parts Palette” on page 44.

Developing Applications

Adding Basic Visual Parts

The purpose of this section is to guide you through constructing a version of the Opportunities Aboard Contract Information window for the OASearch sample application. You construct this part (ContractView) in the Composition Editor using the basic visual parts. When you are finished, the Opportunities Aboard Contract Information window looks like Figure 62.



Figure 62. Completed ContractView Window

Adding the Parts

Before starting this new visual part, make sure that `oanonvis.vbb` is loaded into Visual Builder.

Adding the parts

1. Because you are constructing a new user interface, begin by creating a visual part. Name it ContractView.

When you create a visual part, the Composition Editor opens a part that inherits from `IFrameWindow*`. By default, the client area of the `IFrameWindow*` appears as a canvas.

2. Change the title of the window.
3. Add the following parts to the window part:
 - Static text and entry field parts for Account Number, Company Name, Project Manager, and Department.

Developing Applications

- A group-box part for the Position Details group. Resize the group box.
- Static-text and entry-field parts for the items inside the group box.
- Push-button parts.

See Figure 62 on page 158 to see how the window should look after you add the parts.

Changing the Labels

Directly edit the static text so that the text matches Figure 62 on page 158.

Arranging the Parts

Arrange the parts on the frame window part so they are aligned and positioned as shown in Figure 62 on page 158.

For more information on how to perform these operations, see “Working with Parts on the Free-Form Surface” on page 124.

Changing the Settings

Now that the parts have been added for the user interface, customize the parts by modifying their settings.

Specifying a push button label with mnemonic

In the **Text** field on the **General** settings page for the second push button, type `~Save`. Select the **OK** push button.

The second push button of your window now reads **Save**, with a mnemonic of *S*. This means the user can select the push button by pressing Alt+S.

Note: The tilde is the mnemonic character in OS/2. In Windows, this character resolves at generation time to an ampersand (&), which is the mnemonic character in Windows. You can enter the ampersand directly instead of using the tilde, but your mnemonics will not be portable between OS/2 and Windows.

Repeat this procedure for the push button to the right of the **Save** push button, specifying `C1~ear` as the push button label. This push button now reads **Clear**, with a mnemonic of *E*.

For the rightmost push button, specify `~Cancel` as the label. This push button now reads **Cancel**, with a mnemonic of *C*.

Developing Applications

Specifying the default push button

In the **Text** field on the **General** settings page for the leftmost push button, type ~Refresh. Select the **Apply** push button. The leftmost push button of your window now reads **Refresh**, with a mnemonic of *R*.

On the **Styles** settings page, select the **On** radio button for the defaultButton style. Select the **OK** push button. The **Refresh** push button is now the default button for the window. This means the user can select the push button by pressing the Enter key.

For details on the settings of a particular part, refer to the *Visual Builder Parts Reference*. For more information on how to perform these operations, see “Changing Settings for a Part” on page 138.

Adding a Variable to a Composite Part


Variables serve an important role in complex applications. Use variables instead of parts in the following situations:


- To act as a placeholder inside a composite part for parts not found in the composite part. Using variables for nonvisual parts lets you pass data or function between composite parts.
- To represent part instances created with object factory parts. For more information on using object factory parts, see “Adding Visual Parts as Dynamic Instances” on page 264.

Each variable must be set to the part type it represents. Once set, the variable takes on the interface of that part type. Unlike using a part, using a variable does not cause the part constructor to run.

Begin by opening the ContractView part (found in contract.vbb) in the Composition Editor. If you prefer to use the OAContractView part, you can find this part in oawin.vbb. At this point, the ContractView part appears as shown in Figure 62 on page 158.

We are using a variable instead of an OAContract* part because the contract data varies with the information entered from the OAQueryContract* part.

To add a variable to the free-form surface, select  , the **Models** category,

from the parts palette; then add  , an IVBVariable* part, to the free-form

Developing Applications

surface. Change its name to Contract. The ContractView with a variable added appears as shown in Figure 63 on page 161.

Opportunities Aboard Contract Information

Account Number				
Company Name				
Project Manager				
Department				
Position Details				
Title				
Start Date		End Date		
Contractor				

Refresh Save Clear Cancel

Contract

Figure 63. ContractView Window with Variable



Notice the square bracket symbols around the variable part. You can always identify a variable by these symbols. Also note that tear-off attributes also have square brackets and are in fact variables with a special connection.

Changing the Variable's Type

When you add a variable part, its type is initially `IStandardNotifier*`, meaning that the variable can stand for any part. This variable is supposed to stand for an `OACContract*` in this example, so you must change its type.

Note: You cannot use variables to represent template-based parts, such as `IVSequence*`.

1. First, make sure that the part file holding the `OACContract` part (`oanonvis.vbb`) is loaded into Visual Builder.
2. Select **Change type** on the variable's pop-up menu; then type `OACContract*` in the field.

Developing Applications

Adding the Variable to the Part Interface

So far, the Contract variable is empty. To make the variable available for passing values from the main view (or any part outside ContractView), we must add the variable to the part interface of ContractView). Do this by selecting **Promote part feature** from the variable's pop-up menu. See "Promoting a Part's Features" on page 153 if you need information about doing this.



Chapter 9. Learning to Use Connections

This chapter describes the types of connections that you can make and how to make them. Each connection description provides the following information:

- A definition of the connection
- The color of the connection
- Whether the connection is unidirectional or bidirectional
- Whether the connection requires you to supply values to complete it.

Attribute-to-attribute connection

An *attribute-to-attribute connection* links two attribute values together. The purpose of this type of connection is to cause the value of one attribute to change when the value of another attribute changes, except as noted in Figure 65 on page 164.

An attribute-to-attribute connection uses a bidirectional, dark blue line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source.

Note: In Windows NT, this connection appears with small diamonds at either end.

When your part is constructed in the running application, the target attribute is set to the value of the source attribute. Attribute-to-attribute connections never take any parameters.

Usage Notes:

1. You can use a class interface part as the source of a connection only when making an attribute-to-attribute connection. An attribute of a class interface part can be used to initialize an attribute of another part without using notification.
2. Do not create attribute-to-attribute connections in which the source attribute is not initialized before the *ready* event is signaled. You may get a system error in the resulting application. See “The Ready Event” on page 64 for more information.

In Figure 64 on page 164, the *text* attribute of the entry field is connected to the *accountNum* attribute of the Contract variable. This connection causes the value of the *accountNum* attribute to change whenever the value of the *text* attribute changes, and vice versa.

Developing Applications



Figure 64. Attribute-to-Attribute Connection

The effect of attribute types on connections

It is important to know the types of attributes that you are connecting. Otherwise, you might not achieve the results that you anticipate.

Figure 65 shows the results of connecting attributes that have different behavior types. See “The Attribute Page” on page 57 for descriptions of the attribute types.

Figure 65. Source and Target Considerations for Attribute Types

If the source is a...	And the target is a full attribute...	And the target is a no-set attribute...	And the target is a no-event attribute...
full attribute	All attribute behaviors are available to both the source and target attributes.	Visual Builder automatically reverses the connection.	The target attribute cannot notify the source attribute when the target attribute's value changes.
no-set attribute	The source attribute initializes the target attribute. The target attribute is updated whenever the source attribute's value changes.	This is an invalid connection.	The source attribute initializes the target attribute. The target attribute is updated whenever the source attribute's value changes, but the target attribute cannot notify the source attribute when the target attribute's value changes.
no-event attribute	The source attribute initializes the target attribute; no event notification occurs.	Visual Builder automatically reverses the connection.	The source attribute initializes the target attribute; no event notification occurs.

Event-to-attribute connection

An *event-to-attribute connection* enables the occurrence of the source event to trigger a change in the value of the target attribute. To accomplish this, the connection calls the attribute's set member function whenever the event occurs. If the attribute is a no-set attribute, you cannot make the connection. If you open settings on a connection of this type, the target of the connection appears to be an action with the same name as the target attribute.

An event-to-attribute connection uses a unidirectional dark green arrow with the arrow head pointing to the target. If the attribute's set member function requires you to supply parameter values, the connection line is initially dashed to show that it is incomplete unless event data is provided, causing it to turn solid.

You can supply the missing parameter value or override any event data that is present by connecting the parameter to an attribute, action, member function, or custom logic, or by supplying a constant parameter value. See “Supplying Parameter Values for Incomplete Connections” on page 175 for more information.

In Figure 66, the *buttonClickEvent* feature of the **Refresh** push button is connected to the *text* attribute of the entry field.



Figure 66. Event-to-Attribute Connection

Event-to-action connection

An *event-to-action connection* causes an action to start whenever the source event occurs.

An event-to-action connection uses a unidirectional, dark green arrow with the arrow head pointing to the target. If the action requires you to supply parameter values, the connection line is initially dashed to show that it is incomplete unless event data is provided, causing it to turn solid.

You can supply the missing parameter value or override any event data that is present by connecting the parameter to an attribute, action, member function, or custom logic, or by supplying a constant parameter value. See “Supplying Parameter Values for Incomplete Connections” on page 175 for more information.

Also, the action that is the target of this connection can have a return value. If it does, you can treat the return value as a no-set attribute of the connection and use it

Developing Applications

as the source of another connection. The return value appears in the connection menu for the connection as *actionResult*.

In Figure 67, the *buttonClickEvent* feature of the **Add** push button is connected to the *addAsLast* action of the multiline edit control.

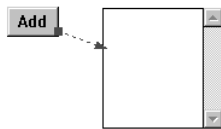


Figure 67. Event-to-Action Connection

Event-to-member function connection

An *event-to-member function* connection calls a member function of the part currently being edited whenever the event that the member function is connected to occurs.

An event-to-member function connection uses a unidirectional, light green arrow with the arrow head pointing to the free-form surface. If the member function requires you to supply parameter values, the connection line is initially dashed to show that it is incomplete unless event data is provided, causing it to turn solid.

You can supply the missing parameter value or override any event data that is present by connecting the parameter to an attribute, action, member function, or custom logic, or by supplying a constant parameter value. See “Supplying Parameter Values for Incomplete Connections” on page 175 for more information.

Also, the member function that is the target of this connection may have a return value. If it does, you can treat the return value as a no-set attribute of the connection and use it as the source of another connection. The return value appears in the connection menu for the connection as *actionResult*.

In Figure 68 on page 167, the *buttonClickEvent* feature of the **Clear** push button is connected to a *resetFields* member function.

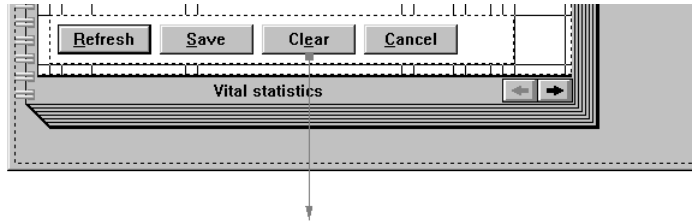


Figure 68. Event-to-Member Function Connection

Attribute-to-action connection

An *attribute-to-action* connection causes an action to start whenever the event identification that is associated with the attribute is triggered. This connection is similar to an event-to-action connection because the connection calls the action whenever the attribute's event is triggered.

An attribute-to-action connection uses a unidirectional, dark green arrow with the arrow head pointing to the target. If the action requires more than one input parameter, the connection line initially appears dashed to show that it is incomplete.

The attribute's value is passed as the first parameter of the action if no parameter is explicitly specified. You can supply any other missing parameter values or override the attribute value that is present by connecting the parameter to an attribute, action, member function, or custom logic, or by supplying a constant parameter value. See "Supplying Parameter Values for Incomplete Connections" on page 175 for more information.

The action that is the target of this connection can have a return value. If it does, you can treat the return value as a no-set attribute of the connection and use it as the source of another connection. The return value appears in the connection menu for the connection as *actionResult*.

In Figure 69, the *text* attribute of the entry field is connected to the *addAsLast* action of the list box.

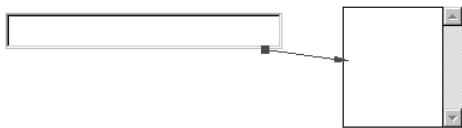


Figure 69. Attribute-to-Action Connection

Developing Applications

Attribute-to-member function connection

An attribute-to-member function connection causes a member function to start whenever the event identification that is associated with the attribute is triggered. This connection is similar to an event-to-member function connection because the connection calls the action whenever the attribute's event is triggered.

An attribute-to-member function connection uses a unidirectional, light green arrow with the arrow head pointing to the free-form surface. If the member function requires more than one input parameter, the connection line initially appears dashed to show that it is incomplete.

The attribute's value is passed as the first parameter of the action if no parameter is explicitly specified. You can supply any other missing parameter values or override the attribute value that is present by connecting the parameter to an attribute, action, member function, or custom logic, or by supplying a constant parameter value. See "Supplying Parameter Values for Incomplete Connections" on page 175 for more information.

The member function that is the target of this connection can have a return value. If it does, you can treat the return value as a no-set attribute of the connection and use it as the source of another connection. The return value appears in the connection menu for the connection as *actionResult*.

In Figure 70, a feature of the list box is connected to a member function that calculates the presentation space in the list box.

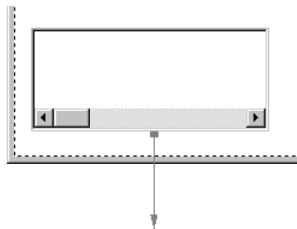


Figure 70. Attribute-to-Member Function Connection

Custom logic connection

A custom logic connection causes your customized logic to run whenever either of the following happens:

- The value of the attribute that you connect it to changes
- The event that you connect it to occurs

Developing Applications

If you connect an attribute to custom logic, the custom logic runs whenever the event associated with the attribute is signaled.

A custom logic connection uses a unidirectional, light blue arrow with the arrow head pointing to the target.

Also, the custom logic that is the target of this connection may have a return value. If it does, you can treat the return value as a no-set attribute attribute of the connection and use it as the source of another connection. The return value appears in the connection menu for the connection as *actionResult*.

In Figure 71, the *buttonClickEvent* of the **Clear** push button is connected to custom logic that clears all entry fields in a notebook. Otherwise, you would need to make a separate connection from the push button's *buttonClickEvent* feature to the *clear* action of each entry field.

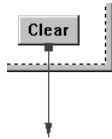


Figure 71. Custom Logic Connection

Parameter connections

A parameter connection supplies a parameter value to an action or member function by passing either an attribute's value or the return value from an action, member function, or custom logic. This connection looks similar to an attribute-to-attribute connection; it uses a bidirectional line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source. The difference you see on your screen is that parameter connections are violet instead of dark blue.

In addition, the parameter names are included in the connection menu. Therefore, if you are in doubt about a connection that you want to make, you can browse a part's features to see the parameter names. Be aware, however, that any parameter names that you specified for an action in the **Parameter names** table on the **Action** page of the Part Interface Editor appear in the connection menu instead of the actual parameter names.

The parameter is always the source of the connection because the parameter cannot store any values. If you connect an attribute, action, member function, or custom logic to a parameter, Visual Builder reverses the direction of the connection to make the parameter the source.

Developing Applications

Whenever the parameter needs a value, Visual Builder attempts to supply one, as follows:




- If the parameter is connected to an attribute, the connection calls the attribute's get member function to get the attribute's value and return it to the parameter.
- If the parameter is connected to an action, the connection code calls the action and passes the action's return value to the parameter. The same is true when a parameter is connected to a member function or to custom logic.
- You can supply a constant parameter value as a setting within the connection. See "Supplying Parameter Values for Incomplete Connections" on page 175 for more information.

If you connect a parameter to two different attributes, the first attribute that you connect the parameter to has precedence over the second. You can change this, if necessary, by reordering the connections or deleting one of the connections.

Visual Builder uses a dashed line to give you a visual cue so that you know when a parameter connection is needed. For example, if you connect an event to a member function that requires parameter values, the connection line between the event and the member function is dashed. See "Supplying Parameter Values for Incomplete Connections" on page 175 for more information.






Connection Type Summary

The following table summarizes the types of connections that Visual Builder provides:

If you want to...	Use this connection type	Color	Arrows	Return value allowed?
Cause one data value to change another	attribute-to-attribute	Dark blue		No
Change a data value whenever an event occurs	event-to-attribute	Dark green		No
Call an action whenever an event occurs	event-to-action	Dark green		Yes ¹

¹ The return value is supplied by the connection's *actionResult* attribute.

Developing Applications

If you want to...	Use this connection type	Color	Arrows	Return value allowed?
Call a member function whenever an event occurs	event-to-member function	Light green		Yes ¹
Call an action whenever a data value changes	attribute-to-action	Dark green		Yes ¹
Call a member function whenever a data value changes	attribute-to-member function	Light green		Yes ¹
Call customized code whenever a data value changes or an event occurs	custom logic	Light blue		Yes ¹
Supply a value to a parameter	parameter	Violet		No

Making the Connections

In this section, you learn how to make attribute-to-attribute, event-to-attribute, event-to-action, and attribute-to-action connections.

Member function connections are discussed in “Connecting Features to Member Function Connections” on page 179. Custom logic connections are discussed in “Connecting Features to Custom Logic” on page 184.

Determining the Source and Target

A connection is directional; it has a source and a target. The direction in which you draw the connection determines the source and target. The part on which the connection begins is the *source* and the part on which it ends is the *target*.

When you make an event connection, Visual Builder draws an arrow on the connection line between the two parts. The arrow points from the source to the target. If information can pass through the connection in both directions, as it can in an attribute-to-attribute connection, a hollow circle indicates the source and a solid circle indicates the target.

Developing Applications

Often, it does not matter which part you choose as the source or target, but there are connections where direction is important.

- With an attribute-to-action, event-to-action, or event-to-attribute connection, the event is always the source and the action or attribute is always the target. In the case of an attribute-to-action connection, the source event is signaled when the attribute changes value. If you try to make an action-to-attribute, action-to-event, or attribute-to-event connection, Visual Builder automatically reverses it for you.
- For attribute-to-attribute connections, if only one of the attributes has a set member function, Visual Builder makes that attribute the target. This is done so that the attribute that has the set member function can be initialized when the application starts.
- When you make attribute-to-attribute connections, the order in which you choose the source and target is important. The source and target attribute values are probably different when your view is first initialized. If they are, Visual Builder resolves the difference by changing the value of the target attribute to match that of the source attribute. Thereafter, if both attributes have set member functions, the connection updates either attribute if the other changes.

Refer to the attributes, actions, and events sections of the particular part in the *Visual Builder Parts Reference* for information that is specific to a part's features.

Browsing a Part's Features

Sometimes it is useful to browse a part's features before using them in a connection. For example, you might want to look at an attribute to see if it has a set member function so that it can update itself when it receives new data from another attribute.

By using **Browse part features**, you can see all of a part's features in one window and browse, but not change, the information about each feature. To modify a feature, use the Part Interface Editor.

There is an important distinction between browsing a part's features and displaying its features for making a connection. When you browse a part's features, you see all of its features, even if some of them are not available for connections. (This includes inherited features if the base parts are loaded into Visual Builder.) When you display a part's connection menu, however, you see only those features that are available for connections.

To browse the features of a part, do the following:

1. Move the mouse pointer over the part and click mouse button 2. Visual Builder displays the part's pop-up menu.
2. Select **Browse part features**.

Developing Applications

Visual Builder displays a browse window that contains three columns: one for actions, one for attributes, and one for events. For example, Figure 72 on page 173 shows the browse window that Visual Builder displays for browsing the features of a push button:

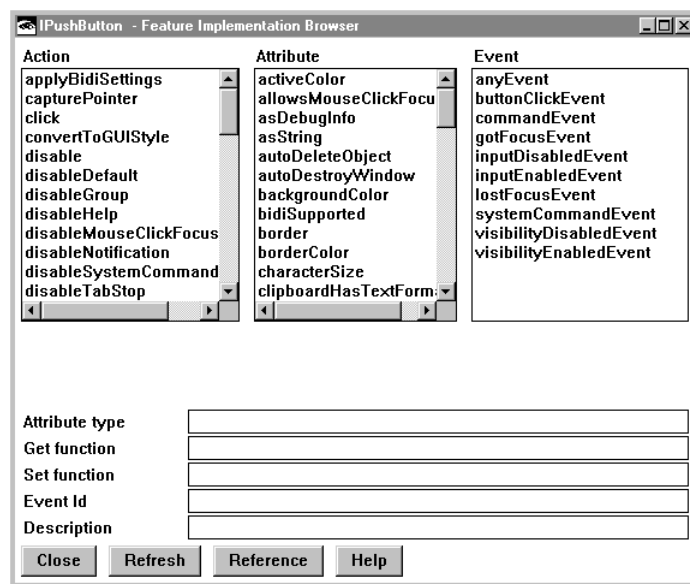


Figure 72. Browse Part Features Window

3. Select the feature you want to browse.

Visual Builder displays information about the feature that you select in the entry fields below the feature columns. Different sets of entry fields are displayed depending on whether you select an action, an attribute, or an event.

The information that Visual Builder displays when you browse a part's features is the same as the information that you would see in the Part Interface Editor. See "The Part Interface Editor" on page 56 to learn about the information that Visual Builder displays for features.

4. If the information about a part's features should change, you can select the **Refresh** push button to see those changes reflected in the browse window.
5. To read more information about the part you are browsing, select the **Reference** push button.
6. When you have finished browsing the features, select the **Close** push button to close the browse window.

Developing Applications

Connecting Features to Features

Follow these steps to connect features:

1. Position the mouse pointer over the *source*, the part or connection that you want to connect from, click mouse button 2, and select **Connect** from its pop-up menu.



To display the connection pop-up menu more quickly, hold down the Alt key while clicking mouse button 2. On Windows, be sure to release the Alt key as soon as you have clicked the mouse.

A menu appears showing the names of the most commonly used attributes, actions, and events, called the *preferred features*. If the source is a part, there is usually a **More** selection at the bottom of the list.

If the **More** selection is not there, this means the list contains all of the available features, not just the preferred ones, and there are no more from which to select.

2. Do one of the following:
 - If the feature you want appears in the list, select it.
 - If the feature you want does not appear in the list, but the **More** selection is available, select **More** and then select the feature you want from the complete list of features.
 - If the feature you want does not appear in either the preferred list or the expanded list that is displayed when you select **More**, you can edit the part to add the feature you need. For more information about this, see “Editing Parts Placed on the Free-Form Surface” on page 151.



If, at this point, you decide not to complete the connection, do one of the following:

- If a pop-up menu is displayed, move the mouse pointer away from the connection menu and click mouse button 1.
- If a window showing all of the features is displayed, select the **Cancel** push button at the bottom of the window.

The menu or window disappears and the connection is not completed.

3. Position the mouse pointer over the part or connection that you want to connect *to*.

While moving the mouse, notice that a dashed line trails from the mouse pointer to the source of the connection.

Developing Applications

4. Click mouse button 1 and a pop-up menu appears, again showing a list of features.
5. Select a name from the pop-up menu or from the **More** list. The same instructions regarding the presence of **More** apply as described previously.

A colored connection line appears when both ends of the connection have been made. The color indicates the connection's type, based on the selections you made in the connection pop-up menu. See "Connection Type Summary" on page 170 for a table that shows the colors that are used for each connection type.

If the line is dashed, it requires parameters, as described in the next section.

Supplying Parameter Values for Incomplete Connections

Event-to-action, attribute-to-action, event-to-attribute, event-to-member function, and attribute-to-member function connections sometimes require parameters, or input arguments. If a connection requires parameters that have not been specified, it appears as a dashed arrow indicating that it is incomplete. When you have made all the necessary parameter connections, the connection line becomes solid indicating that the connection is complete.

Note: Do not make parameter-to-parameter connections. Visual Builder does not prevent you from doing this, but errors occur when you generate the C++ code for your application.

The following sections describe how to complete connections when input parameters are required.

Supplying a parameter value using a connection

One way to supply parameter values is to make connections from the dashed connection lines to the parts that supply the values to the parameters. Most of the time, the values you need are those of attributes of other parts that you are working with in the Composition Editor. Sometimes, however, the value you need is the return value from an action, a member function, or custom logic.

To supply a parameter value, do the following:

1. Start a new connection using the dashed connection line that requires the parameter as the source.
2. For the target, select the attribute, action, member function, or custom logic that is to provide the value that the parameter needs.

Developing Applications



When you make a connection, Visual Builder provides a visual cue to help you position the pointer correctly. When you have the pointer directly over the connection line, a small hollow box appears.

Figure 73 shows an incomplete event-to-action connection. When a user selects the **Add** push button, its *buttonClickEvent* feature notifies the *addAsLast* action of the *IListBox** part to add something to the list box as the last item in the list. The connection is incomplete because the *addAsLast* action has a parameter that needs an input value, which is the text to add to the list box.

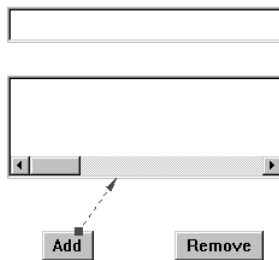


Figure 73. Incomplete Connection Due to Missing Parameter Value

Figure 74 shows how a parameter connection, in which an attribute of the *IEntryField** part is used to supply the input value for the parameter, completes the event-to-action connection shown in Figure 73. The *text* parameter of the incomplete connection is connected to the *text* attribute of the *IEntryField** part. Therefore, when the **Add** push button is selected, its *buttonClickEvent* feature notifies the *addAsLast* action of the *IListBox** part to add the text in the *IEntryField** part as the last item in the list box.

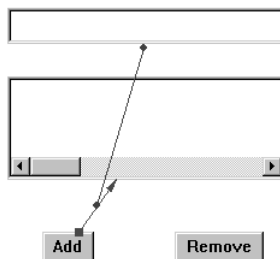


Figure 74. Completing a Connection Using an Attribute as the Input Value

Developing Applications



The return type of an action displays as the *actionResult* attribute of the connection. For example, you can connect the *actionResult* attribute to an attribute of the same part or another part.

Supplying a parameter value using a constant value

When connections need parameters whose input values are constant, you provide these values by using the settings window of the incomplete connection, as follows:

1. Select **Open settings** from the incomplete (dashed) connection's pop-up menu.



A quicker way to open the settings window is to double-click on the connection line.

The settings window of the incomplete connection is displayed.

2. Select the **Set parameters** push button. The Constant Parameter Value Settings window is displayed showing the parameters for which you can set constant values.
3. Enter the constant parameter values you want to use.

Type the parameter values just as you would type them as parameters for the member function if you were coding it by hand. For example, to provide a constant value for a *text* parameter, enter the text string that you want the parameter to receive. Visual Builder copies these values as strings to the output files when you generate your default code.

4. Do one of the following:
 - Select the **OK** push button to apply the values and save them.
 - Select the **Apply** push button if you want to see what effect these values have before saving them.
 - Select the **Cancel** push button to remove the notebook without saving any of the parameter values you entered.

You can select **Help** for additional information about entering parameter values.

Specifying defaults for parameters

There are two places in Visual Builder in which you can specify defaults for the parameters of actions in the declaration of the action's member function:

- In the part's .vbe file
- On the **Action** page of the Part Interface Editor

Developing Applications

Either of these enables a default value to be passed in an event-to-action connection, thus avoiding the need to supply a parameter value.

For example, suppose you want to connect the *buttonClickEvent* feature of a **Remove** push button to a *removeSelected* action that you created for an *IListBox**-based part. Normally, you would also need to connect the *selection* attribute of the *IListBox**-based part to the *index* attribute of the connection between *buttonClickEvent* and *removeSelected*. This connection would be required to get the index of the selected item in the list box.

However, in the .vbe file of the *IListBox**-based part, you can specify the following default parameter value for the *removeSelected* action:

```
//VBAction: removeSelected, "removeSelected",,  
//VB:         removeSelected(unsigned long index=selection())
```

This means that if no attribute of the *IListBox**-based part is connected to the *index* attribute, the selection member function (the get member function of the *selection* attribute) is called by default to provide the index of the selected item.

You could do the same thing by creating a *removeSelected* action on the **Action** page of the Part Interface Editor for the *IListBox**-based part. You would specify the default parameter in the declaration of the *removeSelected* member function in the **Action member function** field as follows:

```
virtual unsigned long removeSelected(unsigned long index=selection())
```

Preventing parameters that are missing or not valid

When an action connection requires arguments, be sure that you make the correct number of parameter connections. Also, be sure that you make the parameter connections before you generate code for your part. If you use the return value of one connection as input to another, make sure the connections appear in the correct order. See “Manipulating Connections” on page 189 to learn how to reorder connections.

Connecting Features to Member Function Connections

A common use for member functions is with event-to-member function connections and attribute-to-member function connections. These connections link an event to a member function so that the member function is called when the event occurs. Event-to-member function connections make this link directly from an event to a member function connection. Attribute-to-member function connections make this link indirectly using the attribute's event identifier, passing the value of the attribute as the first parameter of the member function. The member function that you connect the event or attribute to must be accessible to the part being edited.

For example, suppose you are creating a composite part that consists of several individual parts, such as a frame window, an entry field, a list box, and two push buttons. In this case, you can connect the attributes and events of all individual parts to a member function that belongs to the composite part that you are editing.

When to connect attributes or events to member functions

How do you know when to connect an attribute or event to an action and when to connect it to a member function? After all, an action is simply a member function with a part interface. Therefore, it seems there should be little difference between connecting to one and connecting to the other.

The difference is accessibility. Anyone can use actions that you include in your part's interface, but only you can use the member functions that belong to your part. When you create actions and include them in your part's interface, other programmers who use your part can create their own event-to-action or attribute-to-action connections using the actions that you have defined for the part. That is the reason for including functions in the part's interface as actions—to make them available to other programmers.

However, you might have functions that you want to call for your part without including them in the part's interface. For example, your part might need to perform a calculation internally whenever the value of an attribute changes. However, another programmer who is using your part might not need to be aware that this is happening. This is when you use member functions instead of actions. By using event-to-member function connections and attribute-to-member function connections, you can call these member functions when specific events occur or when attribute values change, but no one else can.

Developing Applications

Adding an Event-to-Member Function Connection

The requirements for connecting an event to a member function are a visual part that has at least one event specified for it, as well as at least one member function.

Figure 75 shows the OACContractorView part used in the OASearch sample application in this book. The steps for adding an event-to-member function connection, which follow the figure, are based on this part. For information about how to construct this part, see “Adding Notebook Parts” on page 237.

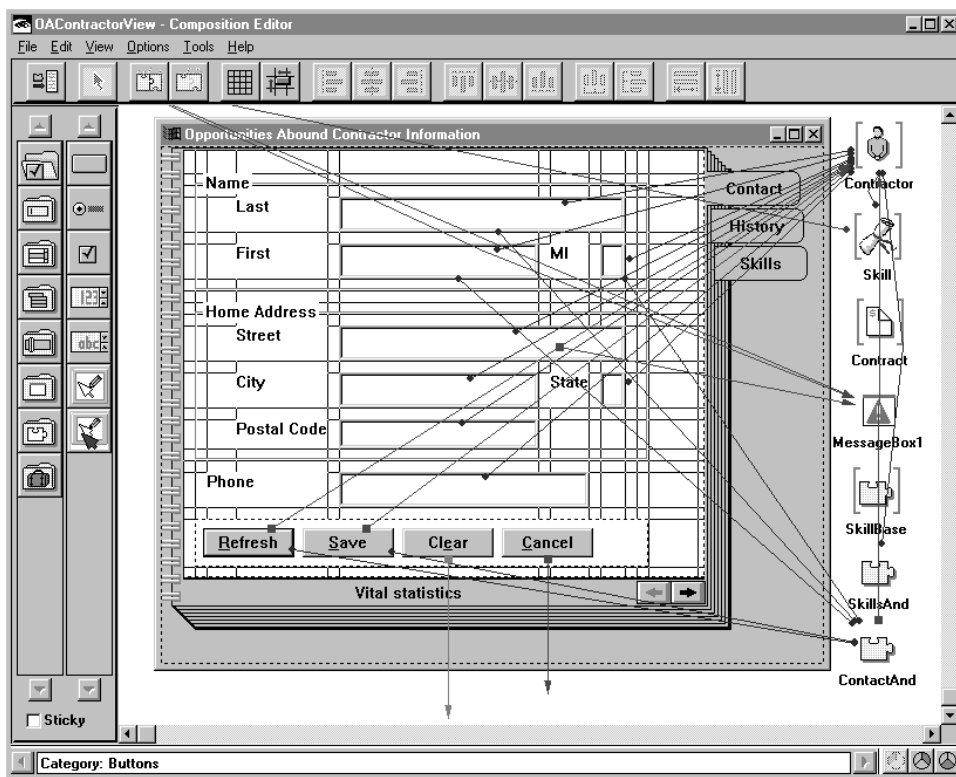


Figure 75. OACContractorView Notebook

1. Open the connection menu for the **Clear** push button.
2. From this menu, select the *buttonClickEvent* choice, an event.
3. Move the mouse pointer to a point on the free-form surface outside the OACContractorView part and click mouse button 1. Doing this indicates to Visual Builder that you want to connect the event to the OACContractorView notebook part instead of one of its subparts.

Developing Applications

Visual Builder displays the connection menu with the preferred features that have been defined for `OACContractorView`. The member function that you want to connect your event to is not on this menu. Member functions are not considered to be preferred features unless you define them as actions in the Part Interface Editor.

4. Select **More**.

Visual Builder displays a window that contains three list boxes. The left and middle boxes show the actions and attributes of the `OACContractorView` notebook that you can connect the `buttonClickEvent` feature to. The third list box is for events; this list box is empty because you cannot connect one event, in this case `buttonClickEvent`, to another event.

Below the three list boxes is another section of the window titled Member Function Connection. This is where you specify the member function that you want to connect the `buttonClickEvent` feature to.

The **For class** field contains the class name of the part that you are editing. This field is actually a drop-down list box. The list box is initially empty. See the definition of the **Browser** pull-down menu item in “Using Browser Information” on page 182 to find out how to display the names of the classes that your part inherits from, as well as how to display the member functions that your part contains in the list box at the bottom of the window.

For this example, the **For class** field contains `OACContractorView`, the part that you are editing.

The **Access** field is another drop-down list box. This field is where you specify the access level (public, protected, or private) for the member functions that you want to see in the list. You must open the Browser data to see a list of member functions. See “Using Browser Information” on page 182 to learn how to do this.

For this example, the **Access** field contains `public`.

5. Type the member function to which the `buttonClickEvent` feature is to be connected in the entry field below the **Access** field. Be sure to type the full declaration of the member function, including return type and parameters.

For this example, enter the following member function to clear all of the fields in `OACContractorView`:

```
void resetFields();
```

You must also create an `.hvp` file that contains this member function declaration and a `.cpv` file that contains the code for this member function. The member function declaration that you enter in this field must be *identical* to the declaration that you put in the `.hvp` file. For information about `.hvp` and `.cpv`

Developing Applications

files, see “Specifying Files to Include When You Build Your Application” on page 55.

In addition, *before* you generate the code for your part, you must switch to the Class Editor and enter the names of these files so that Visual Builder will include them in the code for your part. For more information about including these files, see “Specifying Files to Include When You Build Your Application” on page 55.

6. Select the **OK** push button.

The connection window disappears and a light green connection arrow is drawn from the **Clear** push button toward the edge of the free-form surface. It looks like this:

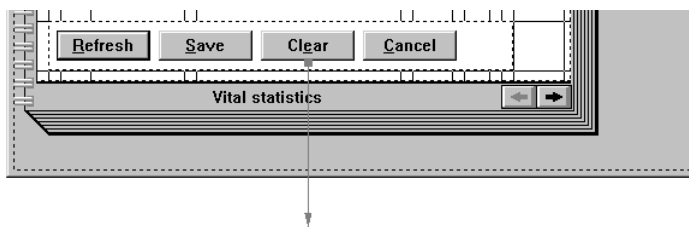


Figure 76. Completed Event-to-Member Function Connection

If the `resetFields` member function had parameters, the connection arrow would be dashed instead of solid to give you a visual cue that you would need to provide values for the parameters. For information about the various ways to do this, see “Supplying Parameter Values for Incomplete Connections” on page 175.

Using Browser Information

When connecting attributes or events to member functions, you might find it helpful to see the classes and member functions in your inheritance tree, as well as member functions that your part contains. You can get this information by selecting **File→Browser** when using any of the Visual Builder editors. This menu choice provides access to a list of the classes that your part inherits from.

You can use this list when you make attribute-to-member function and event-to-member function connections. The window that Visual Builder displays when you are making these connections contains a drop-down list box titled **For class**. In this list box, you can select one of the classes that your part inherits from. This causes Visual Builder to display a list of the member functions that the class contains in another list box at the bottom of the connection window.

The connection window is modal. Therefore, to display browser data in this window, you must open browser data before you begin the connection. Otherwise, the **For**

Developing Applications

class drop-down list box contains only the class name of your part, and the member function list box can display only the member functions that your part contains. You can open browser data by selecting either **Open browser data** or **Quick browse** from the **File→Browser** menu.

The VisualAge for C++ compiler generates the browser information when you have compiled your application. Therefore, except for the **Quick browse** menu choice, browser information is not available until *after* you compile your application.

The VisualAge for C++ compiler stores the browser information in a file whose name is based on the part name and whose extension is .pdb. To generate this file, you must include the -Fb+ compiler option in your make file.

The cascaded menu that Visual Builder displays when you select **File→Browser** contains the following selections:

Quick browse

Causes Visual Builder to retrieve a list of the classes that the part you are editing inherits from, as well as a list of the member functions that your part and the part it inherits from contains. This list appears in the **For class** list box when you open the connection window. If you select one of the classes in the list, Visual Builder retrieves a list of the member functions that the class contains.

This function is available only if you start Visual Builder from a WorkFrame project. It gives you access to browser information before you compile your application.

Open browser data

Works the same as **Quick browse** except you must compile your application with the -Fb+ option before using this function. Also, you do not have to start Visual Builder from a WorkFrame project to use this function.

Refresh browser data

Refreshes the list that is displayed in the member function list box for the current class. If you recompile your application, use this function afterwards to ensure the data in the list boxes is current.

Close browser data

Removes the list of the classes that the part you are editing inherits from, as well as the member functions those classes contain.

Connecting Features to Custom Logic

Custom logic connections enable you to connect small, specialized code snippets to events and attributes. These connections link an event to your code so that it is called when the event occurs. Custom logic connections that use events make this link directly from an event to the custom logic; custom logic connections that use attributes make this link indirectly using the attribute's event identifier.

Use custom logic for unique logic requirements, because the logic is stored as part of the connection and is not reusable.

Adding a Custom Logic Connection

The requirements for connecting an event or attribute to custom logic are a visual part that has at least one event or attribute specified for it. Following are the steps to follow for adding a custom logic connection. Figure 77 shows the **History** page of the OAContractorView part used in the OASearch sample application in this book. The example used in the following steps is based on this part.

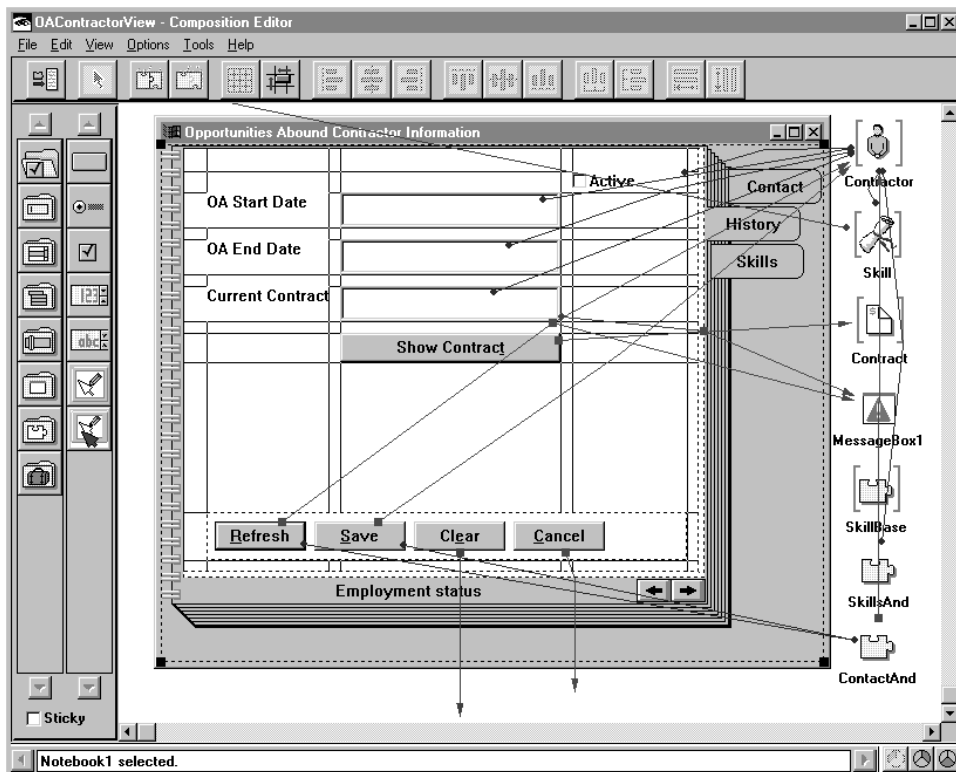


Figure 77. OAContractorView Part

Developing Applications

1. Select the right arrow in the lower-right corner of the notebook to display the **History** page.
2. With the mouse pointer over the **Clear** push button, hold down the Alt key and click mouse button 2.

Visual Builder displays the connection menu for the **Clear** push button.

3. From this menu, select the *buttonClickEvent* choice, an event.
4. Move the mouse pointer to a point on the free-form surface outside the OACContractorView part and click mouse button 1. Doing this indicates to Visual Builder that you want to connect the event to the OACContractorView part instead of one of its subparts.

Visual Builder displays the connection menu with the preferred features that have been defined for the OACContractorView part.

5. Select **Custom logic**.

Visual Builder displays the following window:

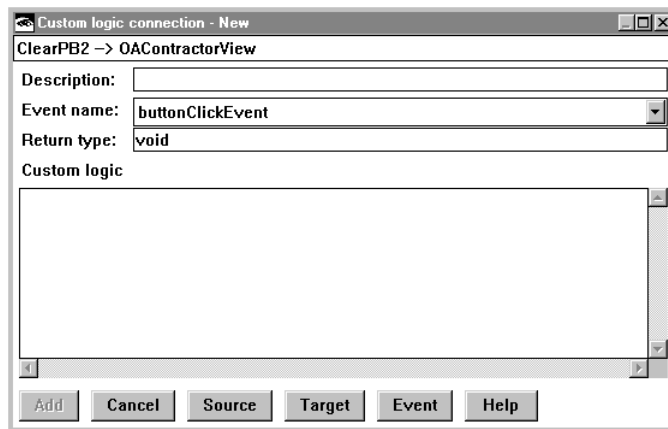


Figure 78. Window for Connecting Event to Custom Logic

6. Enter the following in the **Description** field:

Sample to clear entry fields

The **Description** field serves two purposes. After you complete the connection, if you select the connection line, Visual Builder displays the text that you entered in this field in the information area at the bottom of the Composition Editor.

When you generate the code for this part, Visual Builder inserts this description in the ITRACE_DEVELOP statement that appears in the dispatchNotificationEvent member function for the connection class. When you

Developing Applications

run the compiled application with tracing set on, this description is written to the trace destination whenever the connection code runs.

7. The *buttonClickEvent* feature is the event you want to connect to your custom logic, so do not change the **Event name** field for this example.

This field contains the event that you selected previously from the connection menu. You can change this by selecting an event name from the drop-down list box.

8. The default return type of void is the return type you want to use, so do not change the **Return type** field for this example.

This field contains the return type for your custom logic. If you leave void as the return type, your custom logic will have no return value. If you want to pass a return value from your custom logic, be sure to change the return type to the type of data that you want to pass.

9. Select the **Target** push button.

Visual Builder inserts `target->` in the **Custom logic** field.

The **Custom logic** field is a multiline entry field. This is where you enter your custom logic. You can specify whether the code you enter here is to affect the source part or the target part by selecting the **Source** or **Target** push button before entering a line of code.

You can also select the **Event** push button to insert a reference to the source of the connection (an *INotificationEvent*). This enables you to access the data that is passed by the event. For example, you might use this to test for specific event data and then do different processing based on the result of the test.

In the generated code for the edited part, the entry fields that we want to clear are created using the *new* operator. For the StartEF part, the resulting pointer is called *iStartEF*.

10. Type the following to the right of `target->`:

```
iStartEF->removeAll();
```

This code calls the `removeAll` member function for the StartEF part, the entry field labeled 0A Start Date.

11. Press the Enter key to move the cursor to a new line.

12. Select the **Target** push button again.

13. Type the following to the right of `target->`:

```
iEndEF->removeAll();
```

This code calls the `removeAll` member function for the EndEF part, the entry field labeled 0A End Date.

Developing Applications

14. Press the Enter key to move the cursor to a new line.
15. Select the **Target** push button again.
16. Type the following to the right of target->:

```
iContractEF->removeAll();
```

This code calls the removeAll member function for the ContractEF part, the entry field labeled Current Contract.

The connection window should now look like the following figure:

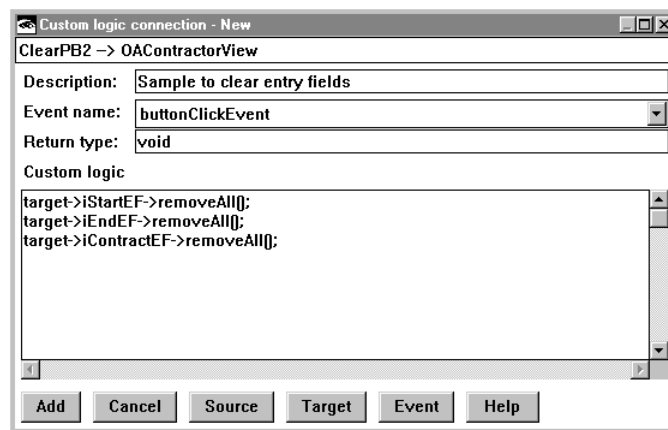


Figure 79. Completed Custom Logic Connection Window

17. Select the **Add** push button.

The connection window disappears and a blue connection arrow is drawn from the **Clear** push button toward the edge of the free-form surface. It should look like this:

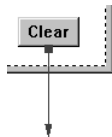


Figure 80. Completed Custom Logic Connection

Connecting Exception Events to Actions and Member Functions

In C++, an *exception* is any user, logic, or system error detected by a function that does not deal with the error itself but passes the error on to a handling routine, called an *exception handler*. In Visual Builder, you can catch exceptions by connecting an exception event to either an action or a member function.

An exception event is a feature of a connection, not a part. It is typically connected to an `IMessageBox*` part, which is used to display an error message associated with the exception.

To connect an exception event, do the following:

1. Use your favorite text editor to create an action or member function that throws an exception.

The easiest way to do this is to include the following in your action or member function source code:

```
throw IException("Error message.");
```

where the text of the error message that you want to display in the message box is the only parameter given for the `IException` constructor.

For example, the following `isEntryFieldEmpty` member function uses the `strcmp` function to check the contents of an entry field in the To-Do List application. The application adds the contents of the entry field to the to-do list whenever the user clicks the **Add** push button. If the entry field is empty and the user clicks on **Add**, the member function throws an exception, which consists of an error message.

```
void ToDoList::isEntryFieldEmpty()
{
    if (strcmp(this->iEntryField->text(), "") == 0)
        throw IException("Type a to-do item in the entry field
                           before selecting Add.");
}
```

2. Connect an event to the action or member function you just created.

The event that you connect here is the event for which you want to throw an exception if an error occurs. In this example, you want to throw an exception if the user clicks on **Add** when the entry field is empty. Therefore, you would connect the `buttonClickEvent` feature of the **Add** push button to the `isEntryFieldEmpty` member function.

3. Drop an `IMessageBox*` part on the free-form surface.

This message box is used to display the error message.

4. Move the mouse pointer to the connection you just made.

5. While holding down the Alt key, press mouse button 2.
6. Select the *exceptionOccurred* event from the connection menu.
7. Move the mouse pointer to the *IMessageBox** part.
8. Click mouse button 1.
9. Select the *showException* action from the connection menu.

This connection causes the application to show a message box that contains the exception error message whenever the exception is thrown.

For an example using the *OACContractorView* part from the *OASearch* application, see “Passing Exceptions to Message Boxes” on page 203.

Manipulating Connections

Once a connection is made, you can manipulate it by doing the following:

- Changing settings for a connection
- Reordering connections
- Deleting connections
- Showing and hiding connections

Changing Settings for a Connection

The settings window of a connection provides a way for you to select different features as the source and target of the connection. If a member function is the target of the connection, this window enables you to specify or select a different member function as the target.

Note: If you are changing the settings of a connection whose target is a member function, you might want the connection settings window to show the classes that your part inherits from, as well as the member functions that those classes contain. To display the class and member function names, you must open the browser data before opening this settings window. See “Using Browser Information” on page 182 for information about opening browser data.

To open the settings window for a connection, move the mouse pointer over the connection and do one of the following:

- Double-click mouse button 1.
- Click mouse button 2 and select **Open settings** from the connection’s pop-up menu.

Visual Builder displays different connection settings windows depending on whether the target of the connection is an attribute, action, or member function. The following sections describe these three windows.

Developing Applications

Changing Settings for Attribute-to-Attribute Connections

The following figure shows the window that Visual Builder displays when you open the settings window for an attribute-to-attribute connection:

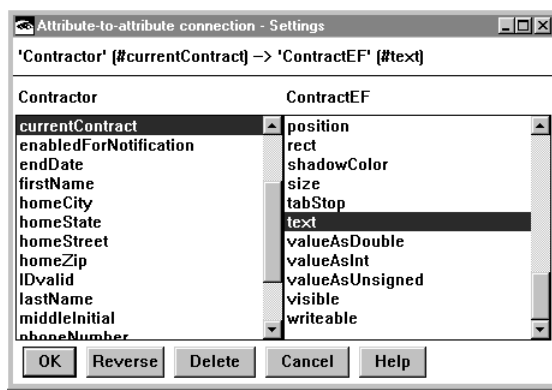


Figure 81. Attribute-to-Attribute Connection Settings Window

The connection settings window for an attribute-to-attribute connection contains two columns of attributes. The left column contains the attributes that belong to the source part. The right column contains the attributes that belong to the target part, excluding any no-set attribute attributes.

To change the attribute to be used as either the source or target of the connection, select an attribute from the list.

This connection settings window has the following push buttons:

OK

Removes the connection settings window and puts any changes made into effect.

Reverse

Reverses the order of the connection; the source part becomes the target and the target part becomes the source.

Delete

Deletes the connection.

Cancel

Removes the connection settings window without putting any changes into effect.

Help

Provides information about the window.

Changing Settings When an Action Is the Target

When you open a settings window for a connection whose target is an action, Visual Builder displays the following window:

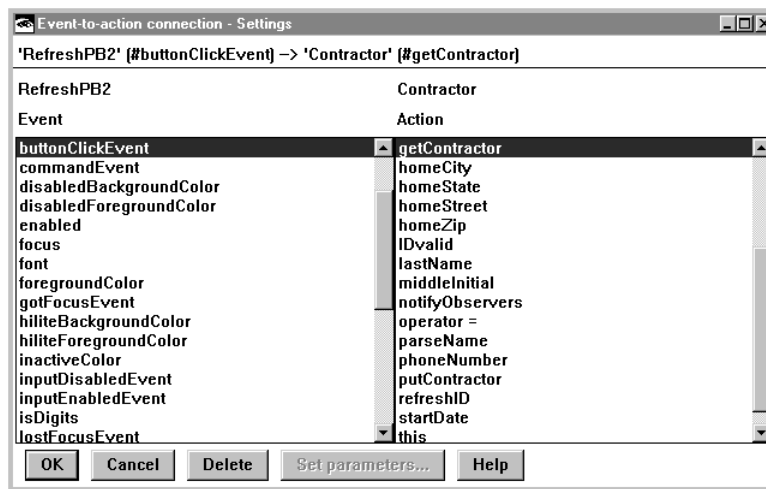


Figure 82. Connection Settings Window with an Action as the Target

The connection settings window contains two columns of features. The features in the left column belong to the source part; these features are the same type of feature as the one currently selected for the source part. For example, if the feature selected for the source part is an event, this column contains a list of the events that belong to the source part, including events associated with attributes of the source part.

Likewise, the features in the right column belong to the target part; these features are all actions or attributes that have set member functions.

The entry fields above each list contain the names of the features that are currently selected. To change the feature to be used as either the source or target of a connection, select a feature from the list.

This connection settings window has the following push buttons:

OK

Removes the connection settings window and puts any changes made into effect.

Cancel

Removes the connection settings window without putting any changes into effect.

Delete

Deletes the connection.

Developing Applications

Set parameters

Opens another window in which you can specify a constant parameter value for each parameter the action has.

Help

Provides information about the window.

Changing Settings When a Member Function Is the Target

Two types of connections can use a member function as the target: attribute-to-member function and event-to-member function connections. The following figure shows the settings window for an event-to-member function connection:



Figure 83. Connection Settings Window with a Member Function as the Target

Changing the name of the source feature

The **Event name** field shows the name of the event that is connected to the member function. If this was an attribute-to-member function connection, this field would be the **Attribute name** field and would contain an attribute name.

To change this name, select an attribute or event from the drop-down list box below the entry field.

Changing the name of the target class

The **For class** field shows the name of the class that the target member function belongs to. This field can contain either the class name of the part that you are editing or the class name of a part that the part you are editing inherits from.

To change this class name, select a class name from the drop-down list box below the entry field if you opened the browser data before opening this window.

Developing Applications

Changing the access type and target member function

The **Access** field shows the current access type, which is either public, protected, or private. Member functions of this access type that belong to the class shown in the **For class** field are displayed in the list box at the bottom of the window. The list includes the currently selected member function, which is shown in the entry field above the list. These member function names are displayed only if you opened the browser data before opening this window. You can open browser data by selecting either **Open browser data** or **Quick browse** from the **File→Browser** menu.

To change the access type, do one of the following:

- Move the mouse pointer over the **Access** field, click mouse button 1, and type the letter p.

Each time you type the letter p, the next access type appears. For example, if the entry field contains public, typing the letter p causes protected to appear.

- Select an access type from the drop-down list box.

To change the member function that is used for the connection, do one of the following:

- Type a different member function in the entry field.
- Select a member function from the list displayed below the entry field if you opened the browser data before opening this window.

To supply a constant parameter value, select **Set parameters**. For more information on specifying parameter values, see “Supplying Parameter Values for Incomplete Connections” on page 175.

The push buttons

The connection settings window for member functions has the following push buttons:

OK

Removes the connection settings window and puts any changes made into effect.

Cancel

Removes the connection settings window without putting any changes into effect.

Delete

Deletes the connection.

Set parameters

Opens another window in which you can specify a constant parameter value for each parameter the member function has.

Developing Applications

Help

Provides information about the window.

Changing Settings When Custom Logic Is the Target

The following figure shows a settings window for a custom logic connection:

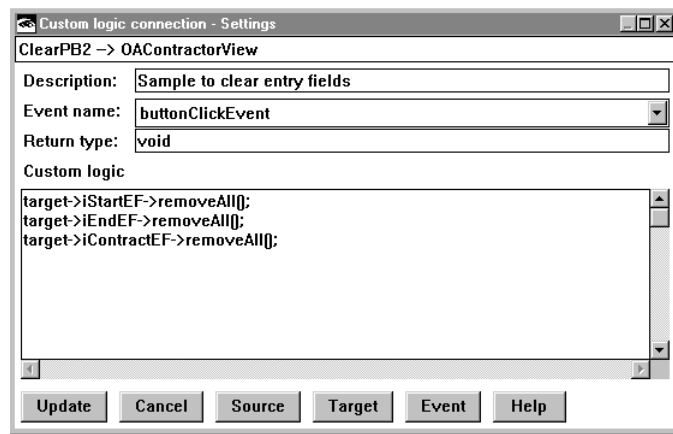


Figure 84. Connection Settings Window with Custom Logic as the Target

This window looks similar to those used to create the connection. See “Adding a Custom Logic Connection” on page 184 for information about how to use this window.

The only push button that differs on this window is the **Update** push button. Use this push button to save any changes you make and close the window.

Reordering Connections

If you make several connections from the same part, they run in the order in which you made the connections. To ensure the correct flow of control when you generate the source code, you may need to reorder the connections. If so, do the following:

1. Select the source part.
2. From the source part’s pop-up menu, select **Reorder connections from**.

Visual Builder displays the Reorder Connections window showing a list of your connections.

3. With the mouse pointer over the connection you want to reorder, press and hold as follows:
 - mouse button 1 in Windows.
 - mouse button 2 in OS/2.

Developing Applications

4. Drag the connection to the place in the list where you want the connection to occur.
5. Release the mouse button.
6. Repeat these steps until the connections are listed in the order in which you want them to occur.
7. Close the window.

Deleting Connections

You can delete a connection in either of the following ways:

- From the connection's pop-up menu.

Note: You do not have to select a connection to delete it using this method.

To delete a connection from its pop-up menu, do the following:


1. Click on the connection with mouse button 2 to display its pop-up menu.
2. Select **Delete**.


- From the connection's settings window

To delete a connection from its settings window, do the following:

1. Open the settings window for the connection by doing one of the following:
 - Double-clicking on the connection
 - Clicking on the connection with mouse button 2 to display its pop-up menu and selecting **Open settings**
2. Select the **Delete** push button.

Showing and Hiding Connections

You can show and hide connections by using  , the **Show Connections** tool,

and  , the **Hide Connections** tool on the Tool bar. These tools show or hide all connections that have the selected part or parts as their source or target.



If you hide connections, the Composition Editor free-form surface is drawn faster and is less cluttered, making it easier for you to work.

Developing Applications

If no parts are selected, these tools show and hide all of the connections on the free-form surface. If at least one part is selected, these tools show and hide the connections from or to the selected parts.

Another way to show and hide connections is to move the mouse pointer over a part, click mouse button 2, and select the **Browse connections** choice from the part's pop-up menu, which displays a cascaded menu. The choices in the menu affect only connections going to and from the part the mouse pointer was over when you displayed the pop-up menu.

The Browse connections cascaded menu contains the following choices:

Show to

Shows all connections for which the part is the target.

Show from

Shows all connections for which the part is the source.

Show to/from

Shows all connections for which the part is either the source or the target.

Show all

Shows all connections that have been made, regardless of where the mouse pointer is when you click mouse button 2.

Hide to

Hides all connections for which the part is the target.

Hide from

Hides all connections for which the part is the source.

Hide to/from

Hides all connections for which the part is either the source or the target.

Hide all

Hides all connections that have been made, regardless of where the mouse pointer is when you click mouse button 2.

Rearranging Connections

You can rearrange a connection by doing the following:

- Selecting connections
- Deselecting connections
- Changing the source and target of connections

Selecting Connections

You select connections in the same way that you select parts. When you select a connection, three boxes called selection handles appear on it to show that it is selected: one at each end and one in the middle. You can use these boxes to change either of the following:

- The end points of the connection, as described in “Changing the Source and Target of Connections” on page 198.
- The shape of the connection line by dragging the middle box to another location. This helps you distinguish among several connection lines that are close together.

Selecting a single connection

1. Move the mouse pointer over the connection you want to select.
2. Click mouse button 1 to select the connection. The connection is selected.

Selecting multiple connections in OS/2

If you want to select several connections, do one of the following:

- To select multiple connections using just the mouse, do the following:
 1. Move the mouse pointer over one of the connections you want to select.
 2. Hold down mouse button 1 instead of clicking it.
 3. Move the mouse pointer over each connection that you want to select.

The selection boxes appear on each connection that the mouse pointer passes over to show they are selected.
 4. After the connections are selected, release mouse button 1.
- To select multiple connections using both the mouse and the keyboard, do the following:
 1. Hold down the Ctrl key.
 2. Move the mouse pointer over a connection.
 3. Click mouse button 1 while the mouse pointer is over the connection line.
 4. Without releasing the Ctrl key, repeat the preceding steps until all connections that you want to select are selected.

Selecting multiple connections in Windows

To select several connections, do the following:

1. Hold down the Shift key.

Developing Applications

2. Move the mouse pointer over a connection.
3. Click mouse button 1 while the mouse pointer is over the connection line.
4. Without releasing the Shift key, repeat the preceding steps until all connections that you want to select are selected.

Deselecting Connections

If you want to deselect a connection without selecting another part or connection, do the following:

1. Move the mouse pointer over the connection line.
2. Hold down one of the following keys:
 - In OS/2, hold down the Ctrl key.
 - In Windows, hold down the Shift key.
3. Click mouse button 1.

Changing the Source and Target of Connections

Visual Builder gives you the ability to change what a connection is pointing to (the target) or pointing from (the source). Of course, you could always just delete the connection and create a new one. However, the following steps show you a quicker way to do this.

Moving either end of a connection when 'hardcopy or online' insert

1. Select the connection.
2. Move the mouse pointer over either selection handle that appears on the ends of the connection.
3. Press and hold mouse button 2.
4. Move the mouse pointer to the new part or connection.
5. Release the mouse button.

Depending on the connection type, you may get a pop-up menu asking you for new information for the connection.

What you can change

You can change the source end of any connection. In most cases, you can also change the target end. However, depending on the feature that you connect to when you make the change, you might get a different type of connection than the one you started with. If you change the target part of a *feature*-to-action connection to a part that does not support the target action, the connection menu appears, and you can select a new action or custom logic.

Developing Applications

Figure 85 gives you a closer look at the connection types and what you can change without changing the connection type:

Figure 85. What Can Be Changed on a Connection, by Type

Connection type	Move either end	Move source end only
attribute-to-attribute	x	
attribute-to-member function		x
attribute-to-action	x	
event-to-action	x	
event-to-attribute	x	
event-to-member function		x
custom logic	x	
parameter connection	x	

Making Connections for the OASearch Application

The following sections show you how to connect the parts that were used to create a GUI for the OAContractView part in “Constructing a GUI: the OASearch Application” on page 157.

Connecting the Variable Part

The Contract variable part was added in “Adding a Variable to a Composite Part” on page 160. Once you have added the variable to the free-form surface and set its name and type, you need to connect it to the OAContractView composite part.

Making the attribute-to-attribute connections

So that recruiters can display and update contract information held by the Contract variable, you must make attribute-to-attribute connections between the composite visual part and the Contract variable part as follows:

From part, feature	To part, feature
Contract,#accountNum	AcctNumEF,#text
Contract,#companyName	CompanyEF,#text
Contract,#projectMgr	ProjMgrEF,#text
Contract,#deptName	DeptEF,#text
Contract,#positionTitle	TitleEF,#text

Developing Applications

Contract,#startDate	StartEF,#text
Contract,#endDate	EndEF,#text
Contract,#currContractor	ContractorEF,#text

So far, the connections appear as shown in Figure 86.

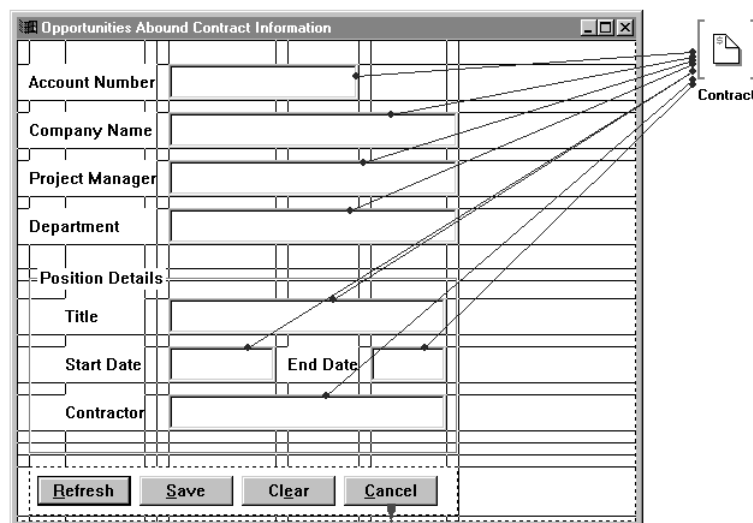


Figure 86. ContractView with Attribute-to-Attribute Connections

Enabling Push Buttons When an Entry Field Contains a Value

This section shows you how to make the connections that enable the **Refresh** and **Save** push buttons in the OAContractView window. When you have made these connections, the window appears as shown in Figure 87 on page 201.

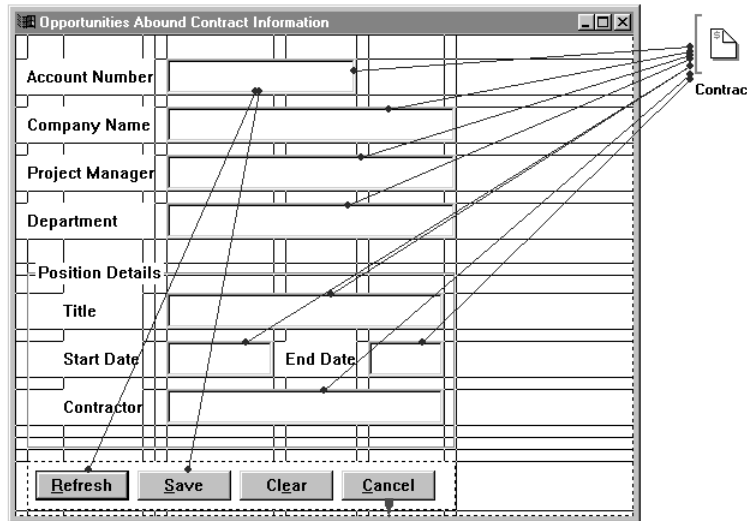


Figure 87. ContractView Window with Variable and Push Button Connections

Enabling the push buttons

1. From the AcctNumEF part's pop-up menu, select **Connect**.
2. Select *textLength* from the list.
3. Click on the **Refresh** push button.
4. Select *enabled* from the pop-up menu.

This connection means that the **Refresh** push button will only be enabled whenever the contents of the **Account Number** text changes, whether by typing text in it or by another connection. Here, you use an attribute-to-action connection because you want the *enabled* action to occur whenever the value of the *textLength* attribute changes.

5. Follow the same procedure to connect the *text* attribute of the AcctNumEF part to the *enabled* action of the **Save** push button part.

Developing Applications

Making the event-to-action connections

To enable push buttons on the composite visual part, you must make event-to-action connections. This is a two-step process, because you specified an input parameter for each of the data access actions. First, connect each push button to the `Contract` variable. The connection line appears dashed, indicating an incomplete connection. Next, connect the attribute that represents the input parameter to the event-to-action connection itself.

For this example, make the following connections:

From part, feature

`RefreshPB,#buttonClickEvent`

`AcctNumEF,#text`

`SavePB,#buttonClickEvent`

`AcctNumEF,#text`

To part, feature

`Contract,#getContract`

The `anAccountNum` parameter of the `RefreshPB,#buttonClickEvent-->Contract,#getContract` connection. Visual Builder draws the parameter connection in the opposite direction.

`Contract,#putContract`

The `anAccountNum` parameter of the `SavePB,#buttonClickEvent-->Contract,#putContract` connection. Visual Builder draws the parameter connection in the opposite direction.

The connections now appear as shown in Figure 88 on page 203.

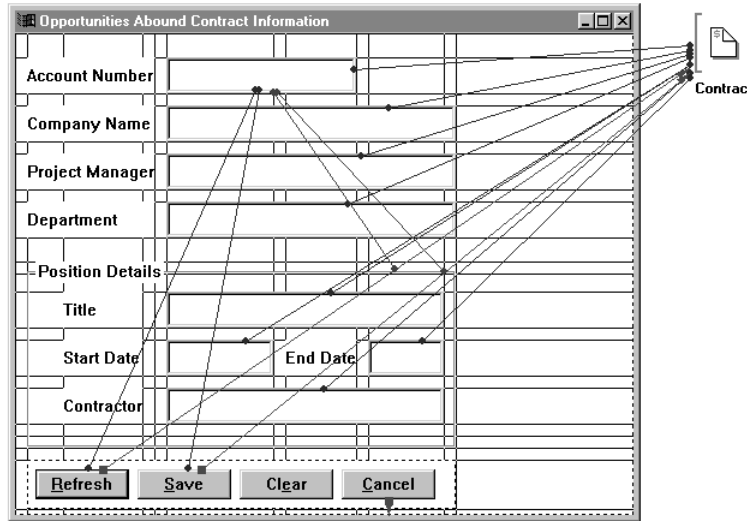



Figure 88. OAContractView with Event-to-Action Connections


The *putContract* and *getContract* actions throw exceptions when the account number is not found in the database.

Passing Exceptions to Message Boxes

You can trigger the display of a message box when an exception is thrown. Although a message box is a user-interface element, Visual Builder treats message boxes as nonvisual parts.

Adding the message box

To add a message box to the free-form surface, select  , the Other category,

from the parts palette; then add  , an *IMessageBox** part, to the free-form surface.

Making the connections

Now, connect the *exceptionOccurred* event of each event-to-action connection to the *showException* action of the message box.

Developing Applications

The connections appear as shown in Figure 89 on page 204. Save and close your visual part.

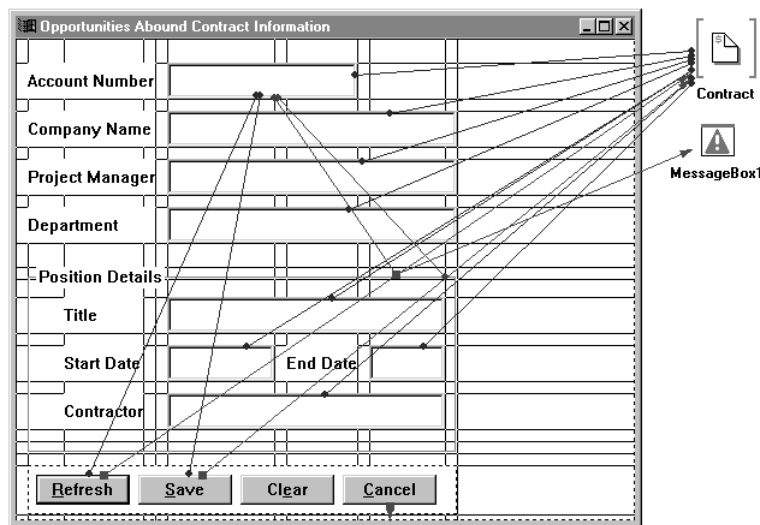


Figure 89. ContractorView with Exception Connections

Enabling a Window to Be Cleared of All Entry Values

You can enable the **Clear** push button to empty all entry fields of their contents using more than one method. One way would be to connect the *buttonClickEvent* feature of the **Clear** push button to the *removeAll* action of each entry field in the window. Another way would be to connect the *buttonClickEvent* feature of the **Clear** push button to a *resetFields* member function of the *OACContractView* composite part.

For this example, assume that the member function has been written. Make the connection as follows:

1. From the Composition Editor, use Alt+mouse button 2 to select the *buttonClickEvent* of the **Clear** push button.
The connection spider appears.
2. Use Alt+mouse button 2 to select an empty section of the free-form surface.
3. Select **More** from the pop-up menu.
4. Enter the member function declaration in the entry field as shown in Figure 90 on page 205.

Developing Applications

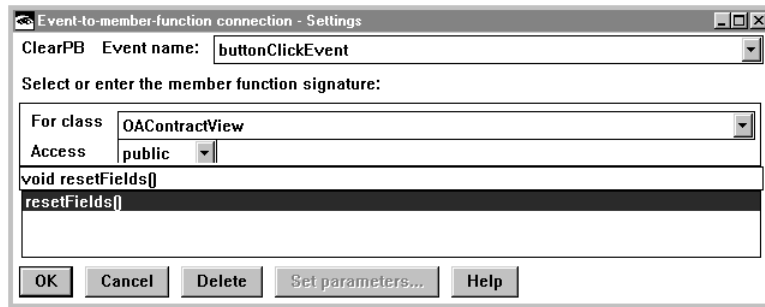


Figure 90. Free-Form Surface Connection Dialog

5. Select the **OK** push button.
6. Switch to the Class Editor.

Include the files containing the member function code. For this example, the file names are oafwclr.cpv and oafwclr.hpv.

- a. In the **User .hpv file** field, type the name of the header file, oafwclr.hpv
- b. In the **User .cpv file** field, type the name of the code file, oafwclr.cpv

For details on the attributes, actions, or events of a particular Visual Builder part, refer to the *Visual Builder Parts Reference*. For more information on how to perform these operations, see “Making the Connections” on page 171.

Developing Applications



Chapter 10. Displaying Objects in a List Box

The following steps show you how to modify the original To-Do List application to use objects instead of text strings. If you have not already done so, complete the original application, shown in Chapter 2, “Creating a Simple Visual Builder Application” on page 7, before continuing with this example.

When you created the simple Visual Builder application, you used an `IListBox*` part to display the items in a to-do list. The items you put in the `IListBox*` part were text strings you created by entering them in an entry field and clicking on a push button. In this case, the text strings displayed in the list box are just text strings.

Besides viewing a list of text strings, you can also use a list box to view a collection of nonvisual objects. List boxes that can display objects include the following:

- `ICollectionViewListBox*`
- `ICollectionViewComboBox*`

By connecting a collection of objects, such as an `IVSequence*` part, to one of these list boxes, you can view a collection of objects instead of simple text strings.

The procedure outlined in this section modifies the original To-Do List application so that the list box contains objects rather than text strings. In this case, text strings are again displayed in the list box, but this time they are used to show the names of the objects in the sequence. When you select an item in the list box after completing this procedure, you are really selecting an object in the sequence, not the text string that represents the object.

The modified To-Do List application differs from the original application in that it uses the following parts:

- An `ICollectionViewListBox*` part instead of an `IListBox*` part: The `ICollectionViewListBox*` part can contain objects instead of just text strings.
- An `IVBFactory*` part: This part creates new objects to put in the list box.
- An `IVSequence*` part: This part contains the sequence of objects that is reflected in the list box.
- A `ToDoItem` part: `ToDoItem` is a nonvisual part that is used as the type of objects that the list box and sequence will contain, and that the object factory will create.

The following prerequisites must be met to reflect a sequence of objects in a list box:

Developing Applications

- The class name of the type of objects in the list box and sequence, as well as the type of objects that the object factory creates, must be the same. This class name must also be the name of a part that inherits from `IStandardNotifier`. It does not have to inherit from `IStandardNotifier` directly, but `IStandardNotifier` must be somewhere above it in its class hierarchy.
- The objects in the sequence must override the `asString` member function inherited from the `IVBase` part. This is necessary so that at least one attribute of each object, such as the object's name, can be displayed in the list box.

In the procedure that you will follow for the To-Do List application, the `ToDoItem` nonvisual part has its own `asString` member function. This member function overrides the `asString` member function of `IVBase` by calling the `todoItemName` attribute's get member function, which returns the name of the to-do item, as follows:

```
IString ToDoItem :: asString () const
{
    return todoItemName();
}
```

Copying the ToDoList Part

First, you must copy the `ToDoList` part, as follows:

1. Select `ToDoList.vbb`. Visual Builder displays the name of the `ToDoList` part in the **Visual Parts** list box.
2. Select the `ToDoList` part.
3. Select **Part→Copy**. Visual Builder displays the Copy Part window.
4. Enter `ToDoLst2` in the **Target part name** field.
5. Enter `todo1st2.vbb` in the **Target file name** field.
6. Select the **Copy** push button. Visual Builder creates a copy of the `ToDoList` part, gives it the name `todo1st2`, and stores it in the `todo1st2.vbb` file.

Creating the ToDoItem Nonvisual Part

The next step is to create a nonvisual part called `ToDoItem`, as follows:

1. Open a new nonvisual part by doing the following:
 - a. In the Visual Builder window, select **Part→New**. Visual Builder displays the Part - New window.
 - b. Fill in the fields in this window as follows.

Class name `ToDoItem`

Developing Applications

Description	Type of objects in To-Do List
File name	todo1st2.vbb
Part type	Nonvisual part
Base class	IStandardNotifier

- c. Select the **Open** push button. Visual Builder displays the Part Interface Editor.
2. Create a new attribute called *todoItemName* by doing the following:
 - a. On the **Attribute** page, fill in the following fields:

Attribute name	todoItemName
Attribute type	IStrng

- b. Select the **Add with defaults** push button. Visual Builder adds the *todoItemName* attribute to the ToDoItem part.



You can add the *todoItemName* attribute to the preferred features list so that it will be readily available for making connections by doing the following:

1. Switch to the Part Interface Editor, if you are not already there.
 2. Select the Preferred page.
 3. Scroll down the list of attributes until you find the *todoItemName* attribute and select it.
 4. Select the **Add>>** push button. Visual Builder adds the *todoItemName* attribute to the list of preferred features so that it will appear in the pop-up connection menu.
3. Specify the .hvp and .cpv files for Visual Builder to use for the default feature code for the *todoItemName* attribute by doing the following:
 - a. Switch to the Class Editor if you followed the steps in the preceding hint.
 - b. Fill in the following fields:

User .hvp file	ToDoItem.hvp
User .cpv file	ToDoItem.cpv

4. Generate the default source and feature code for the ToDoItem part by doing the following:
 - a. Select **File→Save and generate→Part source**.

Visual Builder generates the code for the ToDoItem part and stores it in files named ToDoItem.hpp and ToDoItem.cpp. These files contain the appropriate include statements for the ToDoItem.hvp and ToDoItem.cpv files. Visual

Developing Applications

Builder also creates a header file for the declarations (ToDoItem.h) and a resource include file for the class (ToDoItem.rci).

- b. Select **File**→**Save and generate**→**Feature source**. Visual Builder displays the ToDoItem - Generate Feature Source Code window, as shown in Figure 91.

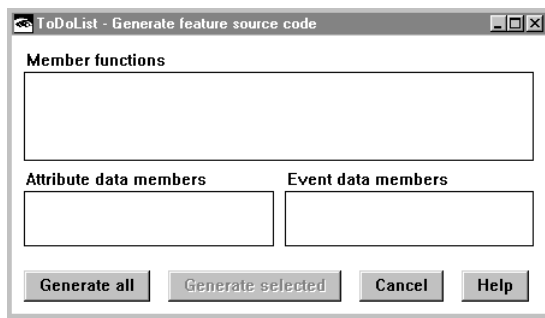


Figure 91. Generate Feature Source Code Window

- c. Select the **Generate all** push button. Visual Builder generates the default code for the *ToDoItemName* attribute and displays a message telling you where the files containing feature source code are stored. Click on **OK** to continue.
5. Override the *asString* member function in the *IVBase** part by doing the following:
 - a. Using your favorite text editor, edit the *ToDoItem.hpv* file and insert the following at the bottom of the public section:

```
    IString ToDoItem :: asString () const;
```
 - b. Save the file.
 - c. Edit the *ToDoItem.cpv* file and insert the following at the bottom of the file:

```
    IString ToDoItem :: asString () const
    {
        return ToDoItemName();
    }
```
 - d. Save the file and close the text editor.
 6. Close the open Visual Builder editor (Class Editor or Part Interface Editor).

The *ToDoItem* part is now completed.

Replacing and Modifying the List Box

You need a list box that can display objects instead of just text strings, so next you must replace the `IListBox*` part with an `ICollectionViewListBox*` part, and then modify the `ICollectionViewListBox*` part to specify the type of objects it can display.

Replacing `IListBox*` with `ICollectionViewListBox*`


1. Open the `ToDoLst2` part.
2. Delete the `IListBox*` part.

The `IListBox*` part only accepts strings. You want a part that accepts objects.


Visual Builder displays a message advising you that it will delete the connections as well as the list box if you continue. You want to delete the connections because you need different connections for this example.

3. Select the **OK** push button. Visual Builder deletes the list box and the connections.



4. Select , the Lists category, from the row of icons on the left-hand side of the parts palette.



5. Select , the `ICollectionViewListBox*` icon, from the row of icons that Visual Builder displays on the right-hand side of the parts palette.
6. Place the crosshairs below the lower-left corner of the second static text part and click mouse button 1.

A list box part that displays the items in a collection is placed beneath the second static text part.

Specifying the type of items the list box can contain

1. Move the mouse pointer to the list box, click mouse button 2, and select **Open settings**. Visual Builder displays the settings notebook for the list box.
2. Enter `ToDoItem` in the **Subpart name** field.
3. Enter `ToDoItem*` in the **Item type** field.
4. Select the **OK** push button.


The list box can now accept only `ToDoItem*` objects.

Placing and Modifying an Object Factory Part


You need an object factory part to create the objects to be displayed in the list box. In addition, you must specify the type of objects the IVBFactory* part can create.

Placing an object factory part on the free-form surface



1. Select , the Models category, from the row of icons on the left-hand side of the parts palette.



2. Select , the IVBFactory* icon, from the row of icons that Visual Builder displays on the right-hand side of the parts palette.
3. Place an object factory part on the free-form surface to the right of the To-Do List window adjacent to the entry field.
4. Change the text beneath the object factory icon to ItemFactory.

Specifying the type of objects the object factory can create

1. Move the mouse pointer to the object factory, click mouse button 2, and select **Change type**. Visual Builder displays a window in which you are to enter the type of objects that the object factory is to create.
2. Enter ToDoItem* in the entry field in this window.
3. Select the **OK** push button.

The object factory can now create only ToDoItem objects.

Placing and Modifying an IVSequence* Part

You also need to place an IVSequence* part to put the objects that the IVBFactory* part creates into a sequence; you must also specify the type of objects the sequence can contain.

Placing an IVSequence* part on the free-form surface

1. Select **Options→Add part**. Visual Builder displays the Add Part window.
2. Enter IVSequence* in the **Part class** field.
3. Enter ItemSequence in the **Name** field.
4. Select the **Add** push button.

5. Place an `IVSequence*` part on the free-form surface to the right of the To-Do List window adjacent to the list box.

Specifying the type of objects in the sequence

1. Move the mouse pointer to the sequence, click mouse button 2, and select **Open settings**. Visual Builder displays the settings notebook for the sequence.
2. Enter `ToDoItem` in the **Subpart name** field.
3. Enter `ToDoItem*` in the **Item type** field.
4. Select the **OK** push button.

The sequence can now accept only `ToDoItem` objects.

Making the New Connections

The following steps describe the connections that this example requires to add objects to and remove objects from the list box. To review the instructions for connecting features, see “Making the Connections” on page 171.

1. Connect the *buttonClickEvent* feature of the **Add** push button to the *new* action of the object factory.

This connection causes the object factory to create a new `ToDoItem` object whenever a user clicks the **Add** push button.

2. Connect the *toDoItemName* attribute of the object factory to the *text* attribute of the entry field.

This connection causes text in the entry field to be used as the name of `ToDoItem` objects that the object factory creates. Notice that the connection line is violet, the color of a parameter connection, instead of cyan, which is the color of an attribute-to-attribute connection. The connection line is violet because the *text* attribute of the entry field supplies a value for the *toDoItemName* attribute only when the object factory creates a new object, just as if it were satisfying a parameter of the *new* action.

3. Connect the *newEvent* attribute of the object factory to the *addAsLast* action of the sequence.

This connection causes each new `ToDoItem` object to be added as the last object in the sequence.

4. Connect the *this* attribute of the sequence to the *items* attribute of the list box.

This connection causes the list box to display all of the `ToDoItem` objects that the sequence contains.

Developing Applications

5. Connect the *buttonClickEvent* feature of the **Remove** push button to the *removeAtPosition* action of the sequence.

This connection causes the object at a specified position in the sequence to be removed. This connection is incomplete because the *removeAtPosition* action's *position* parameter must be satisfied.

6. Connect the *position* parameter of the previous connection to the *selectedCollectionPosition* attribute of the list box.

This connection provides the position of the selected item in the collection of objects, which is required by the previous connection in order to remove an object.

Generating the Source Code for Your Visual Part and main() Procedure

To generate the C++ source code for your visual part, select **File→Save and generate→Part source**. Visual Builder generates the following files in the working directory:

todo1st2.cpp	The C++ code for your todo1st2 part.
todo1st2.hpp	The C++ header file for your todo1st2 part.
todo1st2.h	The resource header file for your todo1st2.cpp file.
todo1st2.rci	The resource file for your todo1st2.rcx file, which will be created when you generate source code for your main() function.

Generating the source code for your main() procedure

To generate the source code for your main() procedure, select **File→Save and generate→main() for part**. Visual Builder generates the following files in the working directory:

todo1st2.app	The main function for your application. Note: If you start Visual Builder from a WorkFrame project, the name of this file is vbmain.cpp.
todo1st2.mak	The make file that you specify when you build your application. Note: You must select Options→Generate make files in the Visual Builder window to generate this file.
todo1st2.rcx	The main resource file. It includes a resource (.rci) file for each part that you generated part source for. In this example, there is only one, todo1st2.rci, because you only needed to generate part source for one part.

Building and Running the Modified Application

You should now be ready to build and run your modified To-Do List application, as follows:

Building the new To-Do List application

1. Open a command window.
2. Change to your Visual Builder working directory.
3. Enter the following command:

```
nmake todolst2.mak
```

This command produces the following files:

todolst2.exe The executable file for your application.

todolst2.map The application configuration map.

todolst2.o The object file for your application.

Note: If you start Visual Builder from a WorkFrame project, the name of this file is vbmain.obj.

todolst2.rc The preprocessed resource file for your application. Visual Builder concatenates and places all resource files into this file.

todolst2.obj The object file for your part. Visual Builder provides a separate object module for your part that is used when compiling this part with other parts.

todolst2.res The binary resource file that is bound to todolst2.exe.

Running the new To-Do List application

To run your application from the same command prompt from which you entered the nmake command, enter the following:

```
todolst2
```

Once your application is running, experiment with it to make sure it works as you designed it.

You can add a finishing touch to your application by creating a program object in OS/2 or a shortcut in Windows. Once you have done this, you can run your application by simply double-clicking on the program object or shortcut you just created.

Developing Applications

Chapter 11. Creating Resizable Windows

The purpose of this chapter is to show you how to use multicell canvases to build a window that can be resized dynamically.

All visual parts of the OASearch sample application except OAContractorView use multicell canvases as client areas. (OAContractorView uses multicell canvases as notebook page clients.) The example built in this chapter is based on Figure 92.

Opportunities Abound Contract Information			
Account Number			
Company Name			
Project Manager			
Department			
Position Details			
Title			
Start Date		End Date	
Contractor			
<input type="button" value="Refresh"/> <input type="button" value="Save"/> <input type="button" value="Clear"/> <input type="button" value="Cancel"/>			

Figure 92. Completed OAContractView Window with Multicell Canvas

For the finished panel, see the OAContractView part in oawin.vbb.

What Are the Benefits of Using Multicell Canvases?

Because multicell canvases are dynamically resizable, they are the best canvas choice for applications that will be used in a variety of display resolutions. Using multicell canvases also makes it easier to plan for translation of your interface into other languages.

Multicell canvases contain rows and columns of *cells* into which you can drop parts. Cells are referred to by row and column location: the cell at row 2, column 3 is called cell (2,3). Each cell is initially very small, 10 pixels wide and 10 pixels high.

Developing Applications

A grid of 5 rows and 5 columns appears when you first drop a multicell canvas into a client area.

Any extra space between the grid and the bounds of the client area appears to the right of the rightmost grid line and below the bottommost grid line. This extra space is called *the expansion area*. The expansion area can also change size at run time when the user resizes the window.

When you drop a visual part into the cell, the cell resizes to fit the *minimum size* of the dropped part. This minimum size is calculated for you based on limits that are set for the dropped part.

For example, suppose you want an entry field that holds 10 characters. You drop the part and use the settings notebook to set its limit to 10 characters. The minimum size of the entry field is then calculated based on holding 10 characters of the font defined for the entry field.

The minimum size of a cell also depends on the size of parts that span the cell but are not contained within it. For specific examples of how layout influences minimum size, refer to the sample visual parts contained in `oawin.vbb`.

Rows and columns are either *fixed* or *expandable*, depending on how their size is calculated.

- The size of a fixed row or column is based on the minimum size of the largest cell in that row or column. If no part occupies a cell in that row or column, the row or column takes the default minimum size.
- Expandable rows or columns can grow from their minimum size (the same as if their size were fixed) to include the expansion area. If two or more expandable rows or columns exist, the expansion area is parceled out among them. Examples of the use of expandable columns appear in `oawin.vbb`.

When first created, all rows and columns have a fixed size. To make them expandable, edit the settings for the multicell canvas. For more information, see “Changing the Settings for a Multicell Canvas” on page 227.

Working with multicell canvases does not require the Visual Builder alignment tools. You alternate between the free-form surface and the settings notebook for the multicell canvas.


Adding a Multicell Canvas


This example is based on `OAContractView`, found in `oawin.vbb`. The finished part is shown in Figure 92 on page 217.

1. Because you are constructing a new user interface, begin by creating a visual part. Name it `contractView`.

When you create a visual part, the Composition Editor opens and an `IFrameWindow*` part is automatically added for you. By default, the client area of the `IFrameWindow*` part appears as a canvas.

2. Delete the default canvas client.

3. Select , the Composers category, from the left side of the parts palette.

4. Select , the `IMultiCellCanvas*` part, from the right side of the parts palette and drop it on the `IFrameWindow*` part.

Your visual part now looks like Figure 93.

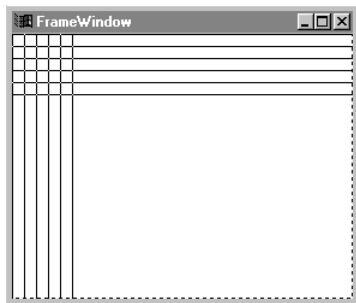




Figure 93. New `contractView` Part with Empty Multicell Canvas

Adding Parts to the Multicell Canvas

When you first drop a multicell canvas, the cell grid appears at its default size in the upper-left corner of the client area. You can now drop parts into the canvas. For the `contractView` example, do the following:

1. Select , the Data Entry category, from the left side of the parts palette.

Developing Applications

2. Select , the IStaticText* part, from the right side of the parts palette. When you move the mouse pointer over the free-form surface, the pointer changes to crosshairs.
3. Drop the part at cell (2,2) of the multicell canvas.
- Row 1 was left as a spacer between the top of the multicell canvas and the window border. When you drop the IStaticText* part, cell (2,2) expands to contain the dropped part. The multicell canvas now looks like Figure 94.

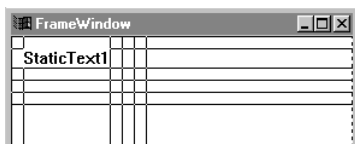


Figure 94. Multicell Canvas with IStaticText* Part at Cell (2,2)

4. Drop another IStaticText* part at cell (4,2), leaving row 3 as a spacer. The multicell canvas now looks like Figure 95.

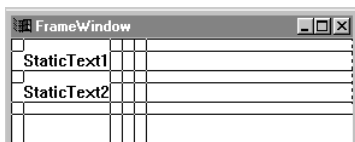


Figure 95. Multicell Canvas with IStaticText* Parts, Showing Spacer Rows

5. Drop two more IStaticText* parts into the expansion area below the StaticText2 part.
- Each time you drop a part into the expansion area, a new row is created for you.
6. Drop three more IStaticText* parts in column 3 as shown in Figure 96 on page 221.

Developing Applications

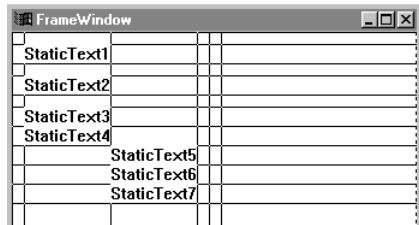




Figure 96. Multicell Canvas Showing Parts Dropped into an Expansion Row

No spacer rows exist yet between these new rows. We will add these in “Adding Rows or Columns Using the Contextual Menu” on page 222. Now add some IEntryField* parts.

7. Select , the Data Entry category, from the left side of the parts palette.
8. Select , the IEntryField* part, from the right side of the parts palette.
9. Drop IEntryField* parts in the expansion area to the right of the IStaticText* parts, starting at row 2. A new column 6 is created, as shown in Figure 97.

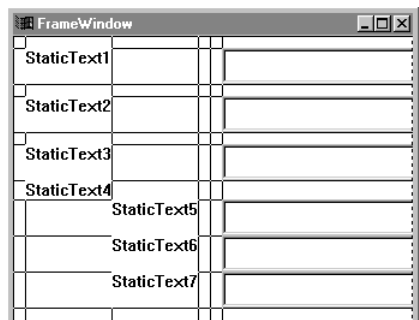


Figure 97. Multicell Canvas Showing Parts Dropped into an Expansion Column

Changing the Multicell Canvas Grid

You can add or delete rows or columns from a multicell canvas as follows:

- To add rows or columns from the expansion area, you can just drop parts into it, as discussed in “Adding Parts to the Multicell Canvas” on page 219.
- To add several rows or columns at a time, you might find it easiest to change the grid using the multicell canvas’s settings notebook. This method is discussed in “Adding Rows or Columns Using the Settings Notebook” on page 229.
- With the multicell canvas selected, you can press mouse button 2 and use the contextual menu.

Adding Rows or Columns Using the Contextual Menu

You can add rows or columns anywhere on the multicell canvas as follows:

1. Select a cell next to where you want to add the row or column.

Note: No selection handles are shown on the cell if it is empty. Also, you can only open the contextual menu for an empty cell.

2. Press mouse button 2 to open the contextual menu.
3. To add a row, select **Rows**. To add a column, select **Columns**.

A cascade menu appears.

4. If you want to insert the row above the cell you selected, select **Add row before** from the cascade menu.

If you want to insert the row below the cell you selected, select **Add row after** from the cascade menu.

For columns, **Add column before** inserts a column to the left of the selected cell. **Add column after** inserts a column to the right of the selected cell.

For practice, add spacer rows to the contractView example as follows:

1. Select the empty cell to the left of **StaticText3**. Add a row after it.
2. Add one row each after **StaticText4** through **StaticText7**.

At this point, the contractView window looks like Figure 98 on page 223.

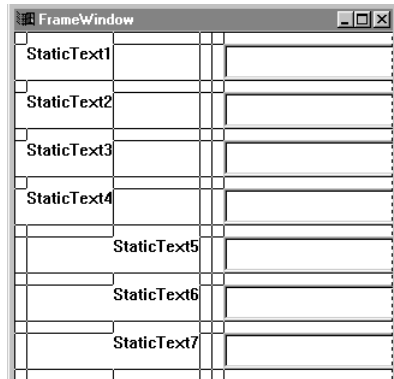


Figure 98. Multicell Canvas with Inserted Rows

Deleting Rows or Columns Using the Contextual Menu

You can delete rows or columns from anywhere on the multicell canvas as follows:

1. Select an empty cell in the row or column you want to delete.
2. Press mouse button 2 to open the contextual menu.
3. To delete the row, select **Rows**. Then select **Delete row**.

To delete the column, select **Columns**. Then select **Delete column**.

Extending a Part to Span More than One Cell

Parts can span more than one column or row. This is necessary when placing a part entirely within one cell would distort the multicell canvas. Consider what happens when you add three push buttons individually to the sample multicell canvas, as shown in Figure 99 on page 224.

Developing Applications

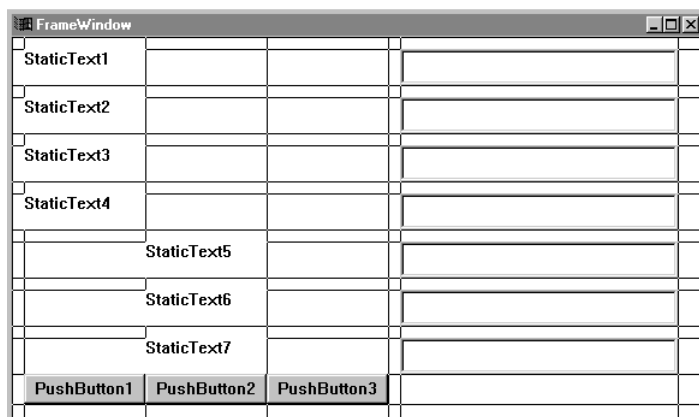


Figure 99. Distorted Multicell Canvas with Individual Push Buttons

Clearly, this was not intended. Instead, build a deck of push buttons using an `ISetCanvas*` part as a base, as follows:

1. Drop an `ISetCanvas*` part in the bottom expansion area of column 2.
2. Extend the span of the `ISetCanvas*` as follows:
 - a. Select the `ISetCanvas*` part.
 - b. Holding the Alt key with one hand, drag either right-hand part handle over to the column that contains the `IEntryField*` parts.
 - c. Release the mouse. The sample window now looks like Figure 100.

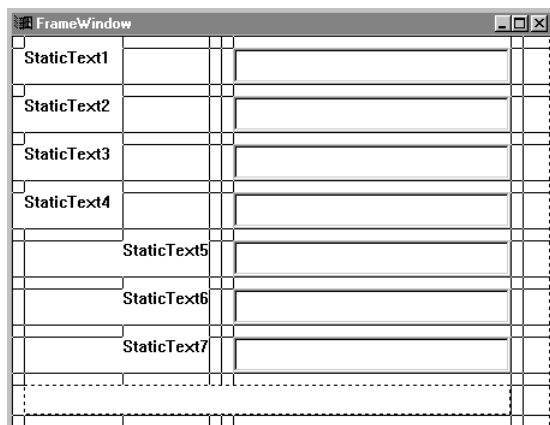


Figure 100. Multicell Canvas with `ISetCanvas*` Spanning Several Cells

3. Add three `IPushButton*` parts to the `ISetCanvas*` part.

Developing Applications

The `ISetCanvas*` and multicell canvas parts expand to contain the newly dropped `IPushButton*` parts.



To change the way minimum size is calculated for the `IPushButton*` parts, edit Pack Type settings for the `ISetCanvas*` part.

4. Edit the `IStaticText*` parts so that they appear as shown in Figure 92 on page 217. Extend each column 2 `IStaticText*` part into column 3.

Adding a Group Box

To organize groups of related visual parts on the multicell canvas, add a group box. You can drop the `IGroupBox*` part before or after you drop the visual parts contained within it.

- If you prefer to drop the `IGroupBox*` part first, consider carefully how many rows and columns you need within the group box. Add any extra rows and columns before you drop the `IGroupBox*` part. It can be difficult to add others in the correct locations later.
- If you prefer to drop the contained parts first, you must edit the depth order after you have dropped the `IGroupBox*` part. This is necessary because parts are added to the window's depth order in the order in which they are dropped. To work properly, the `IGroupBox*` part must appear in the depth order before the parts contained within it. For more information about depth order, see “Changing Depth Order within a Composite Part” on page 146.

In the `contractView` sample, the column 3 `IStaticText*` parts and their entry fields belong to the **Position Details** group shown in Figure 92 on page 217.

The group box will extend from cell (10,2) to cell (18,8).

Adding the extra rows and columns to hold the group box

Extra cells are necessary to hold the group box and provide space between the group box and the parts within it.

1. Add 2 rows above **Title**.
2. Add 2 rows after **Contractor**.
3. Add 2 columns after the corresponding `IEntryField*` parts.

The sample window now looks like Figure 101 on page 226.

Developing Applications

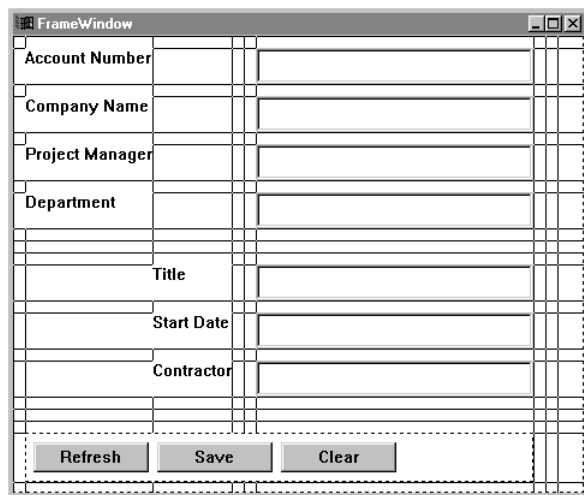




Figure 101. Preparing the Multicell Canvas to Hold an IGroupBox* Part

Dropping and extending the group box

1. Select  , the Data Entry category, from the left side of the parts palette.
2. Select  , the IGroupBox* part, from the right side of the parts palette.
3. Drop the IGroupBox* part into cell (10,2).

This temporarily distorts the multicell canvas. Resize the IFrameWindow* part to see the extra columns on the right.

4. Select the IGroupBox* part. Holding the Alt key with one hand, drag the lower-right part handle to cell (18,8) as shown in Figure 102 on page 227.

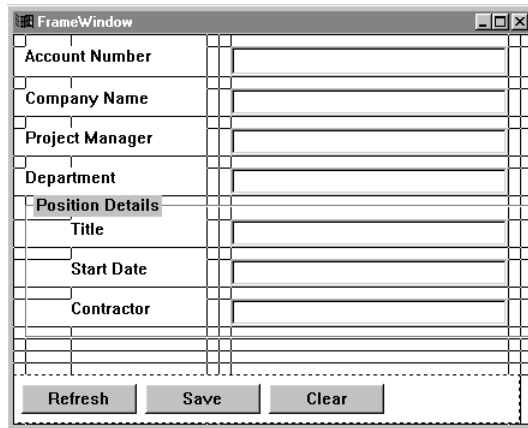


Figure 102. Multicell Canvas Showing Dragged IGroupBox* Part

If necessary, press Alt+Backspace to undo and retry. Edit the IGroupBox* text to read *Position Details*. Remember to move the IGroupBox* part up in the depth order.

Changing the Settings for a Multicell Canvas

You can use the **General** page of the multicell canvas settings notebook to do the following:

- Move a dropped part
- Change the default minimum size for rows or columns
- Add rows or columns
- Delete rows or columns
- Make fixed rows and columns expandable

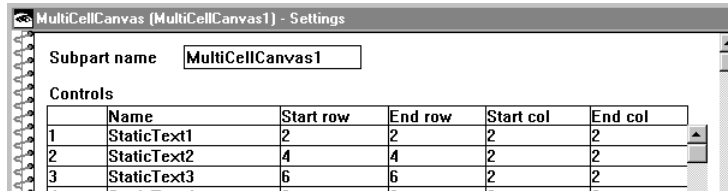
This page consists of a series of directly editable tables.

Moving Dropped Parts

From the free-form surface, you can move dropped parts by selecting and dragging them to their new locations. You can also move dropped parts between cells in a multicell canvas using the settings notebook as follows:

1. Open settings for the multicell canvas.
2. At the top of the **General** page is a scrollable list of all parts contained in the multicell canvas as shown in Figure 103 on page 228.

Developing Applications



	Name	Start row	End row	Start col	End col
1	StaticText1	2	2	2	2
2	StaticText2	4	4	2	2
3	StaticText3	6	6	2	2

Figure 103. Multicell Canvas General Settings, Controls Table

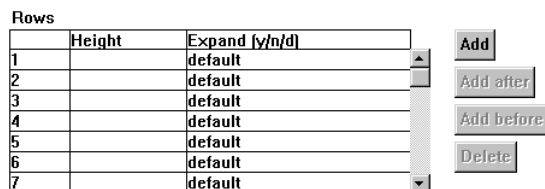
3. Select the table entry you want to change and edit the highlighted value. Row and column numbering starts from (1,1) in the upper-left corner of the multicell canvas.
4. Select the **OK** push button to close settings.

Use this table for reference whenever you edit row and column settings.

Changing Default Minimum Size for Rows or Columns

You can change the minimum size of a row or column when it is empty. The default minimum size is 10 pixels wide (columns) and 10 pixels high (rows), but you can change the default minimum size for individual rows and columns as follows:

1. Open settings for the multicell canvas.
2. The second table on the **General** settings page is called **Rows**, as shown in Figure 104.



	Height	Expand [y/n/d]
1		default
2		default
3		default
4		default
5		default
6		default
7		default

Figure 104. Multicell Canvas General Settings, Rows Table

3. The default height appears as default, To change the minimum height for row 1, select the Row 1 table entry under Height. Change this to the new default value in pixels. For example, the minimum height of row 1 in most views of the OASearch sample application is 20.
4. Scroll down the **General** page a little more to see the third table on the page, called **Columns**. You can edit values in this table in the same way you edited the **Rows** table.
5. Select the **OK** push button to close settings.

Adding Rows or Columns Using the Settings Notebook

To save time when adding more than one row or column at a time, use the settings notebook. The methods for adding rows and columns are similar, as follows:

1. Open settings for the multicell canvas.
2. If necessary, scroll down to the **Rows** table. For a complex layout, consider resizing the window so that the **Controls** table is also displayed.
3. Select the number of a row next to the insertion point. Select the **Add** push button.
4. Select either **Add before** or **Add after**.
 - **Add before** inserts the new row above the selected row.
 - **Add after** inserts the new row below the selected row.
5. Select the **OK** push button to close settings.

Deleting Rows or Columns Using the Settings Notebook

To save time when deleting more than one row or column at a time, use the settings notebook. The methods for deleting rows and columns are similar, as follows:

1. Open settings for the multicell canvas.
2. If necessary, scroll down to the **Rows** table. For a complex layout, consider resizing the window so that the **Controls** table is also displayed.
3. Select the number of the row you want deleted. Select the **Delete** push button.
4. Select the **OK** push button to close settings.

Making Rows or Columns Expandable

To allocate extra client space, make at least one row and one column expandable. In most of the OASearch windows, the row immediately above the deck of push buttons is expandable. In the OAContractView part, the rightmost column is expandable. For a more complicated example involving several expandable rows and columns, see the OAMain part in oawin.vbb.

The methods for making rows and columns expandable are similar, as follows:

1. Open settings for the multicell canvas.
2. If necessary, scroll down to the **Rows** table. For a complex layout, consider resizing the window so that the **Controls** table is also displayed.
3. The default expansion value appears as default. Select the appropriate entry from the **Rows** table under Expand. For the contractView example, select the row 19 entry.
4. The value *d* is highlighted. Change it to *y* and select the **Apply** push button.

Developing Applications

5. Move the settings notebook window over so you can see the result of your work, as shown in Figure 105 on page 230.

The screenshot shows a window titled 'FrameWindow' containing a multicell canvas. The canvas is a table with 4 columns and 19 rows. The first 18 rows are grouped into sections: 'Account Number' (row 1), 'Company Name' (row 2), 'Project Manager' (row 3), 'Department' (row 4), 'Position Details' (row 5), 'Title' (row 6), 'Start Date' (row 7), and 'Contractor' (row 8). The 19th row is an expandable row, indicated by a dashed border. At the bottom of the window, there are three buttons: 'Refresh', 'Save', and 'Clear'.

Figure 105. Multicell Canvas with Expandable Row 19

The expansion area is now hidden. All extra vertical space becomes part of row 19.

6. When you are finished adjusting row and column expansion settings, select the **OK** push button.

To make this row fixed later, enter *n* in the **Expand** column.

Chapter 12. Constructing Containers and Notebooks

The purpose of this chapter is to show how easy Visual Builder makes it to use IBM Open Class Library container and notebook controls.

Container controls hold objects; for example, a program folder holds program objects. You can display containers in different *views* including the following:

- Icon view. Container objects appear inside the container as icons.
- Details view. Attributes of container objects appear in tabular columns. To use this view, you must define the container columns as well as the container itself.
- Tree view, for container objects that are related hierarchically.

Notebook controls present related information on tabbed pages that the user can display sequentially or randomly. For an example of a notebook control, open any Visual Builder settings editor.

Adding Container Parts

In this section, you set up a container in the OASkillView part to hold the OASkill* objects returned as a result of a user's skill query. The completed window looks like Figure 106.

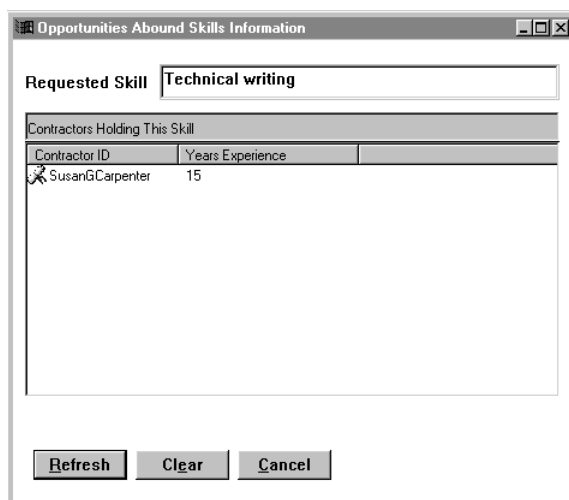


Figure 106. OASkillView Window

Developing Applications

This example is based on the OASkillView* part, found in oawin.vbb. Before constructing this new visual part, make sure that oanonvis.vbb is loaded into Visual Builder.


Adding a container part to the OASkillView* part

To add a container part to the OASkillView* part, do the following:


1. Create a new visual part and name it OASkillView.

A frame window and default canvas appear.



2. Select , the Lists category, from the left side of the parts palette.



3. Select , the IVBContainerControl* part, from the right side of the parts palette and drop the part onto the default canvas.
4. Change the container's name to *SkillCnr*.
5. Resize the container part as needed.

Setting Up the Container

You set some container values in the container part and some in the container column part. Set the following values in the container part:

- Title and title format
- View type
- Type of contained objects

Set the following values in the container column parts:

- Column heading and width
- Attribute to appear in the column
- Vertical and horizontal separators
- Whether contents can be changed by the user

For information about container columns, see “Adding Container Columns” on page 234.

Setting the container titles

To set the container titles, do the following:

1. Open the settings notebook for SkillCnr.

Developing Applications

2. Specify the general settings. In the **Title attributes** group, type Contractors Holding This Skill into the **Title** field.
3. Select both **Show title** and **Show title separator**.
4. Select the **Left** radio button for left title alignment.

Specifying the container type and layout

To specify the container type and layout, do the following:

1. Select **showDetailsView** from the **View type** list box.
2. Scroll down on the **General** page. In the **Container item attributes** group, type OASkill* into the **Item type** field.
3. Select contractorID from the **Text** drop-down combination box. The **Text** field specifies which OASkill attribute would appear as text in the container's icon view.
4. The **Icon** field specifies the icon to be used for each item in the container. Type the following into the **Icon** field:

```
#IDynamicLinkLibrary("cppwv33r").loadIcon(803)
```

cppwv33r.dll is the resource DLL containing the icons for this application, and 803 is the resource ID for the skill icon that would appear in the container's icon view.
5. Select the **Refresh the container after changes** check box.
6. On the **Styles** page, make sure that the **pmCompatible** style is set the way you want it, as follows:

- If you select the **On** radio button, you have access only to functions that are common to OS/2 and Windows. The compiled container looks like a Common User Access (CUA) container, even in Windows.
- If you select the **Off** radio button, you have access to all functions in the native container. This function varies between OS/2 and Windows. In OS/2, the compiled container looks like a CUA container. In Windows, the compiled container looks like a native Windows container.

From now on, the term *CUA container* refers to an OS/2-like container, regardless of platform. The term *Windows container* refers to the native Windows control.


7. Select the **OK** push button to save and close the settings notebook.


Developing Applications

Adding Container Columns

Once you have added a container, add container columns. This is a good idea even if you intend for the container to be used mainly as an icon view.

To add container columns, do the following:

1. Select  , the Lists category, from the parts palette.
2. Select the **Sticky** check box.

3. Select  , the IContainerColumn* part, from the parts palette.
4. Drop three IContainerColumn* parts on the SkillCnr part.

Setting up an icon container column

To set up icon container columns, do the following:

1. Open up the settings notebook for the first container column.
2. On the **General** settings page, type Icon in the **Heading text** field.
Note: In Windows containers, this column title does not appear.
3. Specify the column's width (in pixels) in the **Width** field. For this example, enter 0.
4. Specify the OASkill attribute to appear in this column. Select the **Use the icon attribute set in the container** radio button because you want to display the same information in this container column that the container displays on the icon view.
5. Set the container so that its elements cannot be changed by the user. Select the following from the **Styles** settings page:
 - For the **readOnlyHeading** style, select the **On** radio button.
 - For the **readOnly** data style, select the **On** radio button.
6. Set a vertical separator for container. For the **verticalSeparator** data style, select the **On** radio button.
Note: In Windows containers, this setting is ignored.
7. Select the **OK** push button to close the settings notebook.

Developing Applications

Setting up string container columns

To set up string container columns, do the following:

1. Open up the settings notebook for the second container column.
2. On the **General** settings page, type Contractor ID in the **Heading text** field.
3. Specify the column's width (in pixels) in the **Width** field. For this example, enter 200.



If you specify a container column width of 0, the column is sized according to the width of the longest column element.

4. Specify the OASkill attribute to appear in this column. Select the **Use the text attribute set in the container** radio button because you want to display the same information in this container column that the container displays as text on the icon view.
5. Set the container so that its elements cannot be changed by the user. Select the following from the **Styles** settings page:
 - For the **readOnlyHeading** style, select the **On** radio button.
 - For the **readOnly** data style, select the **On** radio button.
6. Set a vertical separator for container. For the **verticalSeparator** data style, select the **On** radio button.
7. Select the **OK** push button to close the settings notebook.

Repeat this procedure for the third container column, adjusting the width of the column to fit. The second column is supposed to display the number of years of experience each contractor has for a certain skill, so use the following settings:

- Select the **Use an attribute from the part** radio button.
- Choose *yearsExp* to populate the container column.

Do not add a vertical separator.

Developing Applications

Filling the Container

The easiest way to fill the container is to use a collection part. The OASkillView part uses an IVSequence* part named SkillList. Basically, you manage container objects through the collection and use a single connection between it and the container to load the contents of the collection into the container.

Adding the nonvisual parts

The OASearch application uses static parts that are actually found in the OAMain part. All nonvisual parts in this view except SkillList are variables that will be connected later to the actual parts.

To add the nonvisual parts, do the following:

1. Add an OASkill* part as a variable. Name it Skill.

For help on how to do this, see “Adding a Variable to a Composite Part” on page 160.

2. Add an OASkillBase* part as a variable. Name it SkillBase.
3. Add an IVSequence* part. Name it SkillList.
4. Add an IMessageBox* part to handle exceptions.

Populating the collection

The SkillBase part contains the action (*getSkills*) needed to populate the skill list. The *getSkills* action takes two parameters, a skill description and a sequence.

Make the following connections to initialize the collection and fill the container:

From	To
OASkillView,#ready	SkillBase,#getSkills
Skill,#skillName	The <i>aSkillName</i> parameter of the OASkillView,#ready--> SkillBase#getSkills connection. Visual Builder draws the parameter connection in the opposite direction.
SkillList,#this	The <i>aList</i> parameter of the OASkillView,#ready--> SkillBase#getSkills connection. Visual Builder draws the parameter connection in the opposite direction.
SkillList,#this	SkillCnr,#items

Developing Applications

OASkillView,#ready

SkillCnr,#refresh

Make the following connections to enable the push buttons:

From

To

Skill,#skillName

EntryField1,#text

EntryField1,#textLength

RefreshPB,#enabled

RefreshPB,#buttonClickEvent

SkillBase,#getSkills

EntryField1,#text

The *aSkillName* parameter of the RefreshPB,#buttonClickEvent--> SkillBase,#getSkills connection. Visual Builder draws the parameter connection in the opposite direction.

SkillList,#this

The *aList* parameter of the RefreshPB,#buttonClickEvent--> SkillBase,#getSkills connection. Visual Builder draws the parameter connection in the opposite direction.

To handle exceptions, connect the *exceptionOccurred* feature of the *getSkills* connections to the *showException* feature of the message box part.

Clearing the Container

To clear the container, empty the collection part that supports it. For the OASkillView part, make the following connection to enable the **Clear** push button:

From

To

ClearPB,#buttonClickEvent

SkillList,#removeAll

Adding Notebook Parts

The purpose of this section is to guide you through using a notebook part to create the OACContractorView part. The completed window looks like Figure 107 on page 238.

Developing Applications

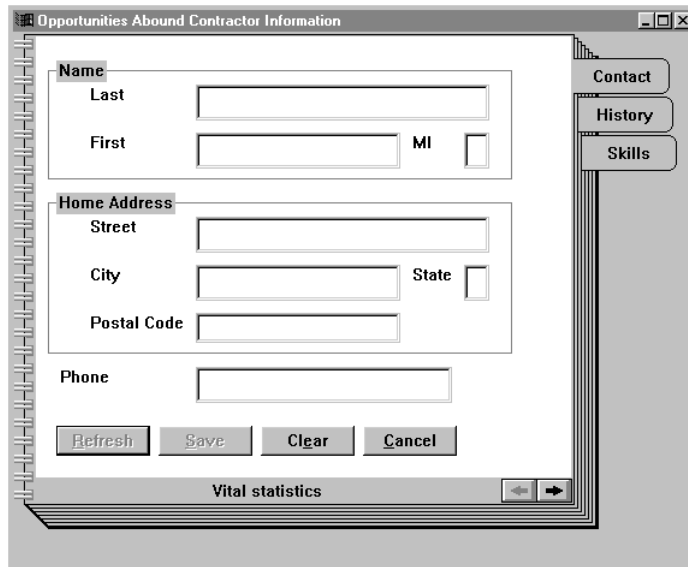


Figure 107. CUA Version of the Opportunities Abound Contractor Information Window

If you prefer a native notebook on Windows, the completed window looks like Figure 108.

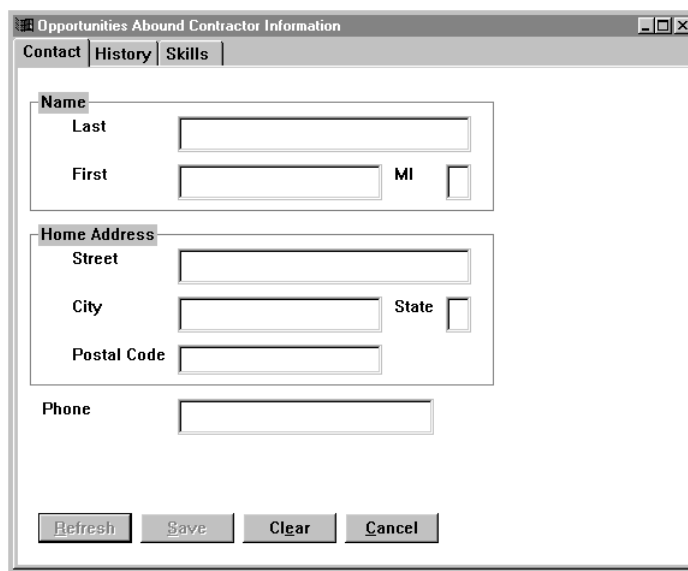


Figure 108. Windows Version of the Opportunities Abound Contractor Information Window

Developing Applications

The steps in creating a notebook are as follows:

- Add the notebook and set it up.
- Add notebook pages and set them up.

Adding the Notebook

Before constructing this new visual part, make sure that `oanonvis.vbb` is loaded into Visual Builder.


To add the notebook, do the following:

1. Create a new visual part. Call it `OACContractorView`.

A frame window and default canvas appear.

2. Delete the default canvas part from the frame window and resize the window as necessary.

3. Select , the Composers category, from the left side of the parts palette.


4. Select , the `INotebook*` part, from the right side of the parts palette and drop the part onto the frame window. An `IVBNotebookPage*` part and an `ICanvas*` part are also dropped for you automatically.

Specifying the notebook layout and appearance

Note: In Windows, the **pmCompatible** style setting determines whether most of these settings are used or ignored. Be sure to read through this entire section.

You change the notebook's appearance using its settings notebook, as follows:

1. Open the settings notebook for the notebook part.
2. On the **General** settings page, select the icon that represents the orientation that

you want for the notebook. For this example, select  from the **Layout** group.

3. From the **Binding** group, select the **Spiral** radio button.
4. From the **Tab shape** group, select the **Round** radio button.
5. From the **Justification** group, select the two **Center** radio buttons.
6. Select the **OK** push button to close the settings notebook.

Developing Applications

7. On the **Styles** page, make sure that the **pmCompatible** style is set the way you want it, as follows:
 - If you select the **On** radio button, you have access only to functions that are common to OS/2 and Windows. The compiled notebook looks like a Common User Access (CUA) notebook, even in Windows.
 - If you select the **Off** radio button, you have access to all functions in the native notebook. This function varies between OS/2 and Windows. In OS/2, the compiled notebook looks like a CUA notebook. In Windows, the compiled notebook looks like a native Windows notebook.

From now on, the term *CUA notebook* refers to an OS/2-like notebook, regardless of platform. The term *Windows notebook* refers to the native Windows control.

Adding Notebook Pages

The first notebook page was added for you when you dropped the INotebook* part. You can add notebook pages from either the VBNotebookPage part or the INotebook* part.

To add notebook pages from the INotebook* part, do the following:

1. In the Composition Editor, open a contextual menu for the part and select **Add page**.
2. From the cascade menu that appears, select either **Before top page** or **After top page**.

For this example, add one page after the initial page.

Setting up the notebook page and tab

To set up the notebook page and tab, do the following:

1. Open the settings notebook for the first notebook page.



The easiest way to open settings for a notebook page is through the Parts List window. Select the notebook; select **View parts list** from the notebook's contextual menu. Double-click the icon that represents the notebook page you want to set.

2. On the **General** page, type ContactPage into the **Subpart name** field.
3. Type Contact into the **Tab text** field. This text is what appears on the notebook tab.
4. Type Vital statistics into the **Status text** field. This text is what appears in the status area at the bottom of the notebook page.

Developing Applications

5. On the **Styles** page, select the **On** radio button for the following styles:

- The **autoPageSize** style, to enable automatic sizing of the notebook page
- The **statusTextOn** style, to enable display of the status text that you entered in the previous step
- The **majorTab** style, to give the notebook page a major tab

Adding parts to a notebook page

Each notebook page initially contains an ICanvas* part. If you want to use a different part, delete the ICanvas* part and select another part from the Composers category, such as IMultiCellCanvas*.

Note: A notebook page allows only one subpart, which should be a part from the Composers category. You can, of course, add other subparts to the Composers part.

The ContactPage subpart contains the primitive parts shown in Figure 109.

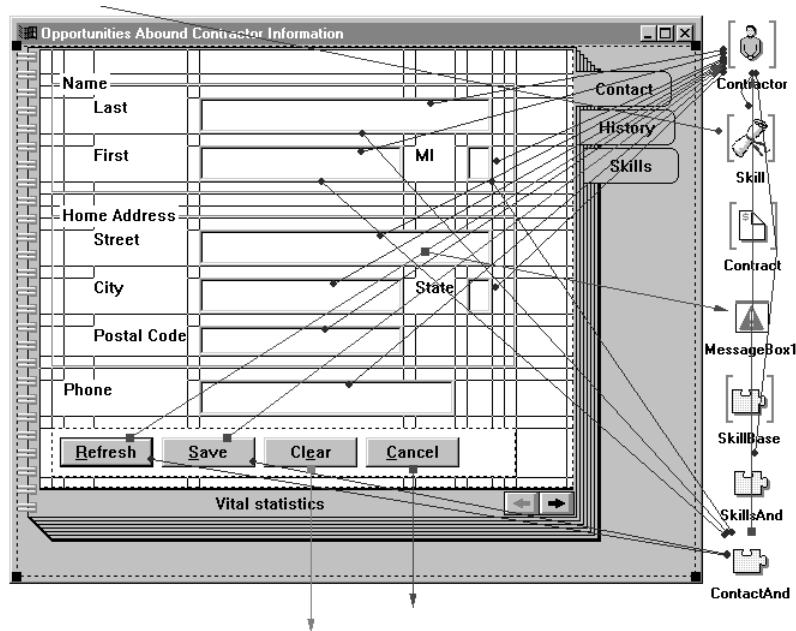


Figure 109. OACContractorView Contact Page

To see the other notebook pages, open the OACContractorView part in oawin.vbb.

Developing Applications

Chapter 13. Adding Menus to Your Application

The purpose of this section is to guide you through adding menus to the welcome window of the sample recruitment application. The completed menu parts and connections are shown in Figure 110.

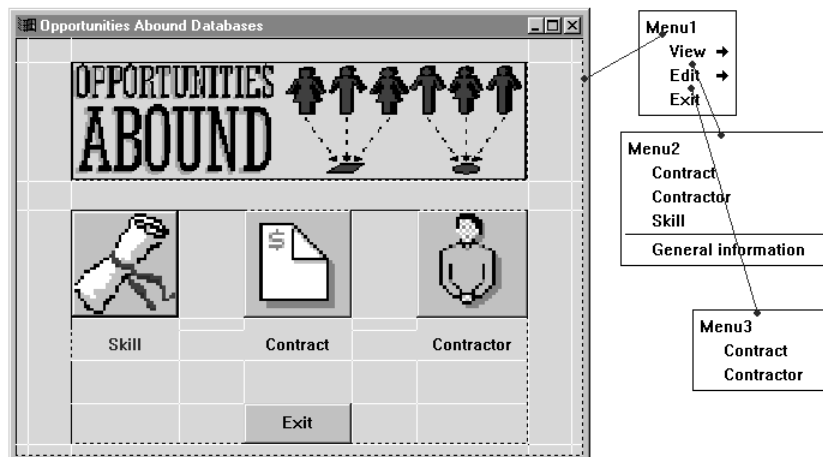


Figure 110. OASearch Welcome Window with Menus

Types of Menus and Menu Items

In Visual Builder, you use the same menu part to build several different menu types, as follows:

Menu bars

Menu type that is attached to a window. It appears horizontally under the window's title bar. If you use a menu bar, be sure to leave room for it between the window's title bar and the control that is closest to the top of the client area. Otherwise, the menu bar might overlay that control.

Pop-up menus

Menu type that is attached to a control, such as an entry field, within the window. It appears vertically when the user selects the control and presses mouse button 2.

Developing Applications


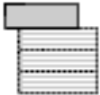
Cascaded menus

Menu type that is attached to a cascaded menu item, as follows:

- If the cascaded menu item is part of a menu bar, the cascaded menu appears below the menu item as a pull-down menu.
- If the cascaded menu item is part of a pop-up menu, the cascaded menu appears beside the menu item.

Adding a Menu Bar

Begin by opening the visual part that contains the OASearch welcome window, the OAMain part.

1. Select , the Frame Extensions category, from the left side of the parts palette.
2. Select , the IMenu* part, from the right side of the parts palette and drop it on the free-form surface next to the window.
3. Connect the *menu* attribute of the IFrameWindow*-based part to the *this* attribute of the menu part. Your part should look similar to what is shown in Figure 111.

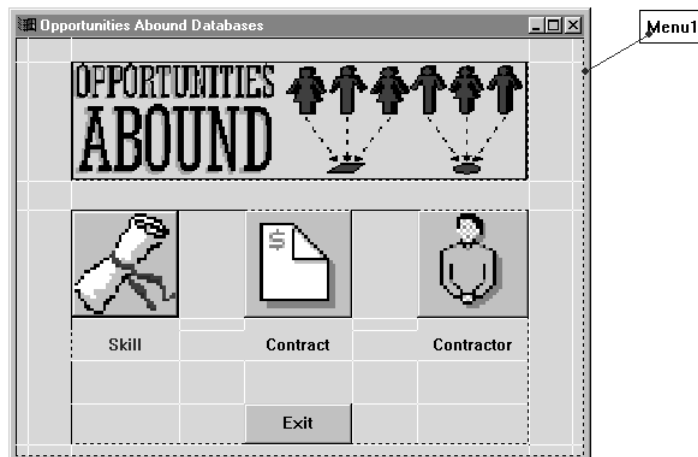


Figure 111. OAMain with Menu Part

4. To add two of the menu choices to the menu bar, add two menu parts on top of the menu part for the menu bar.

Developing Applications

As you add each menu part, a cascade button part is added to the menu bar and a connection is made between the cascade button part and the menu part you added. This connection causes the menu part to be displayed when the user selects the cascade button part. See Figure 112 on page 245.

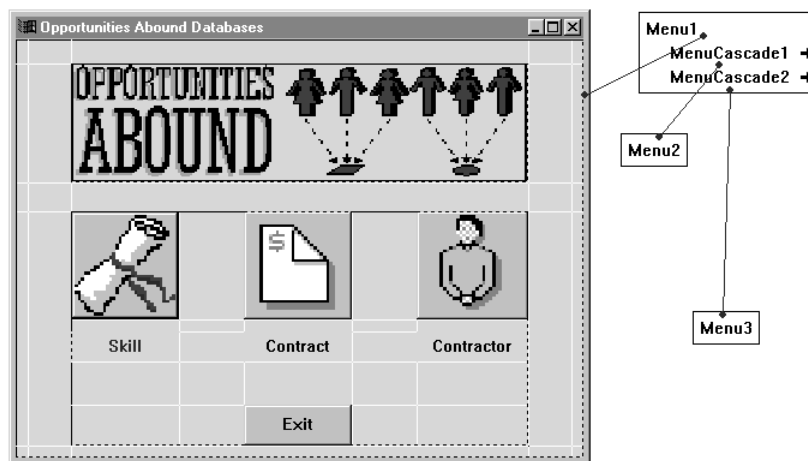


Figure 112. OAMain with Cascade Button Parts

5. Edit the text of the menu choices as shown in Figure 110 on page 243. To do this, select each one and press Alt+mouse button 1.

Connecting the Menu Bar to the Window

To make the menu a menu bar, connect the *this* attribute of the IMenu* part to the *menu* attribute of the window part. Although the menu continues to appear vertically on the free-form surface, this connection defines the menu part as a menu bar.


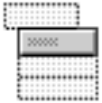


Because the menu bar is shown outside the frame window, be sure to leave enough space for it below the window title.

Making this same connection to a part other than a window part, such as a list part, makes the menu part a pop-up menu instead of a menu bar. See “Types of Menus and Menu Items” on page 243 for more information.

Adding Menu Choices



Once the menu structure is complete, you need to add menu choices that do something other than open other menus.

1. Select  , the Frame Extensions category, from the left side of the parts palette.
2. Select  , the IMenuItem* part, from the right side of the parts palette.
3. The position at which you click the mouse is where the part is added within the menu. Drop menu items as follows:
 - One part as the last item in the Menu1 part
 - Four items in the Menu2 part
 - Two items in the Menu3 part

You can change the position of a menu choice part within the menu part by using mouse button 2 to drag each item to a new position. Position these parts and change their text as shown in Figure 110 on page 243.

Adding Menu Separators

Once you have added and edited the OAMain menu items, add a separator bar to Menu2 between Skill and General information as follows:

1. Select  , the Frame Extensions category, from the left side of the parts palette.
2. Select  , the IMenuSeparator* part, from the right side of the parts palette.
3. Drag the IMenuSeparator* part and drop it between Skill and General Information.

Connecting Menu Choices to Actions

Once you have added menu choices, you can connect them to actions in this or other parts. As an example, connect the **Exit** menu bar item so that when the item is selected, the main window closes, as follows:

From part, feature	To part, feature
MenuItem2 ,# <i>commandEvent</i>	FrameWindow,# <i>close</i>

Command events occur when a user selects a menu item, push button, or accelerator key. In this case, the user's selection of **Exit** generates a command event to perform the *close* action on the frame window.

Note: You cannot promote menu item events to the part interface.

The other menu choices, shown in Figure 110 on page 243 connect to other parts that are not on this free-form surface. These connections are completed in “Completing the Menu Bar” on page 269.

Developing Applications

Chapter 14. Adding Help to Visual Builder Applications

In this section, you learn about adding different types of help to your application. Generally, in application development, you develop your user interface and then provide the necessary help panels. The basic help types are as follows:

Help Type	Description
Context-sensitive help	<p>Help information for the current choice, object, or group of choices or objects. The user can display context-sensitive help by tabbing or cursoring to a subpart and doing either of the following:</p> <ul style="list-style-type: none"> Pressing the F1 key. You do not have to do anything extra to make this happen. The operating system (through the IBM Open Class Library) handles it for you. Selecting the Help push button if you have provided one. See “Providing a Help Push Button” on page 257 for information on how to do this.
General help	<p>Help for a specific window, explaining the purpose of the window and how it operates.</p>

To show you how to add help, we are going to write help for the Request for Skill Information window of the OASearch application, introduced in “The Query Windows” on page 96.

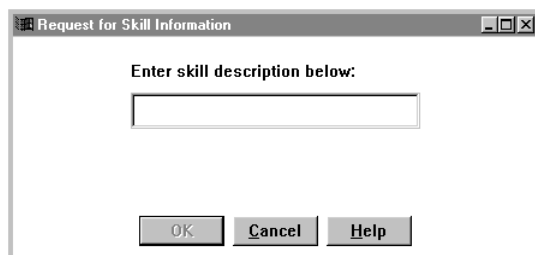


Figure 113. The Request for Skill Information Window (OAQuerySkill)

The example we use in this chapter assumes that you have already built the Opportunities Abound Skill Information window for the OASearch application. If

Developing Applications

you want to follow the example, you must build this view first. To follow along without building the part, see the OAQuerySkill part in oawin.vbb.

The first step in providing help is to create the help file.

Creating the Help File

Before connecting the help, create the help file.



When creating help files, try using one of the two help projects shipped with WorkFrame: IPF documenter help or RTF help. These formats are outlined in the following section.

Writing portable help

If you want your help file to be usable in both OS/2 and Windows, write your help source code in Information Presentation Facility (IPF) format. For more information about creating help source code in IPF format, refer to the *IPF User's Guide*.

If portability is not a concern for your Windows application, you can write your help source code in Rich Text Format (RTF). For information about creating help source code in RTF, see your Windows documentation.

Note: For your convenience, the following IPF help source code appears in the online version of this book, so you do not have to type it yourself.

Using your favorite editor, create a new file and type in the following help source. Save this file with a name similar to cppwv33.ipf.

Note: Be sure to use an .ipf extension when naming the file.

Developing Applications

```
:userdoc.  
:title.Opportunities Abound Databases Help  
:docprof toc=1 ctrlarea=page.  
:ctrl ctrlid=buttons controls='ESC SEARCH PRINT' page.  
:h1 res=9998.  
General Help for the Opportunities Abound Databases  
:il.general help  
:p.Text goes here.  
:h1 res=9999.  
Request for Skill Information Help  
:il.requesting skill information, help for  
:p.Text goes here.  
:h1 id=10000.  
Skill Information Help  
:il.skill information, help for  
:p.Text goes here.  
:h1 res=10001.  
Request for Contract Information Help  
:il.requesting contract information, help for  
:p.Text goes here.  
:h1 res=10002.  
Contract Information Help  
:il.contract information, help for  
:p.Text goes here.  
:h1 res=10003.  
Request for Contractor Information Help  
:il.requesting contractor information, help for  
:p.Text goes here.  
:h1 res=10004.  
Contractor Information Help  
:il.contractor information, help for  
:p.Text goes here.  
:euserdoc.
```

In the previous sample, the `:h1` tags are heading tags. These tags cause IPF to create a new help panel using the text on the tag as the panel's title. The *res* parameter specifies the panel's resource ID.

If using RTF help, you must define resource IDs for each panel in your .rtf file.

Make note of all panel resource IDs; you will need them in order to set help support properly in your visual parts. See "Providing Context-Sensitive Help" on page 252 to learn how these resource numbers are used.

Building an IPF help file

Ensure that you have the Information Presentation Facility (IPF) compiler installed on your system and that your environment variables are set up to run the IPF compiler. The IPF compiler comes with VisualAge for C++.

Developing Applications

To build the help file, simply run the IPF compiler. For example, if you saved your help file with the name `cppwv33.ipf`, you would enter the following command in the directory where you saved your file:

```
ipfc cppwv33.ipf
```

This command generates a file called `cppwv33.hlp`.

You have now built your help file. The next step is to provide context-sensitive help in your application.

Providing Context-Sensitive Help

This section tells you how to provide context-sensitive help for subparts in your application.

For this example, use the entry field in the `OAQuerySkill` part. Before you begin, make sure you have a list of your help panel titles and their corresponding resource IDs.

To provide context-sensitive help, do the following:

1. Open the settings notebook for the part. See “Changing Settings for a Part” on page 138 if you need information on how to do this.
2. Select the **Control** page. It looks like Figure 114 on page 253.

Developing Applications

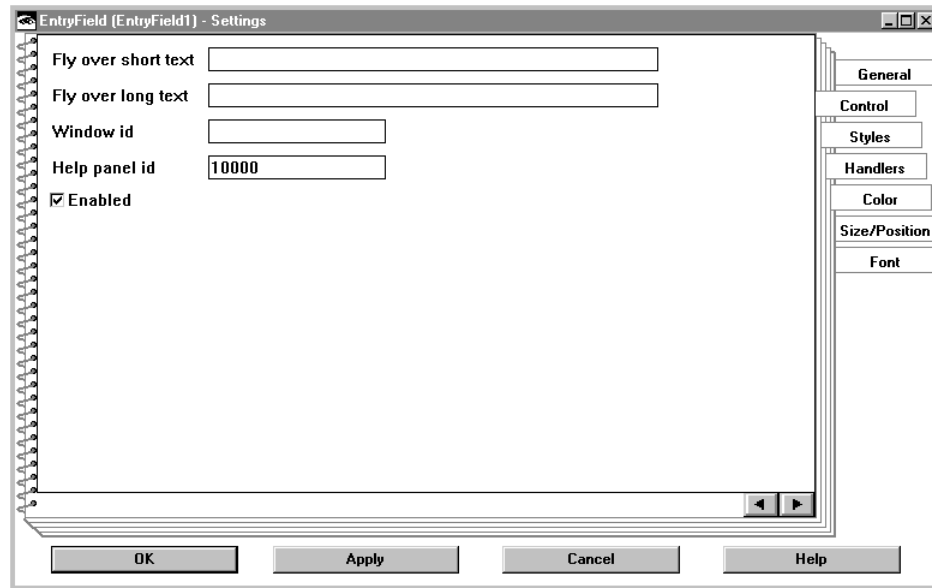


Figure 114. The Control Page of the Skill Description Entry Field's Settings Notebook

3. Enter the resource ID for the appropriate help panel in the **Help panel ID** field. In this example, use 10000 for the Skill Information Help panel.

When you generate the code for your part, Visual Builder creates a help table in the resource (.rci) file that it generates and inserts this number into the help table.

4. If the **Enable** check box is not checked, select it.
5. Select the **OK** push button.

The next step is to provide general help in your application.

Providing General Help

This section tells you how to provide general help for your application.

For this example, use the OAQuerySkill part. Before you begin, make sure you have a list of your help panel titles and their corresponding resource IDs.

To provide general help, do the following:

1. Open the settings notebook for the IFrameWindow*-based part. See “Changing Settings for a Part” on page 138 if you need information on how to do this.
2. Select the **Control** page. It looks like Figure 115 on page 254.

Developing Applications

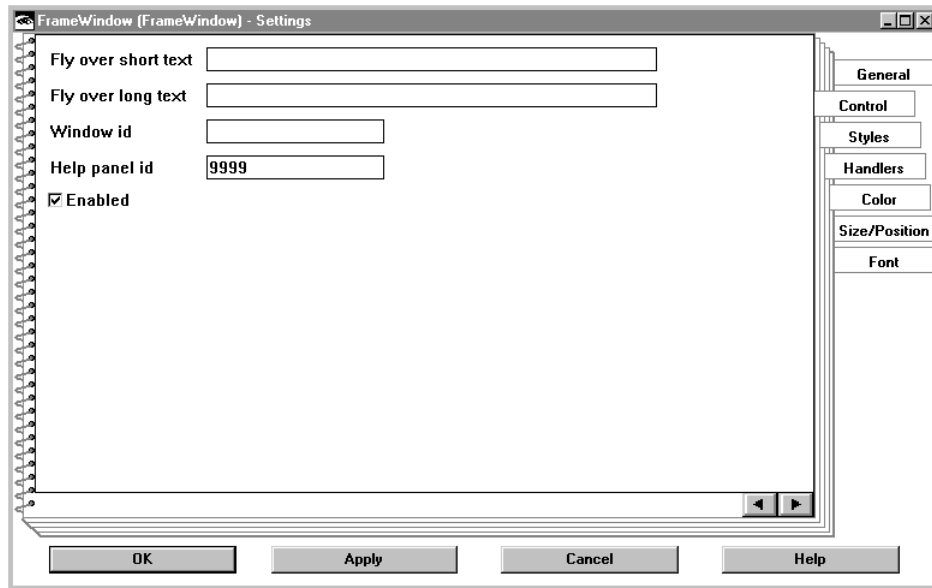


Figure 115. The Control Page of the Frame Window's Settings Notebook

3. Enter the resource ID for the general information help panel in the **Help panel ID** field. For this example, use 9999 for the Request for Skill Information Help panel.

When you generate the code for your part, Visual Builder creates a help table in the resource (.rci) file that it generates and inserts this number into the help table.

4. If the **Enable** check box is not checked, select it.
5. Select the **OK** push button.

The next step is to provide a help window to display the help panels in.



Providing the Application Help Window

Once you have added the context-sensitive help and the general help for the main window, you need a help window to display the help panels in. You must place an `IHelpWindow*` part on the free-form surface to give Visual Builder a window in which to display the help information.

The default owner of the `IHelpWindow*` part is the primary part for your application. If all parts with help enabled are subparts of the primary part, no connections are required. For a more complex implementation where connections are required, see the `OAMain` part in `oawin.vbb`.

Developing Applications

To add a help window to your application, do the following:

1. Select , the Other category, on the parts palette.
2. Select , the IHelpWindow* part, and place it on the free-form surface.
3. Open the settings notebook for the IHelpWindow* part. See “Changing Settings for a Part” on page 138 if you need information on how to do this. It looks like Figure 116.

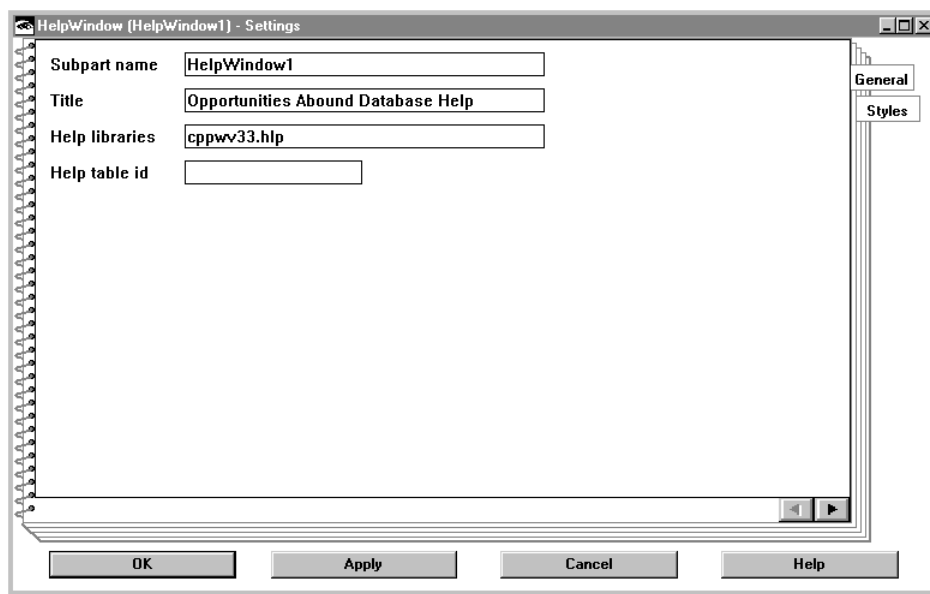


Figure 116. The Help Window's Settings Notebook

4. In the **Title** field, enter the title of the help window.
This must be the same title that you entered on the :title tag in your help source file.
5. In the **Help libraries** field, enter the name of the help file that you compiled, such as cppwv33.hlp.
If you had created multiple help files for your application, you would enter all of their names in this field.

Developing Applications

6. We recommend that you leave the **Help table ID** field empty and let Visual Builder generate it for you unless you need a specific help table ID. Otherwise, you could have conflicts in your resource (.rci) file.
7. To use IPF help in Windows, do the following:
 - a. Select the **Styles** notebook tab.
 - b. Next to the **ipfCompatible** style, select the **On** radio button.
8. Select the **OK** push button.

Providing Help for Factory-Generated Frame Windows

You can associate a help window with a part that is generated by an object factory if the base class of that part is `IFrameWindow*`. To do this, you can do either of the following:

- Edit the part that the object factory generates, place an `IHelpWindow*` part on the free-form surface next to it, and do all the things described in the preceding sections to use the help window properly. Repeat this step for each `IFrameWindow*`-based part that is generated by an object factory.

Use this method if you have created multiple help library (.hlp) files for your library instead of putting all of your help panels in one library file.

- Place an `IHelpWindow*` part on the free-form surface in the same view with the object factory if you have created only one help library file for all of your help panels. Then, do the following:
 1. Open the settings notebook for the `IHelpWindow*` part.
 2. Enter the help window title and the name of the library (.hlp) file.
 3. Close the settings notebook by selecting the **OK** push button.
 4. Connect the *newEvent* feature of the object factory to the *setAssociatedWindow* action of the help window. If the object factory represents a modal window, this connection must occur before the object factory's *showModally* connection. For an example, see the `OAMain` part in `oawin.vbb`.
 5. Repeat step 4 for each object factory in the view.
 6. Edit each `IFrameWindow*`-based part that an object factory generates and specify the appropriate help panel IDs for the subparts that you want to provide help for. Each help panel ID must be a resource ID in the library file that you specified in step 2.



For an example using one help file, see the `OAMain` part in `oawin.vbb`.

Providing a Help Push Button

The main window of the OASearch application, which we have been using for this help example, does not contain a **Help** push button. However, many applications provide such a push button to give users quick and easy access to the help information that the application provides.

To provide a **Help** push button in your application, do the following:



1. Select  , the Buttons category, in the left-hand column on the parts palette.
2. Select  , the IPushButton* part, and place it where you want it to be.
3. Change the text on the push button to Help.
4. Open the settings notebook for the push button.
5. Select the **Styles** tab.
6. Find **help** on the **Styles** page and select the **On** radio button. This style turns a regular push button into a help push button.
7. On the same page, find **noPointerFocus** and select the **On** radio button.

This style keeps the **Help** push button from getting the input focus when a user clicks on it. By setting this style, you enable your application to display help for the part that has the input focus when a user clicks the **Help** push button.

Otherwise, the user sees the help panel that you assign to the **Help** push button.

When this style is set, the user must use the cursor keys to set the input focus on the **Help** push button. After doing this, the user can click on the **Help** push button to display the help panel that you assign to it.

8. Select the **OK** push button to close the settings notebook.

You now have a **Help** push button. If you have followed the steps in the preceding sections, clicking this button causes the contextual help panel for the part that currently has the input focus to be displayed. If no part has the focus, the main help panel for the window is displayed. The behavior of the **Help** push button is identical to that of the F1 key.

To provide a help panel for the **Help** push button itself, follow the instructions in “Providing Context-Sensitive Help” on page 252.

Displaying Fly-Over Help When the Mouse Pointer Is Over a Part

Another type of help that you can provide in your application is called *fly-over help* (or *hover help*), which is intended to provide instant help for novice users. This type of help consists of a text string that your application displays when the user positions the mouse pointer over a subpart, such as a push button or list box. The text string should be short and precise, giving the user information such as the purpose of the subpart.

Your application can provide the following types of fly-over help:

- A short text string that your application displays next to the subpart that the mouse pointer is over
- A longer text string that your application displays in a text control


Note: You cannot use fly-over help for the following controls:


- Group box
- Outline box
- Viewport

Providing fly-over help for a subpart when 'online' insert

To provide fly-over help for a subpart, do the following:

1. Place an IVBFlyText* part on the free-form surface by doing the following:

- a. Select  , the Other category, on the parts palette.

- b. Select  , the IVBFlyText* part, and place it on the free-form surface.

2. Add fly-over help text to one or more subparts by doing the following:
 - a. Open the settings notebook for a subpart, such as an entry field or push button.
 - b. Select the **Control** notebook tab.
 - c. Enter fly-over text strings in the **Fly-over short text** field, the **Fly-over long text** field, or both.

Text that you enter in the **Fly-over short text** field is displayed in a pop-up window next to a subpart when the user positions the mouse pointer over that subpart.

Developing Applications

Text that you enter in the **Fly-over long text** field is displayed in a text control that you specify. For example, you might add an information area to a window and display the long fly-over help text there. See “Displaying Long Fly-Over Text In an Information Area” on page 260 to see how this is done.



If one of your application views contains two or more frame windows, do not use the same IVBFlyText* part to assign long fly-over text strings to text controls in each window. If you do, your fly-over help text probably will not appear in the window that you expect it to appear in.

For example, suppose your view contains two frame windows: one that is displayed when your application starts and another one that is displayed when a menu item in the first window is selected. In this case, place two IVBFlyText* parts on the free-form surface. The first IVBFlyText* part is associated by default with the window that is displayed when your application starts. However, you must specify the owner of the second IVBFlyText* part by opening the IVBFlyText* part's settings notebook and entering the name of the owner part in the **Owner** field, such as iFrameWindow1.

- d. Select the **OK** push button to save the text strings that you just entered.

That is all you need to do. There are no connections to make.

Displaying Help in an Information Area

There are times when it is helpful to provide an *information area* in an application window that your application can use to give the user feedback. For example, the information area might let the user know whether the application performed an operation successfully or it might provide a description of a menu choice.


Visual Builder provides the IVBInfoArea* part that you can use for this purpose. An information area is considered to be an extension of a frame window. You can only add it to an IFrameWindow* part. Visual Builder positions the information area at the bottom of the frame window's client area.

Developing Applications

Adding an Information Area to a Frame Window

To add an information area to a frame window, do the following:



1. Select , the IVBInfoArea* part.
3. Move the mouse pointer over the title bar or window border of the frame window and click mouse button 1.

Visual Builder places an information area at the bottom of the frame window.

4. Open the settings notebook for the information area.
5. Enter text that you want your application to display in the following fields:

Disabled text

Text to display when the selected menu choice is disabled.

Inactive text

Text to display when no menu choice is selected.

Missing text

Text to display when the information area cannot find and display specific help for a menu choice.

Displaying Help for Menu Choices in an Information Area

To display help text for menu choices in an information area, do the following:

1. Create the menu bar and pull-down menus for your application.
2. Open the settings notebook for each menu choice.
3. Enter a description of the menu choice in the **Info area text** field.
4. Select the **OK** push button to save the description you just entered.

Displaying Long Fly-Over Text In an Information Area

Earlier in this chapter, we showed you how to add an IVBFlyText* part and display a short text string in a pop-up window. This section shows you how to display a longer text string in an IVBInfoArea* part, although you can use any text control, such as an IEntryField*.

To display a long fly-over text string in an information area, do the following.

Note: The following steps assume that you have already added an IVBFlyText* part and an IVBInfoArea* part to a frame window. If you have not, you should complete the steps in the following sections and then return here:

Developing Applications

- “Displaying Fly-Over Help When the Mouse Pointer Is Over a Part” on page 258

When following these steps, make sure you enter a text string in the **Fly-over long text** field in the settings notebook for a subpart, such as an entry field or push button.

- “Adding an Information Area to a Frame Window” on page 260

1. Display the connection menu for the information area.
2. Select the *this* attribute.
3. Display the connection menu for the fly-over text part.
4. Select the *longTextControl* attribute.

Displaying Information about Successful Actions

One way to make your applications more user-friendly is to provide feedback to the user when an action or member function performs successfully, such as the successful completion of an action that is triggered by a *buttonClickEvent* feature. We recommend that you display this information in an *IVBInfoArea** part.

To display information in an information area when an action or member function completes successfully, do the following:

1. Create an event-to-attribute connection using the *text* attribute of the *IVBInfoArea** part as the target. The source event can be one of the following:
 - The same event that triggered the action or member function

If the action or member function was triggered by an event, you can use the same event, such as the *buttonClickEvent* or *commandEvent* feature. In this case, be sure to check the connection order to make sure the event-to-action or event-to-member function connection occurs before the event-to-attribute connection.

 - The *actionResult* event of the connection that triggered the action or member function

All connections have both an *actionResult* event and an *actionResult* attribute. Therefore, be sure you use the *actionResult* event as the source of the connection.
2. Double-click on the connection you just made to open the settings window for the connection.
3. Select the **Set parameters** push button to open the Constant Parameter Value Settings window.
4. In the **text** field, enter the text string that you want to assign to the *text* attribute.

Developing Applications

Your application will display this text string in the information area each time the action or member function completes successfully.

5. Select the **OK** push button to close the Constant Parameter Value Settings window.
6. Select the **Cancel** push button to close the connection settings window.

Chapter 15. Integrating Visual Parts into a Single Application

In this chapter, you combine all visual parts of the OASearch sample application in preparation for generating code and compiling. This chapter includes the following tasks:

- Adding nonvisual support parts to the primary part
- Adding static visual parts
- Adding visual parts as dynamic instances
- Making the final connections

To follow this example, load oanonvis.vbb and oawin.vbb into Visual Builder.

Adding Nonvisual Support Parts to the Primary Part

For this example, refer to the primary view, OAMain, in oawin.vbb, shown in Figure 117.

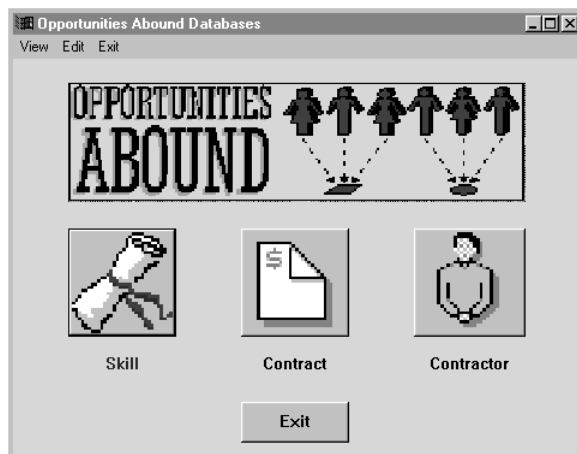


Figure 117. Completed Opportunities Abound Databases Window

1. Begin by building the primary part as you would any IFrameWindow*-based part. For information about building the menu bar, see Chapter 13, “Adding Menus to Your Application” on page 243.

Developing Applications

2. The OASearch application starts up with a small number of static nonvisual instances to support the visual parts. Add the following parts:

- OAContractor*. Name it Contractor.
- OAContract*. Name it Contract.
- OASkill*. Name it Skill.
- OASkillBase*. Name it SkillBase.

You must connect these part instances to the variables that represent them in the other visual parts. First, you must add the visual parts.

Adding Static Parts

One visual part, OAGenInfo, exists statically in the OASearch application. The disadvantage of using static visual parts is that once the user closes the window, the part is destroyed and cannot be instantiated again in the same session.

Add the OAGenInfo* part. Name it VBDevelopment. You must also connect OAGenInfo* to the menu item that causes it to display. For details, see “Completing the Menu Bar” on page 269.

Adding Visual Parts as Dynamic Instances

All OASearch visual parts except for OAMain and OAGenInfo are created at run time as the user requests them from a menu or button selection. The dynamic windows are represented in the OAMain part by object factory parts. Like static visual parts, dynamic visual parts are destroyed when the user closes them. However, the existence of the object factory enables the creation of a new instance of that same part the next time the user requests it.

Like variable parts, object factory parts are placeholders for other parts. Each object factory part is set to the part it represents. The object factory part works in tandem with a variable part that represents the dynamic part instance. You can use object factory parts to create both visual and nonvisual parts.

Unlike variable parts, object factory parts run the corresponding part class constructor, creating a new instance. Variable parts just stand in for instances created elsewhere.

Implementing dynamic parts involves the following tasks:


- Adding and setting object factory parts
- Adding variable parts
- Connecting to the object factory parts
- Connecting the object factory parts to the corresponding variable parts

Adding and Setting Object Factory Parts


Before you can use object factory parts, you must have created the part classes to be represented. Before you start, be sure to load the .vbb files that contain those parts into Visual Builder.

Adding the object factory parts




1. Select , the **Models** category, from the left side of the parts palette.
2. Select the **Sticky** check box.



3. Select , the IVBFactory* part, from the right side of the parts palette.
4. Drop six IVBFactory* parts on the free-form surface.



5. Unload the mouse pointer by selecting , the Selection tool, from the Visual Builder tool bar.

Setting an object factory part

1. Change the part name using the Composition Editor contextual menu.

In the OASearch example, the IVBFactory* parts are named as follows:

ConQFac	Request for Contract Information window
ConVFac	Contract Information window
CtrQFac	Request for Contractor Information window
CtrVFac	Contractor Information window
SklQFac	Request for Skill Information window
SklVFac	Skill Information window

2. Change the part type from the default (IStandardNotifier*) using the Composition Editor contextual menu.

In the OASearch example, the IVBFactory* parts have the following types:

ConQFac	OAQueryContract*
ConVFac	OAContractView*
CtrQFac	OAQueryContractor*
CtrVFac	OAContractorView*
SklQFac	OAQuerySkill*
SklVFac	OASkillView*

3. Set each IVBFactory* part to automatically delete each instance.

Developing Applications

Set modeless windows as follows. In the OASearch sample, VBDevelopment and the xVFac parts are modeless.

- Open the settings notebook for the part.
- Switch to the **General** page.
- Select the **AutoDelete** check box.

Dispose of modal windows using *deleteTarget* connections. For more information about modal windows and *deleteTarget* connections, see “Connecting the Object Factory Parts to Their Corresponding Variable Parts” on page 267.

Adding Variable Parts

When used with object factory parts, variable parts represent the newly created part instance. Add and set variable parts as follows:

Name	Type
ContractQ	OAQueryContract*
ContractorQ	OAQueryContractor*
SkillQ	OAQuerySkill*
ContractV	OAContractView*
ContractorV	OAContractorView*
SkillV	OASkillView*

For more information on variables, see “Adding a Variable to a Composite Part” on page 160.

Connecting to the Object Factory Parts

Once you have added and set both the object factory and variable parts, connect the IGraphicPushButton* parts on the welcome window to the object factory parts representing the query windows, as follows:

From part, feature	To part, feature
GraphicPushButton1, #buttonClickEvent	SklQFac, #new
GraphicPushButton2, #buttonClickEvent	ConQFac, #new
GraphicPushButton3, #buttonClickEvent	CtrQFac, #new

Next, connect promoted button actions in the query windows to the object factory parts representing the information windows, as follows:

From part, feature	To part, feature
ContractQ, #okPBBUTTONClickEvent	ConVFac, #new
ContractorQ, #okPBBUTTONClickEvent	CtrVFac, #new
SkillQ, #okPBBUTTONClickEvent	SklVFac, #new

Connecting the Nonvisual Parts to the Variables that Represent Them

Each visual part contains promoted variables as placeholders for the static nonvisual parts dropped in OAMain. This is also true for the object factory parts. Now, connect the static nonvisual parts to the variables that represent them as follows:

From part, feature	To part, feature
ConQFac, #contract	Contract, #this
ConVFac, #contract	Contract, #this
CtrQFac, #contractor	Contractor, #this
CtrQFac, #skillBase	SkillBase, #this
CtrVFac, #contract	Contract, #this
CtrVFac, #contractor	Contractor, #this
CtrVFac, #skillBase	SkillBase, #this
SklQFac, #skill	Skill, #this
SklQFac, #skillBase	SkillBase, #this
SklVFac, #skill	Skill, #this
SklVFac, #skillBase	SkillBase, #this

Connecting the Object Factory Parts to Their Corresponding Variable Parts

Once you have set both object factory and variable parts, you must connect them. In OAMain, the connections vary depending on whether the window is *modal*. Modal windows retain focus until they are closed; the user cannot switch focus to another window without closing the modal window. In OAMain, the query windows are modal; the other windows are modeless.

Note: You cannot make static windows modal in Windows. The owner of a modal window must be assigned in the window's constructor call. Static windows are constructed when the application is started. The *owner* connections that follow do not run until after the static windows have already been constructed.

Connections for modal windows

Make the following connections for the query window parts. The *deleteTarget* connections dispose of pointers to the windows once they have been closed.

Developing Applications

From part, feature

ConQFac, *#newEvent*
ConQFac, *#owner*
ConQFac, *#newEvent*
ConQFac, *#newEvent*
ConQFac, *#newEvent*
CtrQFac, *#newEvent*
CtrQFac, *#owner*
CtrQFac, *#newEvent*
CtrQFac, *#newEvent*
CtrQFac, *#newEvent*
SklQFac, *#newEvent*
SklQFac, *#owner*
SklQFac, *#newEvent*
SklQFac, *#newEvent*
SklQFac, *#newEvent*

To part, feature

ContractQ, *#this*
FrameWindow, *#this*
ContractQ, *#setFocus*
ContractQ, *#showModally*
ContractQ, *#deleteTarget*
ContractorQ, *#this*
FrameWindow, *#this*
ContractorQ, *#setFocus*
ContractorQ, *#showModally*
ContractorQ, *#deleteTarget*
SkillQ, *#this*
FrameWindow, *#this*
SkillQ, *#setFocus*
SkillQ, *#showModally*
SkillQ, *#deleteTarget*

Connections for modeless windows

Now make the following connections for the modeless information windows. The connections for VBDevelopment are listed in “Completing the Menu Bar” on page 269.

From part, feature

ConVFac, *#newEvent*
ConVFac, *#newEvent*
ConVFac, *#newEvent*
CtrVFac, *#newEvent*
CtrVFac, *#newEvent*
CtrVFac, *#newEvent*
SklVFac, *#newEvent*
SklVFac, *#newEvent*
SklVFac, *#newEvent*

To part, feature

ContractV, *#this*
ContractV, *#setFocus*
ContractV, *#visible*
ContractorV, *#this*
ContractorV, *#setFocus*
ContractorV, *#visible*
SkillV, *#this*
SkillV, *#setFocus*
SkillV, *#visible*

For each connection that has the *visible* feature as its target, you must do the following after making the connection:

1. Double-click on the connection line.

The settings notebook for the connection opens.

Note: Do not double-click on one of the black squares that show that the connection line is selected. If you do, the settings notebook does not open.

2. Select the **Set parameters** push button.

A window opens for setting the parameters.

3. Select the **visible** check box.

Developing Applications

If you do not select this check box, the window will not be visible when you run your compiled application.

4. Select the **OK** push button.

The window for setting parameters closes.

5. Select the **OK** push button on the settings notebook.

The settings notebook closes.

Completing the Menu Bar

Now that all parts appear on the free-form surface, you can complete the menu bar. The connections echo those made in “Connecting to the Object Factory Parts” on page 266.

Connections from the View submenu

From part, feature

MenuItem3, *#commandEvent*
MenuItem4, *#commandEvent*
MenuItem5, *#commandEvent*
MenuItem1, *#commandEvent*
MenuItem1, *#commandEvent*

To part, feature

ConQFac, *#new*
CtrQFac, *#new*
SklQFac, *#new*
VBDevelopment, *#setFocus*
VBDevelopment, *#visible*

Be sure to open the settings notebook for the connection that has the *visible* feature as its target, select the **Set parameters** push button, and then select the **visible** check box. If you need more information, see “Connecting the Object Factory Parts to Their Corresponding Variable Parts” on page 267.

Connections from the Edit submenu

From part, feature

MenuItem7, *#commandEvent*
MenuItem9, *#commandEvent*

To part, feature

ConVFac, *#new*
CtrVFac, *#new*

Once all connections are made, the OAMain part is essentially complete. Make sure to save your work.

Resetting Your Application's Main Resource Library

In order to ship your completed application to customers that do not have VisualAge for C++ installed, you must ship certain IBM product runtime DLLs with your compiled application. Many of these DLLs must be renamed. This process is outlined in “Packaging Runtime DLLs with Your Application” on page 279.

Visual Builder-generated code sets the main resource library for the application to a predefined Open Class resource DLL that contains default bitmaps for operations like **Open** and **Cut**. In OS/2, this DLL is called `cppoor3u`; in Windows, this DLL is called `cppwor3u`.

The main resource library cannot be reset once the application instance has been constructed, so an additional nonvisual part is required to reset the resource library and construct the main part of your application. The `OASearch` part is provided in `oanonvis.vbb` for you to use as an example.

To construct this nonvisual part, follow these steps:

1. Open a new nonvisual part.
2. On the Composition Editor, drop an `IVBFactory*` part and change its type to the main part of your application. For `OASearch`, this part is of type `OAMain*`.
3. Drop an `IVBVariable*` part and change its type to the main part of your application. For `OASearch`, this part is of type `OAMain*`.
4. Make the following connections:
 - a. From the free-form surface, connect the *ready* event of the composite part to the *new* event of the `IVBFactory*` part.
 - b. From the `IVBFactory*` part, connect the *newEvent* feature of the `IVBVariable*` part to the *this* attribute of the `IVBVariable*` part.
 - c. From the `IVBFactory*` part, connect the *newEvent* feature of the `IVBVariable*` part to the *setFocus* action of the `IVBVariable*` part.
 - d. From the `IVBFactory*` part, connect the *newEvent* feature of the `IVBVariable*` part to the *visible* attribute of the `IVBVariable*` part.
 - e. Open the settings for this connection and select the **Set parameters** push button. Select the **visible** check box.
 - f. Close the settings for the connection.
5. Open the settings for the `IVBFactory*` part. Select the **autoDelete** check box.
6. In the Class Editor, enter the following in the **User constructor code** field:

```
IApplication::current().setResourceLibrary("dll_name");
```

Developing Applications

dll_name represents the renamed resource DLL, without the .dll extension.

7. Save the part.

As shipped by IBM, the main part of the OASearch application is OAMain. You generate the main() function for the OAMain part. The compiled program is called oamain.exe.

If the sample program is built for DLL redistribution, the main part of the OASearch application becomes OASearch. In this case, you generate the main() function for the OASearch part. The compiled program is called oasearch.exe.

Developing Applications



Chapter 16. Generating Source Code for Parts and Applications

All work that you do using Visual Builder is stored within the Visual Builder development environment, including information about visual parts, nonvisual parts, and connections.

This chapter describes all you need to know about getting your Visual Builder application ready for others to use. The topics covered are as follows:

- Preparing for source code generation
- Generating C++ source code for the individual parts used within your application
- Generating C++ source code for your application (the `main()` function)
- Preparing generated files for compilation
- Compiling and linking your application

Preparing for Source Code Generation

Before getting started, decide how you want to build your application's make file. You can create make files in either of the following ways:

- From Visual Builder.
- Using WorkFrame's `makemake` program. For more information, see the *User's Guide*.

You must also decide whether you want Visual Builder to generate Hypertext Markup Language (HTML) documentation for your parts. For more information, see "Setting Up Visual Builder to Generate HTML Documentation" on page 274.

Setting Up Visual Builder to Generate Make Files

If you want Visual Builder to generate make files with the C++ source code, follow these steps from the Visual Builder window:

1. From the menu bar, select **Options**.
2. Select **Generate make files**.

The next step is described in "Generating C++ Source Code for Individual Parts" on page 274.

Developing Applications

Setting Up Visual Builder to Generate HTML Documentation

If you want a summary of your part's implementation, you can choose to have Visual Builder generate Hypertext Markup Language (HTML) documentation. The .htm file describes valid features for the part and lists all connections to or from the part.

To get HTML documentation, you must set an environmental variable before starting Visual Builder, as follows:

```
SET VBGENDOC=HTML
```

The default value for this variable is NO.

The next step is described in "Generating C++ Source Code for Individual Parts."


Generating C++ Source Code for Individual Parts

After you finish constructing a part, you must generate source code. If the part is an application in itself or if you want to test a part individually, you must also generate the main() function.

You can generate source code for the part being edited from any of the Visual Builder editors.

1. From the editor's menu bar, select **File**.
2. Select **Save and generate**; then select **Part source**.



If you are using the Composition Editor, you can select , the Generate Part Code tool, from the tool bar instead. There is no difference between selecting this icon and using the menu item described previously.



One of the most common causes of code generation errors is changing the names of features that are connected to other features. For example, suppose feature A is connected to feature B. If you change the name of feature A and then regenerate the source code for your part, Visual Builder displays an error. This can also occur if you change the name of a promoted feature. To correct the error, double-click on the connection and replace the incorrect feature name with the correct one.

Some capabilities found in OS/2 parts are not available in Windows. If you plan to generate Windows-based code for parts created with the OS/2-based product, review the contents of the log window after code generation. Visual Builder writes warning messages to the log window for settings or connections to features that are not portable from OS/2 to Windows.

Developing Applications

Because Visual Builder cannot discern why the settings or features are not currently valid, it writes the same message for nonportable items as it does for items that are no longer valid because the part interface has changed. If you see the following message, the setting that corresponds to the specified attribute is not currently valid:

The X attribute of the Y part cannot be set.

You see the following message if a feature that is not currently valid is either the source or target of a connection:

The X feature was not found in the part interface for Y.

If a feature that is not currently valid is used to supply a parameter value in a connection, you see the following message:

The X feature of the Y part has changed since this part was developed.

For more information about the files generated, see “Source Files Created during Part Code Generation.”

The next step is described in “Generating Source Code for Your Application’s main() Function” on page 276.

Source Files Created during Part Code Generation

For each part processed, Visual Builder generates several source code files. As an example, the following files are created for the OAContractView part:

contrctg.cpp	A C++ code file.
contrctg.hpp	The header file for contrctg.cpp.
contrctg.h	A resource header file for the .cpp file. This file contains the resource IDs for your part.
contrctg.htm	An HTML file, if you opted to have Visual Builder generate one.
contrctg.rci	A resource file that contains any text strings used in the part for entry field labels, push buttons, menus, and so forth.

If you selected **Default to FAT file names** under the **Options** pull-down menu of the Visual Builder window and your part name has more than eight characters, Visual Builder creates an eight-character name for the part when it is created.

Note: If you are using the File Allocation Table (FAT) file system, we recommend that you always use part names and file names that have eight characters or fewer, even if you have selected the **Default to FAT file names** option. Otherwise, Visual Builder might use a file name for a part file that is the same as one that already exists and write over the existing file.

Developing Applications

If you decide to switch a pre-existing part to FAT file naming, be sure to change the file names for code generation in the Class Editor for the part.

You must specify a starting resource ID in all parts for which you want Visual Builder to export translatable strings into resource files. If you do not specify a starting resource ID or indicate that you want the default, Visual Builder writes only text associated with fly-over help and the information area to the resource files. For more information about resource IDs and resource files, see “Using Resource Files for Translation” on page 303. For information about how to specify a resource ID, see “Specifying a Starting Resource ID” on page 53.

Generating Source Code for Your Application's main() Function

To create an executable application, you must generate code for the standard C++ main() function. You can do this for parts that you want to test individually or for your entire application. If you want to compile your entire application, generate the function using the part that represents your application's primary view.

You must first generate the C++ code for all parts that you intend to compile in your application. Load into Visual Builder the part files that represent all parts that will appear in the compiled application. Then, generate the main() function from the part that you want to appear first when your application is started (the *main part*).

You can generate application source code for the main() function as follows if the main part is displayed in any of the Visual Builder editors:

1. From the editor's menu bar, select **File**.
2. Select **Save and generate**; then select **main() for part**.

For more information about the files generated, see “Source Files Created during Generation of main() Function Code.”

Source Files Created during Generation of main() Function Code

For each main part processed, Visual Builder creates several files. For the OAMain part, the following files are created:

oamain.app	The C++ code file containing the main() function declaration. Note: If you start Visual Builder from a WorkFrame project, a file named vbmain.cpp is generated instead of a file named oamain.app.
oamain.mak	A make file, if you opted to generate make files using Visual Builder.

Developing Applications

`oamain.rcx` The main resource file. It includes a resource (.rci) file for each part that you generated part source for.

If you selected **Default to FAT file names** as a preference under the **Options** pull-down menu of the Visual Builder window and your part name has more than eight characters, Visual Builder creates an eight-character name for the generated files.

Note: If you are using the File Allocation Table (FAT) file system, we recommend that you always use part names and file names that have eight characters or fewer, even if you have selected the **Default to FAT file names** option. Otherwise, Visual Builder might use a file name for a part file that is the same as one that already exists and write over the existing file.

If you decide to switch a pre-existing part to FAT file naming, be sure to change the file names for code generation in the Class Editor for the part.

Preparing Generated Files for Compilation

Before compiling your application, be sure you have the following files:

- Header files for all parts.
- .cpp files for all parts.
- Resource files (.rci) for all parts. If you are using class interface parts, make sure that you have .rci files for them as well. For parts that do not otherwise require them, consider creating .rci files with just a comment line, as follows:

```
// Resource file for PARTNAME class interface part
```

- A make file.
- An .app file for the main part.

Note: If you started Visual Builder from a WorkFrame project, you should have a file named `vbmain.cpp` instead of an .app file.

Final preparations for compilation and linking include the following:

- Specifying additional libraries (C++ libraries and DLLs) in the make file
- Specifying debug options for the compiler and linker programs

Developing Applications

Specifying Additional Libraries in the Make File

Review the list of libraries specified in the make file, particularly libraries for parts that you compiled separately.

If your application uses DLLs, add the DLL names as dependent files in the description blocks used to keep the object files up to date. The order in which you list object files is significant. Files with external references must occur after the referred-to files.

Specifying the Option to Generate Browser Information

If you want the VisualAge for C++ compiler to generate Browser information, you must include the Fb+ option when you compile your application. This option causes the compiler to generate a file with an extension of .pdb. Once this file is generated, you can use the Browser data when connecting features to member functions and for other purposes. For more information, see “Using Browser Information” on page 182.

Specifying Debug Options for the Compiler and Linker Programs

If you prefer to compile and link your application using WorkFrame, you should have already specified debug options in your Visual Builder project file. For more information about the options you need to set, see the WorkFrame documentation.

To make Visual Builder connections easier to debug, set up a trace by following these steps:

1. In your config.sys file, add this line:

```
SET ICLUI TRACE=ON  
SET ICLUI TRACETO=OUT
```

2. When you compile and link your application, specify the vbdebug flag. For example, compile myapp.exe as follows:

```
NMAKE VBDEBUG=1 myapp.mak
```

The VBDEBUG flag setting determines whether compilation flags are set to enable tracing. A value of 1 indicates that you want tracing enabled and debug information to be generated. If you do not specify a value, optimization is turned on, and tracing is not enabled.

3. When you run your compiled application, redirect the output to a file. For example, run myapp.exe as follows on Windows NT:

```
myapp >myapp.out 2>&1
```

Run myapp.exe as follows on Windows 95:

```
myapp >myapp.out
```

4. Browse the output file (myapp.out) to see what connections were run and in what order. The file also lists any exceptions that were thrown.

Packaging Runtime DLLs with Your Application

To distribute your application to customers that do not have VisualAge for C++ installed, you must ship certain IBM product runtime DLLs with your compiled application. These DLLs fall into several categories, as follows:

- C runtime file.
- Open Class files, depending on the classes you use in your application. These must be renamed.
- Visual Builder files: cppwv03.dll and cppwov3.dll.

Be sure to read “Resetting Your Application's Main Resource Library” on page 270 for more information before compiling your application for distribution.

Enabling support for the Win32s platform requires a different set of DLLs than for the other Windows platforms. These DLLs are segregated by subdirectory; be sure to collect the appropriate set.

Your license type determines which DLLs you are authorized to redistribute. If you are not sure, check the license card that came with your copy of the product.

For more information about renaming product DLLs for redistribution, refer to the *Programming Guide*. For more information about Open Class DLL support, refer to the *Open Class Library User's Guide*.

Compiling and Linking Your Application

You can compile your application from a command prompt or from WorkFrame. To call both the compiler and linker programs, you can run the Toolkit's nmake program. Regardless of how you choose to compile and link your Visual Builder application, use the following compile and link options:

B"/DE /pmtyp:pm"	Passes the string /DE /pmtyp:pm to the linker as parameters.
C	Compiles and links.
Fb+	Generates Browser information in a file with a .pdb extension.
Ft(dir)	Generates files for template resolution and puts them in the dir directory.

Developing Applications

Gd+ or Gd-	Links to the runtime library. Use Gd+ for dynamic linking or Gd- for static linking.
Gm+	Uses the multithread libraries.
I	Searches the directory of the source file for include files; then searches paths specified in the <i>include</i> environmental variable.
Q	Displays the compiler logo when invoking the compiler.
Ti+	Generates debugger information. This is optional but recommended.
Tdp	Compiles all source files as C++ files and ensures that template functions are resolved.

If you have problems running your generated application, check first for the following situations:

- Be sure your application correctly destroys object factory instances, as follows:
 - To delete a modal window, connect the IVBFactory* part's *newEvent* feature to the variable's *deleteTarget* action. This connection must occur after the connection to the *showModally* action.
 - To delete a modeless window, no connections are necessary. Open the settings editor for the IVBFactory* part and select the **autoDelete** check box on the **General** page.
- If you mix static and dynamic linking, the resulting application might behave unpredictably. We recommend you use DLLs and link dynamically. Alternatively, compile and link one executable file statically with no DLLs. Do not mix static and dynamic libraries.
- When you disable notification on collection parts, unpredictable results occur. For example, if you use a sequence part to manage objects in a container and then disable notification in the sequence, the container is not notified of the change in the sequence and is therefore not refreshed.

For more information on compiling and linking, refer to the *Programming Guide*.

Developing Applications

Developing Applications



Chapter 17. Sharing Parts with Others

The most effective parts can be reused with little effort by others that are previously not familiar with the parts' design. This chapter describes how you can distribute parts to others for reuse in their own applications. Parts can be distributed in several ways, as follows:

- Providing part files (.vbb) for immediate use in Visual Builder. This method is preferred for distributing visual parts.
- Providing part information files (.vbe) for import into Visual Builder. This method works for almost any type of part but must be used to distribute function groups and user primitive parts.

In this chapter, the term *part consumer* refers to the recipient of the parts you distribute.

Providing part files (.vbb)

You can provide either visual or nonvisual parts in a part file, but this method lends itself more to visual parts, for the following reasons:

- In the case of visual parts, part consumers can see the parts in the Composition Editor as they would appear in a finished application.
- Part consumers can modify the parts.

If you want to distribute primitive visual or Composers parts, you must provide part information files (.vbe) instead. For more information, see "Providing Part Information Files (.vbe)" on page 284.

To provide part files, do the following:

1. Using Visual Builder, create a part file containing the parts.
2. Create and assign any icons needed for the new parts.
3. Supply the following to the part consumer:
 - A part file that contains the parts to be distributed
 - Any additional code files (.hvp or .cpv) needed to compile and use the parts
 - Documentation or installation instructions, including any information about how to add the parts to Visual Builder's parts palette

Developing Applications

To use the parts you distributed, the part consumer loads the part files and generates source code.

Providing Part Information Files (.vbe)

You can share nonvisual parts, class interface parts, or function groups through part information files. Use this method to distribute user primitive or Composers parts as well. One advantage to this method is that you can prevent the part consumer from modifying the parts. The IAddress sample part is an example of a part provided using this method.

To provide part information files, do the following:

1. Create parts using Visual Builder or your favorite editor. For dynamic linking, create a dll and import library containing the supplied parts. The header (.h or .hpp) file can use the `#pragma library` statement to specify the library to be used in the link step.
2. Create and assign any icons needed for the new parts.
3. Supply the following to the part consumer:

- The part information file that contains the parts to be distributed
- Any files (.hpp, .h, .hpv, .lib, .dll) needed to use the parts

Note: You must provide resource files (.rci) for class interface parts. If you do not provide these files, the resource compiler issues missing-file error messages. This file may contain as little as an end-of-file character, but consider adding a comment line to remind users why the file is needed.

- Documentation or installation instructions, including any information about how to add the parts to Visual Builder's parts palette

To use the parts you distributed, the part consumer imports the part information file into Visual Builder to create part files (.vbb). If you provided a DLL with the part information file, no code generation or further compilation is required.

Part 4. Extending Visual Builder Applications

This part provides the information you need to extend your applications beyond the basic functions that Visual Builder provides.

Chapter 18. Using Existing C and C++ Code with Visual Builder	287
Defining the Part Interface Using Part Information Files	287
Creating Composers and Primitive Visual Parts	290
Chapter 19. Adding Categories and Parts to the Parts Palette	293
Preparing Icons for the Parts Palette	293
Adding a Category to the Parts Palette	296
Specifying a Unique Icon for a Part You Add to the Parts Palette	298
Adding a Part to the Parts Palette	299
Deleting a Category or Part from the Parts Palette	302
Saving Parts Palette Changes	302
Chapter 20. Enabling National Language Support	303
Using Resource Files for Translation	303
Using Canvases to Adjust Size for Translated Text	305
Specifying Parts with Country-Sensitive Formatting	306
Providing Double-Byte Character Support for Asian Languages	306
Chapter 21. Using Direct-to-SOM (DTS) Objects	309
Creating and Importing the Part Information File	310
Using DTS Objects in a Visual Builder Application	312
Bypassing DTS Limitations	313
Chapter 22. Hints and Tips for Using Visual Builder	315

Extending Applications

Chapter 18. Using Existing C and C++ Code with Visual Builder

This chapter describes how you can use your existing C and C++ code in applications that you create with Visual Builder.

Defining the Part Interface Using Part Information Files

If C++ code already exists for your Visual Builder application, you can more efficiently define the part interface using part information files. This involves the following steps:

1. Determine the part's features.
2. Create a part information file using your favorite editor. This file can include information for as many parts as you need.
3. In Visual Builder, import the part.

Creating a Part Information File

To create a part information file, add information about your part's part interface to a file using your preferred editor. The following example shows how you could specify part information for the OACContractor part:

```
//VBBeginPartInfo: OACContractor,"Contractor part for OASearch sample"
//VBPParent: IStandardNotifier
//VBIncludes: "Ctrctor.hpp" _OACONTRACTOR_"iprofile.hpp","istring.hpp","iexcbase.hpp"
//VBPartDataFile: OANONVIS.VBB
//VBConstructor: OACContractor()
//VBComposerInfo: nonvisual,802,cppov33r
//VBEvent: ready, "ready", readyId
//VBAction: getContractor,
//VB: "Get contractor data from database",
//VB: OACContractor&,
//VB: OACContractor& getContractor()
//VBAction: putContractor,
//VB: "Add or update contractor data in database",
//VB: OACContractor&,
//VB: OACContractor& putContractor()
//VBAction: parseName,
//VB: "Parse user input to get contractor's name",
//VB: OACContractor&,
//VB: OACContractor& parseName(const IString& aName)
//VBAttribute: contractorID,
//VB: "Contractor's employee identifier",
//VB: IString,
//VB: IString contractorID() const,,
//VB: contractorIDId
//VBAttribute: lastName,
//VB: "Contractor's last name",
//VB: IString,
//VB: IString lastName() const,
```

Extending Applications

```
//VB:      OACContractor& setLastName(const IString& aLastName),
//VB:      lastNameId
//VBAttribute: firstName,
//VB:      "Contractor's first name",
//VB:      IString,
//VB:      IString firstName() const,
//VB:      OACContractor& setFirstName(const IString& aFirstName),
//VB:      firstNameId
//VBAttribute: middleInitial,
//VB:      "Contractor's middle initial",
//VB:      IString,
//VB:      IString middleInitial() const,
//VB:      OACContractor& setMiddleInitial(const IString& aMiddleInitial),
//VB:      middleInitialId
//VBAttribute: homeStreet,
//VB:      "Contractor's home street address",
//VB:      IString,
//VB:      IString homeStreet() const,
//VB:      OACContractor& setHomeStreet(const IString& aHomeStreet),
//VB:      homeStreetId
//VBAttribute: homeCity,
//VB:      "Contractor's home city",
//VB:      IString,
//VB:      IString homeCity() const,
//VB:      OACContractor& setHomeCity(const IString& aHomeCity),
//VB:      homeCityId
//VBAttribute: homeState,
//VB:      "Contractor's home state or province",
//VB:      IString,
//VB:      IString homeState() const,
//VB:      OACContractor& setHomeState(const IString& aHomeState),
//VB:      homeStateId
//VBAttribute: homeZip,
//VB:      "Contractor's home postal code",
//VB:      IString,
//VB:      IString homeZip() const,
//VB:      OACContractor& setHomeZip(const IString& aHomeZip),
//VB:      homeZipId
//VBAttribute: phoneNumber,
//VB:      "Contractor's daytime phone number",
//VB:      IString,
//VB:      IString phoneNumber() const,
//VB:      OACContractor& setPhoneNumber(const IString& aPhoneNumber),
//VB:      phoneNumberId
//VBAttribute: startDate,
//VB:      "Contractor's starting date with OA",
//VB:      IString,
//VB:      IString startDate() const,
//VB:      OACContractor& setStartDate(const IString& aStartDate),
//VB:      startDateId
//VBAttribute: endDate,
//VB:      "Contractor's last day with OA (empty if active)",
//VB:      IString,
//VB:      IString endDate() const,
//VB:      OACContractor& setEndDate(const IString& aEndDate),
//VB:      endDateId
//VBAttribute: activeStatus,
//VB:      "Whether contractor actively seeks contract work",
//VB:      Boolean,
//VB:      Boolean isActiveStatus() const,
//VB:      OACContractor& enableActiveStatus(Boolean enable = true),
//VB:      activeStatusId
//VBAttribute: currentContract,
//VB:      "Contractor's current assignment, if any",
//VB:      IString,
//VB:      IString currentContract() const,
//VB:      OACContractor& setCurrentContract(const IString& aCurrentContract),
```

Extending Applications

```
//VB:          currentContractId
//VBPreferredFeatures: enabledForNotification, getContractor, putContractor, this
//VBEndPartInfo: OACContractor
```

Note the following syntax:

- The `VBBeginPartInfo` and `VBEndPartInfo` statements delimit the part information for `OACContractor`.
- The `VBParent` statement specifies the base class for `OACContractor`, `IStandardNotifier*`.
- The `VBIncludes` statement specifies a header file to be added in an `#include` directive when the code is generated.
- The `VBPartDataFile` statement specifies the `.vbb` file that holds the information for `OACContractor`.
- The `VBComposerInfo` statement indicates that this is a nonvisual part. The absence of the abstract keyword indicates that this is a concrete part that can be dropped on the free-form surface.
- The `VBEvent`, `VBAction`, and `VBAttribute` statements define features for this part.

If you plan to compile any parts separately into a DLL, use the `VBLibFile` statement in the part information files for those parts to specify the name of the `.lib` file that links to your DLL. As an alternative, you can specify this file name in the Class Editor after you import the parts.

For information about part definition syntax, refer to *Building VisualAge for C++ Parts for Fun and Profit*.

Importing the Part

Before importing the part, you must create a part information file. To import the part, follow these steps from the Visual Builder window:

1. From the menu bar, select **File**. Select **Import part information**.
The Enter Name for Part Information File window appears.
2. Specify the path and name of the part information file that contains the information that you want to import. When the import is finished, the name of the part appears in the Visual Builder window.

If C++ code for your part already exists, your part is finished. If you want to change the part interface later, do either of the following:

- Use the Part Interface Editor to edit feature specifications. You must use this method to delete features.

Extending Applications

- Edit your part information file and re-import the part information.

If C++ code for your part does not exist, see “Adding Code to Your Part” on page 105.

Creating Composers and Primitive Visual Parts

If the Composers or primitive visual parts shipped with Visual Builder do not meet your requirements, you can create your own and drop them on the free-form surface. When dropped, user primitive parts appear as gray boxes on the free-form surface so you can adjust their placement and attribute values, but they do not otherwise behave like Visual Builder parts. Visual Builder uses an ICanvas* part to represent user Composers parts on the free-form surface. The settings window for a user part looks like that for the part’s base class, except that an extra page appears in the notebook for attributes added in the newly derived part.

In the compiled application, your user primitive parts replace the gray boxes.

To create your own primitive visual or Composers part, follow these steps:

- Write code that derives the part from an IWindow-based class. IWindow does not have to be the new part’s immediate base class.

Because user Composers parts are represented by ICanvas* parts on the free-form surface, user code to add primitive parts to the new Composers part must be of the same form as that for ICanvas*. No function exists within the Composition Editor to call customized code in the new Composers part.

- Create a part information file for the part. You must include a VBComposerInfo statement.

If the part’s constructor does not take a style flag, you must add VBConstructor and VBAttribute statements to prevent problems with the part at run time.

- Import the part.

For more information on part syntax, see *Building VisualAge for C++ Parts for Fun and Profit*.

Extending Applications

Extending Applications

Chapter 19. Adding Categories and Parts to the Parts Palette

parts palette. You can modify the parts palette at any time and from any of the Visual Builder editors or from the Visual Builder window.

One reason to modify the parts palette is so that you can quickly and easily place parts that you have created and that you use often on the free-form surface. Otherwise, you have to place them by selecting **Options**→**Add part**, which requires you to know the exact class name of the part that you are placing.


Another reason to modify the parts palette is to give everyone who is working on the same project access to the same set of standardized parts. Your company could have a parts builder who builds these standardized parts and puts them in a category on the parts palette for you to use.

Group parts that behave similarly in the same category. By looking at the parts palette you can see how we grouped the parts that we provided into categories based on their behavior. For example, all parts that are used for data entry are in one category, all parts that contain and display lists are in another category, and so forth.

This chapter contains an example that uses the **OACContractor** nonvisual part created in Chapter 7, “Creating Nonvisual Parts” on page 101, but you can use any part that you have created. When you finish the example, you have a new **OAModels** category and a new **OACContractor** part on the parts palette.

Preparing Icons for the Parts Palette

Each category and part on the Visual Builder parts palette is represented by a bitmap so you can recognize it visually. With Visual Builder, you can create and use your own bitmaps when you extend the parts palette. If you do not, you can still extend



the parts palette and accept the default category icon,  , and the default part

icon,  .

Extending Applications

Preparing a Resource DLL (OS/2 Version Only)



This example uses  for the **OAModels** category and  for the **OAContractor** part, which are stored in the `cppwv33r.dll` file as resource numbers 800 and 802, respectively.



To prepare bitmaps for use with Visual Builder, do the following:

1. Create your icons. One way to do this is to use the OS/2 icon editor, which is available in the operating system toolkit.

Bitmaps used on the parts palette must be no larger than standard icons for the display resolution being used. For VGA displays on OS/2, use the Independent VGA form (32x32). For higher display resolutions on OS/2, use the 8514-16 colors form (40x40).

2. Create a resource DLL that contains your icons. Use files similar to the following:

- `userpal.c`

```
empty()
{
}
```
- `userpal.rc`

```
icon 800 oamodels.ico
icon 801 oacontractor.ico
```
- `userpal.def`

```
library userpal
description 'Icons for user-extended palette'
```
- `userpal.mak`

```
userpal.dll: userpal.obj userpal.def userpal.res
icc userpal.obj /Feuserpal.dll userpal.def
rc userpal.res userpal.dll

userpal.obj: userpal.c
icc /C+ userpal.c

userpal.res: userpal.rc
rc -r userpal.rc
```

Once you have the files ready, type the following in a command window to build the resource DLL:

```
nmake userpal.mak
```



Extending Applications

3. Place the resource DLL in a directory in your LIBPATH statement.

Your icons are now ready for use with Visual Builder.

Preparing a Resource DLL (Windows Version Only)



This example uses  for the **OAModels** category and  for the **OAContractor** part, which are stored in the `cppwv33r.dll` file as resource numbers 800 and 802, respectively.

To prepare bitmaps for use with Visual Builder, do the following:

1. Create your icons using an icon editor.
2. Create a resource DLL that contains your icons. Use files similar to the following:
 - `cppwv33r.c`

```
empty()
{
}
```
 - `cppwv33r.rc`

```
800 icon    oamodels.ico
801 icon    contract.ico
802 icon    contrctr.ico
803 icon    diploma.ico
804 bitmap  oalog2.bmp
805 bitmap  oalog3.bmp
```
 - `cppwv33r.mak`

Extending Applications

```
cppwv33r.dll: cppwv33r.obj cppwv33r.lib cppwv33r.res
    icc /Gd- /Ge- /I. /Tdc /B"/DLL /NOE" -Fm$(@B).map /Fecppwv33r.dll \
        cppwv33r.exp \
        cppwv33r.obj \
        cppwv33r.res

cppwv33r.lib: cppwv33r.obj cppwv33r.res
    copy << cppwv33r.def
    LIBRARY cppwv33r
    DESCRIPTION 'Icons and bitmaps for OASearch sample application'
    EXPORTS
<<
    cppfilt -b -p $** >> cppwv33r.def
    set ILIB=
    ilib /Q cppwv33r.def

cppwv33r.obj: cppwv33r.c
    icc /Gd- /Ge- /I. /Tdc -c -O+ cppwv33r.c

cppwv33r.res: cppwv33r.rc
    irc cppwv33r.rc
```

Once you have the files ready, type the following in a command window to build the resource DLL:

```
nmake cppwv33r.mak
```

3. Place the resource DLL in a directory in your PATH statement.

Your icons are now ready for use with Visual Builder.

Adding a Category to the Parts Palette

Once you have prepared an icon in a resource DLL, you are ready to extend the parts palette. To add a category to the parts palette, do the following:

1. In the Composition Editor, select **Modify palette.**→**Add new category.** from the **Options** pull-down menu. The Add Palette Category window is displayed as follows:

Extending Applications

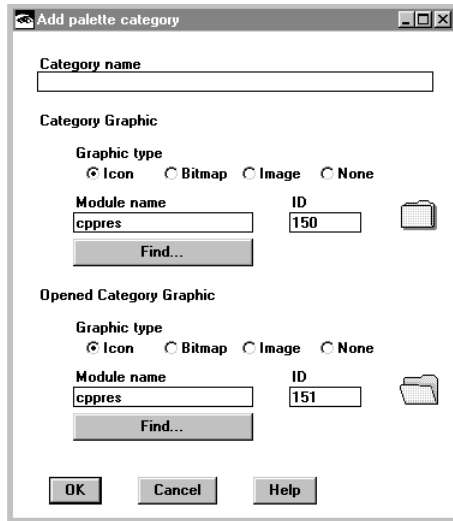



Figure 118. The Add Palette Category Window

Notice that the default category icon, , is specified. It is stored as resource ID 150 in the cppwv33r.dll resource file provided with Visual Builder.

2. Enter the name that you want for your category in the **Category name** field.
3. Enter cppov33r for OS/2 or cppwv33r for Windows or the name of your resource DLL in the **Module name** field.

Note: Do not type the .dll file extension in the **Module name** field.

4. Type 800 or the resource ID of the icon in your resource DLL in the **Resource ID** field.


After entering the resource ID number, move the cursor to another component in the window, such as the **Module name** field, if you want to see the graphic that will be used before continuing.

5. Select the **OK** push button.

Your category with the icon specified is added to the parts palette.

Extending Applications

Note: If you do not specify a DLL, Visual Builder uses the default icon. If you specify a DLL but Visual Builder cannot find it, Visual Builder uses the

question mark icon,  .

If the question mark icon appears, make sure the following conditions are met:

- The DLL exists and is in the current directory.
- The DLL file name is correct.
- The resource ID for the icon (in the .rc file) exists in the DLL.


Specifying a Unique Icon for a Part You Add to the Parts Palette

You can specify a unique icon for a part that you add to the parts palette, but you must do so before you add it to the parts palette. To give your part a unique icon, do the following:

1. Open the part.
2. Switch to the Class Editor.
3. Enter the name of the DLL file that contains the icon you want to use in the **DLL Name** field.
4. Enter the resource ID number for the icon in the **Resource ID** field.

If you enter a valid DLL file name and resource ID number, Visual Builder displays the icon below the **Resource ID** field. This enables you to verify the icon before adding it to the parts palette.

Note: If you do not specify a DLL, Visual Builder uses the default icon. If you specify a DLL but Visual Builder cannot find it, Visual Builder uses the

question mark icon,  .

If the question mark icon appears, make sure the following conditions are met:

- The DLL exists and is in the current directory.
- The DLL file name is correct.
- The resource ID for the icon (in the .rc file) exists in the DLL.

5. Select **File**→**Save** to save the resource DLL and resource ID information in the Class Editor.

Adding a Part to the Parts Palette

You can add a part to any category on the parts palette using any of the following methods:

- Add a part that is selected in the Visual Builder window
- Add the part that you are currently editing
- Add any part whose .vbb file is loaded

Adding a Part That Is Selected in the Visual Builder Window

To add a part to the parts palette from the Visual Builder window, do the following:

1. Load the .vbb file that contains the part you want to add to the parts palette if it is not already loaded.
2. Select the .vbb file. For this example, select `oanonvis.vbb`.
3. Select the part you want to add. For this example, select **OACContractor**.

Note: You can add multiple parts by holding down the Ctrl key and clicking on each part that you want to add.

4. Select **Part→Add to palette**. Visual Builder displays the Add to Palette window, as shown in Figure 119.

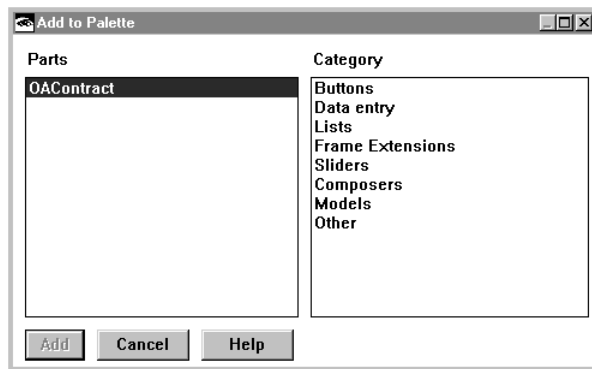


Figure 119. Add to Palette Window

5. Select the part that you want to add. For this example, select **OACContractor**.
6. Select the category that you want to add the part to. For this example, select **OAModels**.
7. Select the **Add** push button. Visual Builder adds the **OACContractor** part to the parts palette in the **OAModels** category.

Extending Applications

Adding the Part That You are Currently Editing

You can add the part that you are currently editing to the parts palette from either the Composition Editor, the Class Editor, or the Part Interface Editor. To add the part that you are currently editing to the parts palette, do the following:

1. Double-click on the **OACContractor** part in the Visual Builder window. Visual Builder opens the **OACContractor** part in the Part Interface Editor.
2. Select **File→Add to palette**. Visual Builder displays the Add to Palette window, as shown in Figure 120.

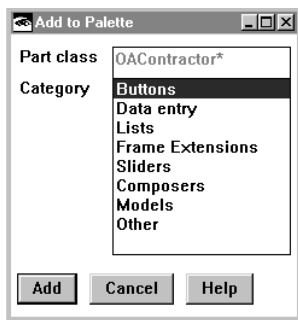


Figure 120. The Add to Palette Window

The **Part name** field shows the name of the part that you are editing. This is the part that is to be added to the parts palette. You cannot change the name of the part displayed in this field. In this example, **OACContractor** is displayed in this field.

3. Select the category that you want to add the part to. For this example, select **OAModels**.
4. Select the **Add** push button. Visual Builder adds the **OACContractor** part to the **OAModels** category on the parts palette. To see this, switch to the Composition Editor and select the **OAModels** category. The icon for the **OACContractor** part is displayed in the parts column.

Adding Any Part Whose .vbb File Is Loaded

You can add any part to the parts palette as long as its .vbb file is loaded in the Visual Builder window. The following steps explain how to do this:

1. In the Composition Editor, select **Modify palette**→**Add new part** from the **Options** pull-down menu. The Add to Palette window is displayed as shown in Figure 121.

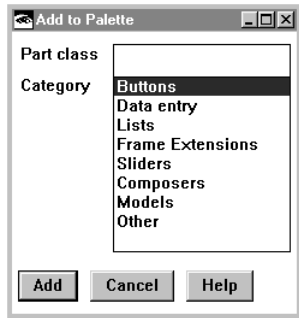



Figure 121. The Add to Palette Window

To add a part to the parts palette, do the following:

- a. Type `OACContractor` in the **Part name** field or the class name of the part you want to add.
- b. Select `OAModels` in the **Category** list or the name of the category to which you want to add your part.
- c. Select the **Add** push button.

Your part is added to the parts palette in the specified category.

Notice that the part you just added uses the same icon as the part it inherits from. If you inherit from a part whose .vbb file is not loaded or for which you have not

provided a resource DLL, Visual Builder uses the default part icon, .

Extending Applications

Deleting a Category or Part from the Parts Palette

To delete a part from the parts palette, do the following:

1. Select the part on the parts palette.
2. Select **Modify palette→Delete category** from the **Options** pull-down menu. The selected part is deleted from the parts palette.

To delete a category from the parts palette, do the following:

1. Select the category on the parts palette.
2. Select **Modify palette→Delete category** from the **Options** pull-down menu. The selected category and all of the parts in it are deleted from the parts palette.

Saving Parts Palette Changes

Visual Builder automatically saves all parts palette changes for you. When you create a new category or part, Visual Builder stores information about that category or part in a file named vbpalet.dat, which is stored in your startup directory (or in your target directory, if you are using WorkFrame). This file is written automatically.

Once you add or delete categories or parts, the vbpalet.dat file is read each time you start Visual Builder. The information this file contains causes any categories or parts that you have added to be included on the parts palette. It also prevents any categories or parts that you have deleted from appearing on the parts palette.

If you update the icon associated with a part, the parts palette is updated the next time you select the category in which the icon appears.

Removing a category or part that you just added

The vbpalet.dat file also allows you to undo and redo any changes you make to the parts palette, but only during the current Composition Editor session. For example, after adding a category or part, you can select **Edit→Undo** to remove the part or category you just added. Selecting **Edit→Redo** would put the part or category back on the parts palette, again.

Once you close the Composition Editor, you can no longer undo or redo any changes. However, you can still add categories and add parts, as well as delete categories and parts.



Chapter 20. Enabling National Language Support

Visual Builder provides for ease of national language support in the following ways:

- Using resource files for translation
- Using canvases to adjust the size for translated text
- Specifying data types with country-sensitive formatting
- Providing double-byte character set (DBCS) support for Asian languages



A good rule to follow: Do not place any translatable text or country symbols directly into your source code. Place them in an external file where they are easily accessible for translation.

For additional information about national language and DBCS support, refer to the *IBM Open Class Library User's Guide*.

Using Resource Files for Translation

In Chapter 16, “Generating Source Code for Parts and Applications” on page 273, you learned that Visual Builder generates the following resource files for you:

- A resource file (.rci), which contains the text strings and other program resources used in your part
- A resource header file (.h), which contains the resource ID definitions for your application

In order for Visual Builder to write most resource definitions to the .h and .rci files, you must specify a starting resource ID in the Class Editor for every part. For information about how to specify a starting resource ID, see “Specifying a Starting Resource ID” on page 53. For information about how to choose a starting resource ID, see “Guidelines for Specifying Starting Resource IDs” on page 305.

The Resource File (.rci)

The .rci file groups resources into two categories: window resources and nonwindow resources. *Window resources* are those that are associated with window IDs, such as information area text, fly-over text, and help tables.

Nonwindow resources are the text strings that are displayed in your composite part. Examples are window titles, static text used to label entry fields and list boxes, and the text on push buttons and menu items. These text strings are delimited by quotation marks (" ") and can be translated into another language.

Extending Applications

The window resources and nonwindow resources are grouped into separate string tables so that translators can easily find the text that is to be translated.

You may have text strings in your part, such as the application name, that you do not want translated. If that is the case, you can prevent those text strings from being inserted in the .rci file by inserting a number sign (#) at the beginning of the text and enclosing the text in quotation marks (" "). This change must be made in the settings window for the part, not in the Composition Editor.

For example, suppose you do not want the text in the window title to be translated. To prevent Visual Builder from inserting this text string in the .rci file, open the settings window for the part and edit the entry field on the **General** page that contains the title text. In the case of the ToDoList part, the modified title would appear as follows:

```
#"To-Do List"
```

Each text string that Visual Builder inserts in the .rci file is preceded by a resource name that begins with STRRC, as follows:

```
STRRC_ToDoList_FrameWindow_title, "To-Do List"
```

Visual Builder defines numeric resource IDs for these resource names in the .h file.

For Visual Builder to define most string resources properly, you must specify the starting resource ID for your part in the Class Editor.

The Resource Header File (.h)

In the .h file, Visual Builder uses #define statements to assign unique resource IDs to each of the text strings listed in the .rci file. Visual Builder also assigns unique window IDs to all primitive visual parts. The only resource ID that you need to specify is the starting resource ID for the part.

For an example, look at the todolist.h file that Visual Builder generates for the To-Do List application shown in Chapter 2, “Creating a Simple Visual Builder Application” on page 7. The first #define statement in the todolist.h file appears as follows:

```
#define RC_ToDoList 10000
```

The number in this #define statement, 10000, is the starting resource ID. When you select the check box next to the **Starting resource id** field in the Class Editor for this part, the number 10000 appears in the entry field. During code generation, Visual Builder uses this number as the resource ID of the first text string and increments the resource ID of each successive text string by 1.

Guidelines for Specifying Starting Resource IDs

You must specify a starting resource ID in all parts for which you want Visual Builder to generate resource files. You can either use the default or specify an alternative, but consider the following:

- The resource ID must be a number.
- The number specified must be either high enough or low enough that the resource IDs produced do not conflict with the resource IDs that Visual Builder generates for other parts that comprise your application.

For example, suppose you have a reusable Address part (a canvas with entry fields and static text) that you want to embed as a subpart in the frame window of your application's main view. You might give the main view a starting resource ID of 5000 and the Address part a starting resource ID of 6000. Doing this would prevent conflicts with the resource IDs that Visual Builder generates for the main view and those it generates for the Address part.

Consider using starting resource IDs between 100 and 14500 for most applications, for the following reasons:

- The operating system has reserved many resource IDs below 100 for its own use.
- When determining resource IDs for window resources, Visual Builder begins with 15000 and increments the resource ID of each successive primitive part by 5. Starting resource IDs between 100 and 14500 are low enough to prevent you from experiencing any resource ID conflicts in most cases.

For more information about how to specify starting resource IDs, see “Specifying a Starting Resource ID” on page 53.

Using Canvases to Adjust Size for Translated Text

When text is translated from one language to another, the translated text often occupies more space than the original text occupied. This can cause problems because the layout of the user interface can be disrupted by the longer text strings.

Two of the canvas parts that Visual Builder provides solve this problem for you: `ISetCanvas*` and `IMultiCellCanvas*`. These parts allow you to insert translated text and rebuild your application without having to change the position of any of the parts in the user interface. The `ISetCanvas*` and `IMultiCellCanvas*` parts automatically adjust their size at run time to allow for longer text strings, taking into account the current window's text size and font.

For example, suppose you are using an `ISetCanvas*` part with three vertical decks and three rows of `IRadioButton*` parts in each deck. Once the text strings for the

Extending Applications

IRadioButton* parts are translated, all you have to do is rebuild your application. The decks in the ISetCanvas* part automatically adjust their widths to allow for the size of the translated text strings.

The IMultiCellCanvas part is also good for translation purposes because the rows and columns automatically adjust themselves to fit the translated text.

Specifying Parts with Country-Sensitive Formatting

Visual Builder provides the following class interface parts that allow you to specify how information is presented for specific national languages:

- | | |
|--------------|--|
| IDate | This part allows you to customize the date formatting for the selected part. This includes specifying the order for the month, day, and year, and the character to use as a separator. |
| ITime | This part allows you to customize the time formatting for the selected part. This includes specifying the 12- or 24-hour format and the character to use as a separator. |

Providing Double-Byte Character Support for Asian Languages

The following Visual Builder parts provide DBCS support:

IBuffer

A class interface part that defines the contents of an IString. This part provides attributes to determine whether part or all of the characters in a buffer are DBCS or multibyte character set (MBCS) characters, and whether they are valid DBCS or MBCS characters.

IDBCSBuffer

A class interface part that implements the version of IString contents that supports mixed DBCS characters. This part ensures that MBCS characters are processed properly.

IEntryField

A visual part that creates and manages an entry field control. On the **General** page of the settings window for this part, you can specify the type of data that the user can enter. It can be one of the following:

- | | |
|--------------|---|
| SBCS | Sets the entry field to accept SBCS text only. |
| DBCS | Sets the entry field to accept DBCS text only. |
| Mixed | Sets the entry field to accept text that is a mixture of SBCS and DBCS characters. Conversion from an ASCII DBCS code page to an EBCDIC DBCS code page can result in a possible increase in the |

Extending Applications

length of the data because of the addition of shift-in and shift-out characters, but it does not exceed the text limit of the entry field.

Any Sets the entry field to accept text that is a mixture of SBCS and DBCS characters. This setting is the opposite of mixed. If the text contains both SBCS and DBCS characters and is to be converted from an ASCII code page into an EBCDIC code page, this style causes an entry field to properly handle shift-in and shift-out characters that would otherwise be introduced into its text.

IFrameWindow

A visual part that creates and manages a frame window control. This part has a style option, `appDBCSStatus`, that includes a DBCS status area in the frame window when it is displayed in a DBCS environment. `IFrameWindow` also has a member function, `shareParentDBCSStatus`, that causes a child frame window to share the DBCS status area of its parent.

IKeyboardEvent

A class interface part that represents a keyboard-related event. An `IKeyboardEvent` object is created by a keyboard handler when a user presses or releases a key. The part provides a *virtualKey* attribute that returns the virtual key code of the key. Two of the codes it can return are `firstDBCS` and `lastDBCS`.

IString

A class interface part that is an array of characters. This part provides attributes to determine whether part or all of the characters in a string are DBCS or MBCS characters, and whether they are valid DBCS or MBCS characters.

Extending Applications



Chapter 21. Using Direct-to-SOM (DTS) Objects

This chapter shows you how to use a Direct-to-SOM object in a Visual Builder application. *SOM* is the IBM System Object Model, which defines an interface between programs, or between libraries and programs, so that an object's interface is separated from its implementation. *SOM* allows classes of objects to be defined in one programming language and used in another, and it allows libraries of such classes to be updated without requiring client code to be recompiled.

You can make classes and member functions in existing C++ programs *SOM*-accessible without having to rewrite class and member function definitions. You do this by adding *SOM* C++ compiler directives to your code. Although *SOM* imposes some restrictions on C++ coding conventions, you should be able to convert most C++ programs for *SOM* support with minimal effort.

VisualAge for C++ can convert existing C++ classes to *SOM* classes. The VisualAge for C++ compiler translates C++ code into Interface Definition Language (IDL) code, which you then compile with the *SOM* compiler to create your *SOM* objects. This method of creating *SOM* objects is referred to as the C++ Direct-to-*SOM*, or DTS, method. Currently, the only *SOM* objects that Visual Builder supports are DTS objects.

To use a DTS object in a Visual Builder application, you must do the following:

1. Create the DTS object or objects.

You will need the .hh and .lib files that you created in this step when compiling your Visual Builder application. You need the .dll when you run your compiled Visual Builder application.

The DTS objects that you create must have the following characteristics:

- DTS objects can have actions only; no attributes or events are allowed. DTS objects are not able to notify other parts when events occur. Therefore, no events or attribute event identifiers can be used.
- You can implement actions using C++ and classes in the IBM Open Class Library. However, the actions can return only basic C data types and their parameters must be basic C data types. For example, instead of returning an *IStrng* data type, an action could return a *char**.

To learn how Visual Builder can help you avoid the DTS restrictions stated in the preceding list, see "Bypassing DTS Limitations" on page 313.

2. Create and import a part information file (.vbe file) for your DTS objects.

Extending Applications

3. Create a Visual Builder application using DTS objects.
4. Generate and compile the code.
5. Run your application.

Make sure you have the .dll for the DTS object for this step.

The sections that follow assume that you have already created your DTS object or objects, so they only discuss creating and importing the part information file, and creating a Visual Builder application using DTS objects.

For information on creating DTS objects, refer to the *VisualAge for C++ User's Guide*. For information on generating and compiling your Visual Builder application code, see Chapter 16, "Generating Source Code for Parts and Applications" on page 273.

The vbsom.vbe sample contains Direct-to-SOM parts from the SOMObjects Toolkit that you can import into Visual Builder. Also, the vbcc.vbe sample contains collection class parts for you to use. You can find these files in the \ibmcppw\samples\visbuild\vbsample directory.

We begin with creating and importing the part information file.

Creating and Importing the Part Information File

"Defining the Part Interface Using Part Information Files" on page 287 discusses how to create a part information file. Create one for your DTS objects. The following guidelines apply:

- DTS objects can only have actions. This makes DTS objects the equivalent of class interface parts because they have no notification capability. Therefore, set the VBComposerInfo statement for each DTS object to *class*.
- Do not code any attributes or events in your part information file.

The following example shows a part information file, mydtsdt.vbe, that contains the information for the following sample DTS objects:

MyDTSDate Returns the current date.

MyDTSTime Returns the current time.

Extending Applications

```
//  
// SOM classes as Visual Builder classes  
//  
//VBBeginPartInfo: MyDTSDate, "My DTS SOM Date Class"  
//VBIncludes: "mydtsdat.hh" MyDTSDate_hh  
//VBPpartDataFile: 'mydtsdt.vbb'  
//VBLibFile: 'mydtsdat.lib'  
//VBComposerInfo: class  
//VBAction: getTodaysDate  
//VB:      , "Get today's date action.", char*,  
//VB:      char* getTodaysDate()  
//VBPreferredFeatures: this, getTodaysDate  
//VPEndPartInfo: MyDTSDate  
//  
//VBBeginPartInfo: MyDTSTime, "My DTS SOM Time Class"  
//VBIncludes: "mydtstim.hh" MyDTSTime_hh  
//VBPpartDataFile: 'mydtsdt.vbb'  
//VBLibFile: 'mydtstim.lib'  
//VBComposerInfo: class  
//VBAction: getCurrentTime  
//VB:      , "Get current time action.", char*,  
//VB:      char* getCurrentTime()  
//VBPreferredFeatures: this, getCurrentTime  
//VPEndPartInfo: MyDTSTime  
//
```

The mydtsdt.vbe file contains statements that refer to the following files that are needed for each of the sample DTS objects:

mydtsdat.hh and mydtstim.hh

The header files for the MyDTSDate and MyDTSTime SOM classes.

mydtsdat.lib and mydtstim.lib

The library files that were created when the DTS objects were compiled.

mydtsdt.vbb

The part (.vbb) file that is to contain the information about the DTS parts when you import the part information from the mydtsdt.vbe file. You must load the mydtsdt.vbb file into Visual Builder before you can add the MyDTSDate and MyDTSTime parts to the free-form surface in the Composition Editor.

Once you create your part information file, you must import it into Visual Builder before you can use your DTS objects in a Visual Builder application. For information on how to do this, see “Importing Part Information” on page 114.

The next step is using DTS objects in a Visual Builder application.

Extending Applications

Using DTS Objects in a Visual Builder Application

Using DTS objects in a Visual Builder application is no different from using a class interface part. You simply place the DTS objects on the free-form surface and make the necessary connections to use their actions.

The following figure shows how we used the MyDTSTime and MyDTSDate objects in a simple application.

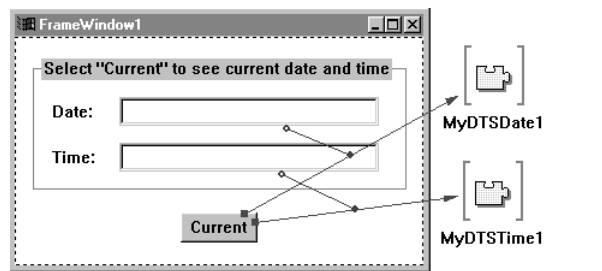


Figure 122. Sample Application Using DTS Objects

To create the application, we placed the MyDTSDate and MyDTSTime parts on the free-form surface by selecting **Options**→**Add part** and by providing the necessary information in the Add Part window for each part.

In this application, when a user clicks on the **Current** push button, the *buttonClickEvent* feature causes the *getTodaysDate* and *getTodaysTime* actions to get the date and time that is currently set in your computer's operating system. In addition, the *actionResult* attribute of each connection updates the *text* attribute of each entry field with the result of the two actions, the current date and time.

Earlier, we mentioned that actions in DTS classes can return only basic C data types. The *getTodaysDate* and *getTodaysTime* actions in our example both return a data type of *char**. Therefore, when we connected the *actionResult* attribute to the *text* attribute of the entry field, Visual Builder displayed a message saying that the types did not match and asking if we wanted to continue. In this case, we could make the connection because *IString* has a constructor that takes a *char**. You can find out whether to complete a connection in situations like this by looking at the *IBM Open Class Library Reference*.

Extending Applications



When you have generated your code and are ready to compile your application, make sure you have the .hh and .lib files that contain the code for your DTS objects in the current directory. Also, make sure the SOM Toolkit is installed because the VisualAge for C++ compiler must have access to several of its files.

When you are ready to run your application, the .dll for the DTS objects must be accessible.

Bypassing DTS Limitations

Earlier, we told you that DTS objects have certain limitations, such as not being able to use attributes and events, and not being able to notify other parts. You can bypass these limitations by using nonvisual parts to manage the things your DTS objects cannot do.

In the example shown in “Using DTS Objects in a Visual Builder Application” on page 312, we could have created two nonvisual parts called MyDate and MyTime in addition to the two DTS parts, MyDTSDate and MyDTSTime. We could give these nonvisual parts attributes, such as *theDate* and *theTime*, and notify other parts when the values of these attributes change. We could also give these attributes get and set member functions that call actions in the DTS objects to get and set the values of the attributes.

Once this is done, we would place the nonvisual parts MyDate and MyTime on the free-form surface, instead of MyDTSDate and MyDTSTime, and make the same connections as described previously.

Extending Applications

Chapter 22. Hints and Tips for Using Visual Builder

This chapter contains hints and tips that you might find useful when using Visual Builder.

Workstation beeps

If your workstation beeps while you are running Visual Builder, a C++ exception has probably been thrown. Check for the following conditions:

- Missing resources (bitmaps or icons)
- Incorrect or incompatible part settings

No connections run in user code

You must explicitly instantiate connections and enable notification for them. Visual Builder does this in generated code by means of the `initializePart()` function. This function initializes any static subparts and all connections from the part. If you instantiate a generated part using the new operator in your own code, be sure to call the part's `initializePart()` function, as follows:

```
//
// I used Visual Builder to generate the class definition for myPart.
//
myPart* pMine = new myPart;
pMine->initializePart();
//
// Now I can do something.
//
pMine->show();
```

Duplicate notifications

When *commandEvent* is signaled in an *IFrameWindow** client, duplicate notifications can occur. For example, for a contextual pop-up menu in a container that is the client of a frame window, selecting an item from the pop-up menu signals a *commandEvent* notification in the container. The frame window responds to the notification by propagating the *commandEvent* (a duplicate) back to its client, the container.

To block the duplicate *commandEvent* notification without otherwise changing the behavior of the window at run time, add an *IMultiCellCanvas** part between the container and the frame window part. The *IMultiCellCanvas** part becomes the client of the frame window. In this situation, the second notification is passed to the *IMultiCellCanvas** part, not to the container.

Offset calculation for windows

If compiled application windows appear partially off the screen, be aware that Visual Builder calculates window position based on the x and y coordinates of the

Extending Applications

upper-left corner of the window. When you create the part that contains the window, Visual Builder calculates a position offset from the upper-left corner of the scrollable free-form surface to the upper-left corner of the window. This does not usually present a problem because most parts are built from the upper-left corner of the free-form surface.

If you have added enough other parts to increase the size of the scrollable free-form surface, the offset might become large enough to push a compiled window subpart off the edge of the desktop display. To prevent this, place the window subpart immediately to the right of the primary window part. Place variable or nonvisual parts at the extreme right of the free-form surface.

Cannot enter text into entry fields on Windows

The default font for Windows is taller than that for OS/2. This size difference can cause problems if you drop IEntryField* parts on an ICanvas* part in OS/2 and then try to use the composite part in Windows.

The height of an entry field is calculated in OS/2 based on the size of the OS/2 font; the size of the entry field is then fixed. Compiled in Windows, this entry field cannot accommodate the taller Windows font.

To prevent this problem in OS/2, consider using IMultiCellCanvas* parts instead of ICanvas* parts. If you experience this problem but want to continue using the ICanvas* part, reset all IEntryField* parts in Windows to their default size from the parts' contextual menu.

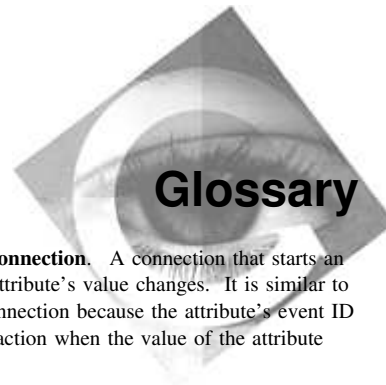
A similar problem occurs if you create a composite part in Windows NT or Windows 95 and try to run the application in Win32s.

Stop after exception

To stop the calling of subsequent connections when a previous connection throws an exception, group all your connections into one custom logic connection. All the code that you put within a custom logic connection gets called within one try-catch block. When an exception is thrown, no subsequent code is called.

Single-byte names

You must enter part names and other C++ keywords in single-byte mode. If you enter them in double-byte mode, Visual Builder generates the code without errors, but the code does not compile correctly.



This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York:McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. A class that provides common behavior across a set of subclasses but is not itself designed to have instances that work. An abstract class represents a concept; classes derived from it represent implementations of the concept. For example, *IControl* is the abstract base class for control view windows; the *ICanvas* and *IListBox* classes are controls derived from *IControl*. An abstract class must have at least one pure virtual function.

See also *base class*.

access. A property of a class that determines whether a class member is accessible in an expression or declaration.

action. A specification of a function that a part can perform. The visual builder uses action specifications to generate connections between parts. Actions are resolved to member function calls in the generated code.

Compare to *event* and *attribute*.

argument. A data element, or value, included as part of a member function call. Arguments provide additional information that the called member function can use to perform the requested operation.

attribute. A specification of a property of a part. For example, a customer part could have a name attribute and an address attribute. An attribute can itself be a part with its own behavior and attributes.

The visual builder uses attribute specifications to generate code to get and set part properties.

Compare to *event* and *action*.

attribute-to-action connection. A connection that starts an action whenever an attribute's value changes. It is similar to an event-to-action connection because the attribute's event ID is used to notify the action when the value of the attribute changes.

See also *connection*. Compare to *event-to-action connection*.

attribute-to-attribute connection. A connection from an attribute of one part to an attribute of another part. When one attribute is updated, the other attribute is updated automatically.

See also *connection*.

attribute-to-member function connection. A connection from an attribute of a part to a member function. The connected attribute receives its value from the member function, which can make calculations based on the values of other parts.

See also *connection*.

B

base class. A class from which other classes or parts are derived. A base class may itself be derived from another base class.

See also *abstract class*.

behavior. The set of external characteristics that an object exhibits.

C

caller. An object that sends a member function call to another object.

Contrast with *receiver*.

category. In the Composition Editor, a selectable grouping of parts represented by an icon in the left-most column. Selecting a category displays the parts belonging to that category in the next column.

See also *parts palette*.

class. An aggregate that can contain functions, types, and user-defined operators, in addition to data. Classes can be defined hierarchically, allowing one class to be an expansion of another, and can restrict access to its members.

Class Editor •derivation

Class Editor. The editor you use to specify the names of files that Visual Builder writes to when you generate default code. You can also use this editor to do the following:

- Enter a description of the part
- Specify a different .vbb file in which to store the part
- See the name of the part's base class
- Modify the part's default constructor
- Enter additional constructor and destructor code
- Specify a .lib file for the part
- Specify a resource DLL and ID to assign an icon to the part
- Specify other files that you want to include when you build your application

Compare to *Composition Editor* and *Part Interface Editor*.

class hierarchy. A tree-like structure showing relationships among object classes. It places one abstract class at the top (a base class) and one or more layers of less abstract classes below it.

class library. A collection of classes.

class member function. See *member function*.

client area object. An intermediate window between a frame window (IFrameWindow) and its controls and other child windows.

client object. An object that requests services from other objects.

collection. A set of features in which each feature is an object.

Common User Access (CUA). An IBM architecture for designing graphical user interfaces using a set of standard components and terminology.

composite part. A part that is composed of a part and one or more subparts. A composite part can contain visual parts, nonvisual parts, or both.

See also *nonvisual part*, *part*, *subpart*, and *visual part*.

Composition Editor. A view that is used to build a graphical user interface and to make connections between parts.

Compare to *Class Editor* and *Part Interface Editor*.

concrete class. A subclass of an abstract class that is a specialization of the abstract class.

connection. A formal, explicit relationship between parts. Making connections is the basic technique for building any

visual application because that defines the way in which parts communicate with one another. The visual builder generates the code that then implements these connections.

See also *attribute-to-action connection*, *attribute-to-attribute connection*, *attribute-to-member function connection*, *parameter connection*, *custom logic connection*, *event-to-action connection*, *event-to-attribute connection*, and *event-to-member function connection*.

const. An attribute of a data object that declares that the object cannot be changed.

construction from parts. A software development technology in which applications are assembled from existing and reusable software components, known as parts.

constructor. A special class member function that has the same name as the class and is used to construct and possibly initialize class objects.

CUA. See *Common User Access*.

cursor emphasis. When the selection cursor is on a choice, that choice has cursor emphasis.

custom logic connection. A connection that causes your customized C or C++ code to be run. This connection can be triggered either when an attribute's value changes or an event occurs.

D

data abstraction. A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

data member. Private data that belongs to a given object and is hidden from direct access by all other objects. Data members can only be accessed by the member functions of the defining class and its subclasses.

data model. A combination of the base classes and parts shipped with the product and the classes and parts you save and create. They are saved in a file named vbbase.vbb.

data object. A storage area used to hold a value.

declaration. A description that makes an external object or function available to a function or a block.

DEF file. See *module definition file*.

derivation. The creation of a new or abstract class from an existing or base class.

destructor •loaded

destructor. A special class member function that has the same name as the class and is used to destruct class objects.

DLL. See *dynamic link library*.

dynamic link library (DLL). In OS/2, a library containing data and code objects that can be used by programs or applications during loading or at run time. Although they are not part of the program's executable (.exe) file, they are sometimes required for an .exe file to run properly.

E

encapsulation. The hiding of a software object's internal representation. The object provides an interface that queries and manipulates the data without exposing its underlying structure.

event. A specification of a notification from a part.

Compare to *action*, *attribute*, and *part event*.

event-to-action connection. A connection that causes an action to be performed when an event occurs.

See also *connection*.

event-to-attribute connection. A connection that changes the value of an attribute when a certain event occurs.

See also *connection*.

event-to-member function connection. A connection from an event of a part to a member function. When the connected event occurs, the member function is executed.

See also *connection*.

expansion area. The section of a multicell canvas between the current cell grid and the outer edge of the canvas. Visually, this area is bounded by the rightmost column gridline and the bottommost row gridline.

F

feature. (1) A major component of a software product that can be installed separately. (2) In Visual Builder, an action, attribute, or event that is available from a part's part interface and that other parts can connect to.

full attribute. An attribute that has all of the behaviors and characteristics that an attribute can have: a data member, a get member function, a set member function, and an event identifier.

free-form surface. The large open area of the Composition Editor window. The free-form surface holds the visual parts contained in the views you build and representations of the nonvisual parts (models) that your application includes.

G

graphical user interface (GUI). A type of interface that enables users to communicate with a program by manipulating graphical features, rather than by entering commands. Typically, a graphical user interface includes a combination of graphics, pointing devices, menu bars and other menus, overlapping windows, and icons.

GUI. See *graphical user interface*.

H

handles. Small squares that appear on the corners of a selected visual part in the visual builder. Handles are used to resize parts.

Compare to *primary selection*.

header file. A file that contains system-defined control information that precedes user data.

I

inheritance. (1) A mechanism by which an object class can use the attributes, relationships, and member functions defined in more abstract classes related to it (its base classes). (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

instance. Synonym for *object*, a particular instantiation of a data type.

L

legacy code. Existing code that a user might have. Legacy applications often have character-based, nongraphical user interfaces; usually they are written in a nonobject-oriented language, such as C or COBOL.

loaded. The state of the mouse pointer between the time you select a part from the parts palette and deposit the part on the free-form surface.

main part •object-oriented programming

M

main part. The part that users see when they start an application. This is the part from which the main() function C++ code for the application is generated.

The main part is a special kind of composite part.

See also *part* and *subpart*.

member. (1) A data object in a structure or a union. (2) In C++, classes and structures can also contain functions and types as members.

member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class.

member function call. A communication from one object to another that requests the receiving object to execute a member function.

A member function call consists of a member function name that indicates the requested member function and the arguments to be used in executing the member function. The member function call always returns some object to the requesting object as the result of performing the member function.

Synonym for *message*.

member function name. The component of a member function call that specifies the requested operation.

message. A request from one object that the receiving object implement a member function. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the member function to be implemented, and any arguments the member function needs for implementation.

Synonym for *member function call*.

model. A nonvisual part that represents the state and behavior of a object, such as a customer or an account.

Contrast with *view*.

module definition file. A file that describes the code segments within a load module.

Synonym for *DEF file*.

N

nested class. A class defined within the scope of another class.

nonvisual part. A part that has no visual representation at run time. A nonvisual part typically represents some real-world object that exists in the business environment.

Compare to *model*. Contrast with *view* and *visual part*.

no-event attribute. An attribute that does not have an event identifier.

no-set attribute. An attribute that does not have a set member function.

notebook part. A visual part that resembles a bound notebook containing pages separated into sections by tabbed divider pages. A user can turn the pages of a notebook or select the tabs to move from one section to another.

O

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) A collection of data and member functions that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and member functions. Each object in the class is said to be an instance of the class. (3) An instance of an object class consisting of attributes, a data structure, and operational member functions. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and member functions as other instances of the object class, though it has unique values assigned to its attributes.

object class. A template for defining the attributes and member functions of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

object factory. A nonvisual part capable of dynamically creating new instances of a specified part. For example, during the execution of an application, an object factory can create instances of a new class to collect the data being generated.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on those data objects that comprise

observer • pure virtual function

the problem and how they are manipulated, not on how something is accomplished.

observer. An object that receives notification from a notifier object.

operation. A member function or service that can be requested of an object.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

palette. See *parts palette*.

parameter connection. A connection that satisfies a parameter of an action or member function by supplying either an attribute's value or the return value of an action, member function, or custom logic. The parameter is always the source of the connection.

See also *connection*.

parent class. The class from which another part or class inherits data, member functions, or both.

part. A self-contained software object with a standardized public interface, consisting of a set of external features that allow the part to interact with other parts. A part is implemented as a class that supports the INotifier protocol and has a part interface defined.

The parts on the palette can be used as templates to create instances or objects.

part event. A representation of a change that occurs to a part. The events on a part's interface enable other interested parts to receive notification when something about the part changes. For example, a push button generates an event signaling that it has been clicked, which might cause another part to display a window.

part event ID. The name of a part static-data member used to identify which notification is being signaled.

part interface. A set of external features that allows a part to interact with other parts. A part's interface is made up of three characteristics: attributes, actions, and events.

Part Interface Editor. An editor that the application developer uses to create and modify attributes, actions, and events, which together make up the interface of a part.

Compare to *Class Editor* and *Composition Editor*.

parts palette. The parts palette holds a collection of visual and nonvisual parts used in building additional parts for an application. The parts palette is organized into *categories*. Application developers can add parts to the palette for use in defining applications or other parts.

preferred features. A subset of the part's features that appear in a pop-up connection menu. Generally, they are the features used most often.

primary selection. In the Composition Editor, the part used as a base for an action that affects several parts. For example, an alignment tool will align all selected parts with the primary selection. Primary selection is indicated by closed (solid) selection handles, while the other selected parts have open selection handles.

See also *selection handles*.

private. Pertaining to a class member that is accessible only to member functions and friends of that class.

process. A program running under OS/2, along with the resources associated with it (memory, threads, file system resources, and so on).

program. (1) One or more files containing a set of instructions conforming to a particular programming language syntax. (2) A self-contained, executable module. Multiple copies of the same program can be run in different processes.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its arguments.

primitive part. A basic building block of other parts. A primitive part can be relatively complex in terms of the function it provides.

process. A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task.

property. A unique characteristic of a part.

pure virtual function. A virtual function that has a function definition of = 0;.

receiver •visual programming tool

R

receiver. The object that receives a member function call.

Contrast with *caller*.

resource file. A file that contains data used by an application, such as text strings and icons.

S

selection handles. In the Composition Editor, small squares that appear on the corners of a selected visual part. Selection handles are used to resize parts.

See also *primary selection*.

server. A computer that provides services to multiple users or workstations in a network; for example, a file server, a print server, or a mail server.

service. A specific behavior that an object is responsible for exhibiting.

settings view. A view of a part that provides a way to display and set the attributes and options associated with the part.

sticky. In the Composition Editor, the mode that enables you to add multiple parts of the same class (for example, three push buttons) without going back and forth between the parts palette and the free-form surface.

structure. A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

subpart. A part that is used to create another part.

See also *nonvisual part*, *part*, and *visual part*.

superclass. See *abstract class* and *base class*.

T

tear-off attribute. An attribute that an application developer has exposed to work with as though it were a stand-alone part.

template. A family of classes or functions with variable types.

thread. A unit of execution within a process.

tool bar. The strip of icons along the top of the free-form surface. The tool bar contains tools to help you construct composite parts.

U

UI. See *user interface*.

unloaded. The state of the mouse pointer before you select a part from the parts palette and after you deposit a part on the free-form surface. In addition, you can unload the mouse pointer by pressing the Esc key.

user interface (UI). (1) The hardware, software, or both that enable a user to interact with a computer. (2) The term *user interface* normally refers to the visual presentation and its underlying software with which a user interacts.

V

variable. (1) A storage place within an object for a data feature. The data feature is an object, such as number or date, stored as an attribute of the containing object. (2) A part that receives an identity at run time. A variable by itself contains no data or program logic; it must be connected such that it receives runtime identity from a part elsewhere in the application.

view. (1) A visual part, such as a window, push button, or entry field. (2) A visual representation that can display and change the underlying model objects of an application. Views are both the end result of developing an application and the basic unit of composition of user interfaces.

Compare to *visual part*. Contrast with *model*.

virtual function. A function of a class that is declared with the keyword *virtual*. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called. This is determined at run time.

visual part. A part that has a visual representation at run time. Visual parts, such as windows, push buttons, and entry fields, make up the user interface of an application.

Compare to *view*. Contrast with *nonvisual part*.

visual programming tool. A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

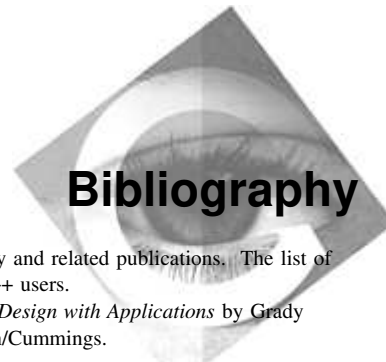
white space • window

W

white space. Space characters, tab characters, form-feed characters, and new-line characters.

window. (1) A rectangular area of the screen with visible boundaries in which information is displayed. Windows can overlap on the screen, giving it the appearance of one

window being on top of another. (2) In the Composition Editor, a window is a part that can be used as a container for other visual parts, such as push buttons.



This bibliography lists the publications that make up the IBM VisualAge for C++ library and related publications. The list of related publications is not exhaustive but should be adequate for most VisualAge for C++ users.

The IBM VisualAge for C++ Library

The following books are part of the IBM VisualAge for C++ library.

- Installation Guide & Product Overview, S33H-5030
- User's Guide, S33H-5031
- Programming Guide, S33H-5032
- Visual Builder User's Guide, S33H-5034
- Visual Builder Parts Reference, S33H-5035
- Building VisualAge for C++ Parts for Fun and Profit, S33H-5036
- Open Class Library User's Guide, S33H-5033
- Open Class Library Reference, S33H-5039
- Language Reference, S33H-5037-00
- C Library Reference, S33H-5038
- SOM Programming Guide, S33H-5044
- SOM Programming Reference, &dnsomrf.

C and C++ Related Publications

- *Portability Guide for IBM C*, SC09-1405
- *American National Standard for Information Systems / International Standards Organization — Programming Language C (ANSI/ISO 9899-1990[1992])*

Non-IBM Publications

Many books have been written about the C++ language and related programming topics. The authors use varying approaches and emphasis. The following is a sample of some non-IBM C++ publications that are generally available. This sample is not an exhaustive list. IBM does not specifically recommend any of these books, and other C++ books may be available in your locality.

- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.
- *C++ Primer* by Stanley B. Lippman, Addison-Wesley Publishing Company.

- *Object-Oriented Design with Applications* by Grady Booch, Benjamin/Cummings.
- *Object-Oriented Programming Using SOM and DSOM* by Christina Lau, Van Nostrand Reinhold.

Related Information

The following publications can help you find more information on OO software design and Common User Access (CUA) user interface design.

OO Programming and Design

- Booch, Grady. *Object Oriented Design with Applications*.

This book explores OO concepts and design. It shows you, through various techniques and examples, how to develop applications using OO.

It is available through Benjamin/Cummings, Redwood City, California. Its ISBN number is 0-8053-0091-0.

- Cox, Brad J. *Object-Oriented Programming: An Evolutionary Approach*.

This book explores OO programming. It shows you how to build reusable objects through examples and provides a comparison between OO programming and traditional programming.

It is available through the Addison-Wesley Publishing Company, Reading, Massachusetts. Its ISBN number is 0-201-10393-1.

User Interface

- *Object-Oriented Interface Design - IBM Common User Access Guidelines*.

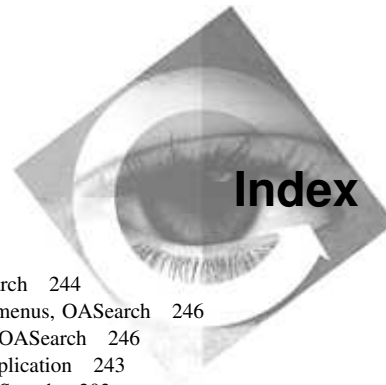
This book describes the guidelines that define the Common User Access user interface. The Common User Access user interface is an OO graphical user interface that provides a consistent look and feel for products that adopt the CUA interface as their standard.

It is available through QUE Corporation, Carmel, Indiana. Its ISBN number is 1-56529-170-0.

- Rubenstein, R. and Hersch, H. *The Human Factor: Designing Computer Systems For People*.

This book discusses user-centered design in its full scope: task analysis, prototyping, empirical evaluating, interface techniques, and guidelines.

It is available through Digital Press, Maynard, Massachusetts.



A

- abstract classes 88
- access type, changing 193
- action
 - adding 70
 - changing in Part Interface Editor 72
 - changing parameter names 72
 - changing settings when target 191
 - connecting menu choices, OASearch 247
 - connecting to exception events and member functions 188
 - defining on Action page 69
 - deleting in Part Interface Editor 72
 - getContractor, modifying code 109
 - OASkillBase part 93
 - parseName, modifying code 110
 - putContractor, modifying code 110
 - refreshID, modifying code 110
 - setting defaults, Part Interface Editor 72
 - successful, displaying feedback about 261
 - updating in Part Interface Editor 72
- Action page, defining actions 69
- activating settings changes 144
- adding
 - action 70
 - attribute in Part Interface Editor 58
 - basic visual parts to GUI, OASearch 158
 - categories and parts to parts palette, introduction 293
 - category to parts palette, steps 296
 - code created outside Visual Builder 112
 - code to part 105
 - code to set activeStatus attribute 110
 - code to set contractorID attribute 111
 - container column 234
 - container part to OASkillView* part 232
 - container parts 231
 - custom logic connection 184
 - event 65
 - extra rows and columns to hold group box on multicell canvas 225
 - group box on multicell canvas 225
 - handler 140
 - help 249
 - Help push button 257
 - information area to frame window 260

adding (*continued*)

- menu bar, OASearch 244
- menu choices to menus, OASearch 246
- menu separators, OASearch 246
- menus to your application 243
- message box, OASearch 203
- multicell canvas 219
- nonvisual part to container 236
- nonvisual parts to primary part, OASearch 263
- notebook 239
- notebook pages 240
- notebook parts 237
- object factory parts, OASearch 265
- object factory parts, OASearch, introduction 265
- parameters and types 66
- part to parts palette 299
- part when editing 300
- part, vbb file loaded 301
- parts to multicell canvas 219
- parts to notebook page 241
- preferred feature 76
- rows and columns on multicell canvas using contextual menu 222
- rows and columns on multicell canvas using settings notebook 229
- selected part, Visual Builder window 299
- several copies of same part 125
- static parts, OASearch 264
- variable parts, OASearch 266
- variable to free-form surface 160
- variables to part interface 162
- visual parts as dynamic instances, OASearch 264
- adding variable
 - composite part 160
 - free-form surface 160
 - to part interface 162
- adjusting size for translated text using canvases 305
- Align bottom tool 137
- Align center tool 136
- Align left tool 136
- Align middle tool 136
- Align right tool 137
- Align top tool 136
- aligning
 - parts 12
 - parts on free-form surface 136

aligning (*continued*)

- push buttons, To-Do List 15
- static text, entry field, list box, To-Do List 14
- aligning static text, entry field, and list box, To-Do List 14
- alignment tools 43
- all part files
 - deselecting 33
 - selecting 33
- application
 - adding help 249
 - aligning parts, To-Do List 12
 - basic, developing 79
 - build, files 55
 - building modified To-Do List application 215
 - building new To-Do List 215
 - bypassing limitations, Direct-to-SOM objects 313
 - combining all visual parts, OASearch 263
 - compiling and linking 279
 - composing parts, Composition Editor 40
 - creating To-Do List, introduction 7
 - design, object-oriented approach 84
 - designing 83
 - designing parts for single purpose 83
 - finished To-Do List 7
 - generating source code 273
 - help window 254
 - integrating visual parts into single application 263
 - matching widths of Add to Remove push buttons, To-Do List 13
 - matching widths, list box and entry field, To-Do List 12
 - OASearch, adding static parts 264
 - OASearch, adding visual parts as dynamic instances 264
 - OASearch, designing parts 90
 - OASearch, making connections 199
 - object-oriented, high-level steps for creating 84
 - packaging 279
 - placing another static text part, To-Do List window 11
 - placing parts, To-Do List application window 10
 - placing static text part in To-Do List window 10
 - resizing parts, To-Do List 12
 - running modified To-Do List application 215
 - running new To-Do List 215
 - sample, designing parts 90
 - starting Visual Builder for To-Do List 8
 - steps for building OASearch 99
 - To-Do List, aligning entry field, list box, static text 14
 - To-Do List, aligning push buttons 15
 - To-Do List, building 21
 - To-Do List, centering entry field and list box 14
 - To-Do List, centering push buttons 15

application (*continued*)

- To-Do List, changing title of application window 10
 - To-Do List, connecting Add push button to list box 16
 - To-Do List, connecting parts 16
 - To-Do List, connecting Remove push button to list box 18
 - To-Do List, creating new visual part 8
 - To-Do List, dragging and dropping parts 13
 - To-Do List, exiting Composition Editor and Visual Builder 22
 - To-Do List, generating source code 20
 - To-Do List, placing entry field in window 11
 - To-Do List, placing list box in window 11
 - To-Do List, placing push buttons in window 12
 - To-Do List, running 21
 - To-Do List, spacing push buttons evenly 16
 - To-Do List, steps 7
 - Visual Builder, Direct-to-SOM objects 312
 - Visual Builder, extending 285
 - Visual Builder, readying for others 273
 - window, resizing, To-Do List 13
 - yours, adding menus 243
- arranging
- parts 159
 - parts, introduction 134
- Asian languages, DBCS support for Asian languages 306
- attribute
- activeStatus, adding code to set 110
 - adding in Part Interface Editor 58
 - changing 61
 - contractorID, adding code to set 111
 - deleting 61
 - description 60
 - event identification 60
 - example 62
 - get member function 59
 - name 58
 - no-event 58
 - no-set 58
 - OACContract part 92
 - OACContractor part 91
 - OASkill part 93
 - set member function 59
 - setting defaults, Part Interface Editor 61
 - tear-off, variable, example 161
 - tearing-off 156
 - type 58
 - type as source in connections 60
 - types, effect on connections 164
 - updating 61

- attribute (*continued*)
 - when to connect to member functions 179
- Attribute page fields, clearing 61
- attribute type 58
- attribute-to-action connection 167
- attribute-to-attribute connection
 - changing settings 190
 - definition 163
 - OASearch 199
- attribute-to-member function connection 168

B

- base class of part, displaying in Class Editor 51
- base files, displaying 34
- base part, new part derived from 151
- benefits of using Visual Builder 3
- bounding box, spacing parts 137
- bracket, square, symbols 161
- Browser information 182
- browsing features 172
- building
 - application, files 55
 - application, To-Do List 21
 - derived part using inheritance 86
 - IPF help file 251
 - modified To-Do List application 215
 - new To-Do List application 215
 - OASearch application, steps 99
 - resizable windows 217
 - windows with multicell canvas 217
- Buttons category 45

C

- C and C++ code, existing, using with Visual Builder 287
- C/C++ window (OS/2), starting Visual Builder from 23
- C++ code
 - existing 101
 - generating, To-Do List 20
- C++ folder (VisualAge for Windows), starting Visual Builder 24
- C++ source code
 - generating for main() function 276
 - individual parts, generating for 274
- C++ Tools folder (VisualAge for OS/2), starting Visual Builder 24
- canvases, adjusting size for translated text 305
- category
 - adding to parts palette, introduction 293

- category (*continued*)
 - adding to parts palette, steps 296
 - deleting from parts palette 302
 - removing from parts palette, just added 302
 - saving changes on parts palette 302
 - Visual Builder, table 128
- centering
 - entry field and list box, To-Do List 14
 - push buttons, To-Do List 15
- changing
 - (renaming) part name, free-form surface 133
 - (sizing) parts, free-form surface 136
 - access type 193
 - actions in Part Interface Editor 72
 - activating settings 144
 - arranging parts, introduction 134
 - attribute 61
 - code for getContractor action 109
 - code for parseName action 110
 - code for putContractor action 110
 - code for refreshID action 110
 - color 142
 - default minimum size, rows and columns, multicell canvas 228
 - depth order, composite part 146
 - event 67
 - font for part 142
 - generated feature code 109
 - group 150
 - labels 159
 - name of source feature 192
 - name of target class 192
 - parameter names and types 66
 - part size (matching) 136
 - part's constructor in Class Editor 51
 - parts on free-form surface 151
 - parts palette, saving 302
 - promoted feature 74
 - redoing 157
 - settings 159
 - settings for connection 189
 - settings for multicell canvas 227
 - settings for part 138
 - settings, action as target 191
 - settings, attribute-to-attribute connections 190
 - settings, custom logic as target 194
 - settings, member function as target 192
 - source and target of connection 198
 - tab and depth order 147
 - tab order 148, 149

- changing (*continued*)
 - tab stop 150
 - target member function 193
 - title of To-Do List application window 10
 - undoing 157
 - variable type 161
- charts
 - connection types you can change 199
 - optimal sizes for components 89
 - types of connections, summary 170
 - Visual Builder categories 128
- Class Editor
 - base class of part, seeing 51
 - constructor code 52
 - entering description of part 50
 - features 49
 - introduction 49
 - library file 52
 - modifying part's constructor in Class Editor 51
 - moving part to different part file 51
 - naming code generation files 54
 - starting resource ID 53
 - unique icon for part 53
 - your destructor code 52
- classes, abstract 88
- clearing
 - Action page fields 73
 - Attribute page fields 61
 - containers 237
 - Event page fields 67
 - Promote page fields 75
- clipboard, copying part 132
- code
 - adding to part 105
 - adding to set `activeStatus`. attribute 110
 - adding to set `contractorID` attribute 111
 - adding, created outside Visual Builder 112
 - C++, generating, To-Do List 20
 - constructor, yours in Class Editor 52
 - destructor, yours in Class Editor 52
 - modifying for `getContractor` action 109
 - modifying for `parseName` action 110
 - modifying for `putContractor` action 110
 - modifying for `refreshID` action 110
 - source, sample of help 252
 - steps for developing outside Visual Builder 112
 - strings, using to supply field values 143
- code generation files, naming in Class Editor 54
- color area, group box 142
- Color page 142
- color, changing 142
- colors list box 142
- combining all visual parts, OASearch 263
- compilation, preparing generated files 277
- compiler program
 - browser option 278
 - linker programs, debug options 278
- compiling application 279
- completed To-Do List 7
- completing menu bar, OASearch 269
- components, suggested size 89
- Composers category 47
- Composers parts
 - spacing subparts 137
 - toggling grid, showing and hiding 135
- composing parts for application, Composition Editor 40
- composite part
 - adding variable 160
 - changing depth order 146
 - listing parts 146
 - supplementary, as subpart 128
- Composition Editor
 - exiting, To-Do List 22
 - free-form surface 48
 - introducing 40
 - promoting part's features 154
 - undoing and redoing changes 157
- concepts
 - connections 4
 - parts 4
 - source code generation 5
 - Visual Builder 4
- connecting
 - Add push button to list box, To-Do List 16
 - attributes or events to member functions, when 179
 - exception events to actions and member functions 188
 - features to custom logic 184
 - features to features 174
 - features to member function connections 179
 - incomplete, supplying parameter values 175
 - menu bar to window, OASearch 245
 - menu choices to actions, OASearch 247
 - nonvisual parts to variables, OASearch 267
 - object factory parts to variables, OASearch 267
 - object factory parts, OASearch 266
 - parts to fill container 236
 - parts, To-Do List 16
 - variable part, OASearch 199

- connection
 - attribute type as source 60
 - attribute types' effect 164
 - attribute-to-action 167
 - attribute-to-attribute 163
 - attribute-to-member function 168
 - changing source and target 198
 - concepts 4
 - custom logic, adding 184
 - custom logic, overview 168
 - deleting 195
 - deselecting 198
 - Edit submenu, OASearch 269
 - event-to-action 165
 - event-to-attribute 165
 - event-to-member function 166, 180
 - hiding 195
 - making 171
 - making for OASearch 199
 - making new ones, To-Do List 213
 - manipulating 189
 - modal window, OASearch 267
 - modeless window, OASearch 268
 - moving end points 198
 - parameter 169
 - rearranging 196
 - Remove push button to list box, To-Do List 18
 - reorder 194
 - selecting 197
 - selecting multiple ones in OS/2 197
 - selecting multiple ones in Windows 197
 - selecting one 197
 - settings, changing 189
 - showing 195
 - source and target 171
 - supplying parameter value 175
 - types you can change, table 199
 - types, introduction 163
 - types, summary 170
 - using 163
 - View submenu, OASearch 269
- connection menu for part, adding preferred feature 76
- connection menu for part, removing all preferred features 77
- connection menu for part, removing preferred feature 77
- connection settings window, push buttons 193
- connection, adding event-to-member function 180
- connections between features and custom logic 184
- connections between features and member functions 179
- constant value, using to supply parameter value 177
- constructing
 - containers and notebooks 231
 - user interface using OASearch 157
- constructor
 - code, yours in Class Editor 52
 - part's, modifying in Class Editor 51
- container
 - adding columns 234
 - adding nonvisual part 236
 - clearing 237
 - columns, setting up for icons 234
 - columns, setting up for strings 235
 - connecting parts to fill 236
 - constructing 231
 - controls, IBM Open Class Library 231
 - filling 236
 - part, adding to OASkillView* part 232
 - parts, adding 231
 - populating collection 236
 - setting up 232
 - setting values 232
 - title, setting 232
 - type and layout 233
- container columns, adding 234
- context-sensitive help, providing 252
- contextual menu
 - adding rows and columns on multicell canvas 222
 - deleting rows and columns on multicell canvas 223
- control
 - container, IBM Open Class Library 231
 - notebook, IBM Open Class Library 231
- Control settings page 139
- convention, naming of OACContractor part 91
- copying part
 - dragging parts 131
 - one part file to another 121
 - ToDoList part 208
 - with clipboard 132
- country-sensitive format for parts 306
- creating
 - basic application 79
 - Composers and primitive visual parts (your own) 290
 - derived part using inheritance 86
 - help file 250
 - help file (IPF) 251
 - new part 117
 - new visual part, To-Do List 8
 - nonvisual parts 101
 - OASearch application, steps 99
 - part information file 287

- creating (*continued*)
 - part information file, Direct-to-SOM objects 310
 - part, adding code 105
 - part, generating feature code 106
 - part, generating feature code, OACContractor 106
 - parts, overview 101
 - resizable windows 217
 - source files, part code generation 275
 - To-Do List application, introduction 7
 - To-Do List, steps 7
 - ToDoItem nonvisual part 208
 - using multicell canvas to build windows 217
- custom logic
 - changing settings when target 194
 - connecting features 184
 - connecting to features 184
 - connection 168
 - connection, adding 184
- customizing information area 34

D

- Data entry category 45
- data type (default), int 58
- data type of attribute 58
- database, simulated, OASearch 90
- DBCS support for Asian languages 306
- debug options for compiler and linker programs 278
- default data type 58
- default push button 160
- defaults for parameters 177
- defaults, setting for attributes in Part Interface Editor 61
- defaults, setting for events, Part Interface Editor 67
- defining
 - actions on Action page 69
 - part interface using part information files 287
 - part interface, Part Interface Editor 102
 - parts 101
- definition of Visual Builder 3
- deleting
 - action in Part Interface Editor 72
 - attribute 61
 - category from parts palette, steps 302
 - connections 195
 - event 67
 - parameter names and types 66
 - part from parts palette, steps 302
 - parts from free-form surface 132
 - parts from part file 123
 - promoted feature 75

- deleting (*continued*)
 - rows and columns from multicell canvas 222
 - rows and columns on multicell canvas using contextual menu 223
 - rows and columns on multicell canvas using settings notebook 229
- depth and tab order for multiple parts, changing 147
- depth order, changing in composite part 146
- derived part, building using inheritance 86
- deriving new part from base part 151
- description of attribute 60
- description of part, Class Editor 50
- deselecting
 - all part files 33
 - all parts 114
 - connections 198
 - parts 131
 - parts, introduction 129
- design approach, object-oriented 84
- designing
 - applications 83
 - nonvisual parts 90
 - parts for single purpose 83
 - parts, OASearch 90
 - visual parts, OASearch 94
- destructor code, yours in Class Editor 52
- determining source and target of connection 171
- developing basic application 79
- developing code outside Visual Builder, steps 112
- directory, setting working 36
- display, refreshing 37
- displaying
 - base files 34
 - help for menu choices in information area 260
 - inherited preferred features only 78
 - long fly-over text in information area 260
 - part names 113
 - pop-up menu 131
- Distribute horizontally tool 137
- Distribute vertically tool 137
- distributing parts to others 283
- distribution tools on tool bar 44
- DLL, resource, preparing for OS/2 294
- DLL, resource, preparing for Windows 295
- dragging and dropping parts, To-Do List 13
- dragging parts to copy 131
- dropping group box, multicell canvas 226
- DTS object
 - creating and importing part information file 310
 - introduction 309

DTS object (*continued*)
objects 309
objects, bypassing limitations 313
objects, Visual Builder application 312

E

Edit submenu, connections from, OASearch 269
editing
 adding part to parts palette 300
 parts on free-form surface 151
 text strings 133
editor symbols, fast-path 39
enabling
 national language support 303
 national language support, introduction 303
 push buttons in OASearch 201
 push buttons, entry field contains value, OASearch 200
 window to be cleared of all entry values, OASearch 204
end points of a connection, moving 198
ending Composition Editor and Visual Builder, To-Do List 22
entry field
 aligning list box and static text with, To-Do List 14
 centering with list box, To-Do List 14
 enabling push buttons when contains value, OASearch 200
 matching width with list box, To-Do List 12
 placing in To-Do List window 11
 read-only 150
 setting tab stop 150
entry values, all, enabling window to be cleared of, OASearch 204
event
 adding 65
 adding parameters 66
 changing 67
 changing parameter names and types 66
 deleting 67
 deleting parameter names and types 66
 description 65
 example 67
 identifier, name 65
 name 65
 parameters 65
 Ready event 64
 setting defaults, Part Interface Editor 67
 when to connect to member functions 179
event identification, attribute 60
Event page, adding event 65
Event page, clearing fields 67
event-to-action connection 165
event-to-attribute connection 165
event-to-member function connection 166
examples
 attribute 62
 event 67
 ready event 65
 tear-off attribute 161
exception events, connecting to actions and member functions 188
exceptions, passing to message box 203
existing C++ code 101
exiting Composition Editor and Visual Builder, To-Do List 22
expandable rows and columns on multicell canvas 229
exporting part information 116
extending
 group box, multicell canvas 226
 part to span more than one cell on multicell canvas 223
 Visual Builder applications 285

F

factory-generated frame windows, providing help 256
fast-path, editor symbols 39
FAT file names 36
feature
 Class Editor 49
 connect to member function connections 179
 connecting features to 174
 connecting to custom logic 184
 deleting promoted one 75
 member functions, connections between 179
 Part Interface Editor 56
 part's, browsing 172
 part's, promoting 153
 part's, promoting from Composition Editor 154
 preferred, adding 76
 preferred, removing 77
 Promote page 73
 promoting in Part Interface Editor 74
 removing all preferred ones 77
 showing inherited preferred features only 78
feature code, generated, modifying 109
field value, supplying using code strings 143
fields
 clearing for Attribute page 61
 clearing on Action page 73

- fields (*continued*)
 - clearing on Event page 67
- file allocation table (FAT) file names 36
- files
 - application build 55
 - base, displaying 34
 - code generation, naming in Class Editor 54
 - HTML documentation, generating 274
 - make, generating 273
 - make, setting generation 36
 - part information, creating and importing,
 - Direct-to-SOM 310
 - source, created, generation of main() function 276
 - source, created, part code generation 275
 - storing parts, vbb file 31
- filling
 - container by connecting parts 236
 - containers 236
- finished To-Do List 7
- fly-over help 258
- fly-over help for subpart 258
- fly-over text, long, displaying in information area 260
- Font page 142
- font, changing for part 142
- Frame extensions category 46
- frame window, adding information area to 260
- frame windows, help for factory-generated ones 256
- free-form surface
 - adding message box, OASearch 203
 - adding variable 160
 - deleting parts 132
 - editing part 151
 - introduction 48
 - object factory part (IVBFactory*) on free-form surface,
 - To-Do List 212
 - placing IVSequence* part, To-Do List 212
 - placing part, introduction 124
 - placing parts on when not on parts palette 125
 - renaming parts 133
 - sizing parts 136
- full file names, showing 34

G

- general help, providing 253
- General settings page 139
- generated feature code, modifying 109
- generating
 - Browser information, option 278
 - C++ source code for individual parts 274

- generating (*continued*)
 - code for application, To-Do List 20
 - feature code 106
 - feature code for OASearch part 106
 - HTML documentation 274
 - main() function, source files created 276
 - make files, menu choice 273
 - part code, source files created 275
 - preparing files for compilation 277
 - setting, for make files 36
 - source code for main() function 276
 - source code for parts and applications 273
 - source code for visual part, To-Do List 214
 - source code for your visual part 20
 - source code, concepts 5
 - source code, main() procedure, To-Do List 214
 - source code, preparing 273
 - source code, your main() function 20
- generic settings notebook 144
- get member function, attribute 59
- getContractor action, modifying code 109
- graphical user interface (GUI)
 - adding basic visual parts, OASearch 158
 - changing labels 159
 - changing settings 159
 - constructing using OASearch 157
 - construction, introduction 157
 - default push button 160
 - mnemonic, push button label 159
 - push button label, mnemonic 159
- grid
 - changing multicell canvas grid 222
 - hiding 135
 - positioning parts 134
 - showing 135
 - spacing 135
 - toggle 135
 - tools 43
- group box
 - adding extra rows and columns on multicell canvas 225
 - adding on multicell canvas 225
 - color area 142
 - dropping and extending, multicell canvas 226
 - minimum size 142
- group, setting 150
- groups of radio buttons 150
- groups, style guidelines 150
- GUI
 - See* graphical user interface (GUI)

- guidelines
 - placing parts 127
 - setting groups and tab stops 150
 - starting resource IDs 305

H

- h, resource header file 304
- handler
 - adding 140
 - moving 141
 - removing 141
- Handlers page 140
- help
 - adding 249
 - application help window 254
 - context-sensitive 252
 - creating file for 250
 - displaying for menu choices in information area 260
 - displaying long fly-over text in information area 260
 - fly-over 258
 - fly-over for subpart 258
 - for factory-generated frame windows 256
 - general 253
 - Help push button, adding 257
 - hover 258
 - hover for subpart 258
 - information area, displaying in 259
 - information area, feedback on successful actions 261
 - sample source code 252
 - writing file for portability 250
- help file
 - building in IPF 251
 - creating for help 250
- Help push button, adding 257
- help window, application, providing 254
- help, fly-over for subpart 258
- help, hover for subpart 258
- hiding
 - Composers part (toggle to showing) 135
 - connections 195
 - grid 135
- highlighting conventions xv
- hints for using Visual Builder 315
- hover help
 - displaying 258
 - subpart 258
 - text, long, displaying in information area 260
- hover help for subpart 258

- HTML documentation, generating from Visual Builder 274
- Hypertext Markup Language documentation, generating from Visual Builder 274

I

- IBM Open Class Library, container and notebook controls 231
- ICollectionComboBox to display objects, introduction, To-Do List 207
- ICollectionListBox replaces IListBox, To-Do List 211
- ICollectionListBox to display objects, introduction, To-Do List 207
- icon
 - preparing for parts palette 293
 - unique, for part added to parts palette 298
 - unique, for part in Class Editor 53
- icon container columns, setting up 234
- identifying reusable parts 85
- IDs, starting resource, guidelines 305
- IListBox, replacing with ICollectionListBox, To-Do List 211
- incomplete connections, supplying parameter values 175
- information area
 - adding to frame window 260
 - customizing 34
 - displaying feedback about successful actions 261
 - displaying help in 259
 - displaying help in, for menu choices 260
 - displaying long fly-over text in 260
- inheritance
 - tree for browsing 182
 - using 86
 - using to build derived part 86
- inherited preferred features, showing only 78
- int, default data type 58
- integrating visual parts into single application 263
- interface
 - adding basic visual parts, OASearch 158
 - changing labels 159
 - changing settings 159
 - constructing using OASearch 157
 - construction, introduction 157
 - default push button 160
 - push button label, with mnemonic 159
- introduction
 - adding and removing categories and parts from parts palette 293
 - adding and setting object factory parts, OASearch 265
 - adding categories and parts to parts palette 293

- introduction (*continued*)
 - arranging parts 134
 - Class Editor 49
 - Composition Editor 40
 - connecting features to custom logic 184
 - connection types 163
 - creating To-Do List application 7
 - Direct-to-SOM objects 309
 - enabling national language support 303
 - GUI construction 157
 - list boxes to display objects, To-Do List 207
 - Part Interface Editor 56
 - parts palette 44
 - placing parts on free-form surface 124
 - selecting and deselecting parts 129
 - starting Visual Builder 23
 - tool bar 41
 - Visual Builder editors, introduction 39
 - Visual Builder window 29
- invalid parameters 178
- item types in list box, To-Do List 211
- IVBFactory* (object factory part), modifying, To-Do List 212
- IVSequence* part on free-form surface, To-Do List 212
- IVSequence* part, placing and modifying, To-Do List 212

K

- key concepts about parts 4
- key concepts about Visual Builder 4

L

- labels, changing on interface 159
- languages, Asian, DBCS support 306
- layout, notebook 239
- learning to use parts 113
- library file
 - additional, make file 278
 - Class Editor 52
- limitations, bypassing, Direct-to-SOM objects 313
- linker programs, debug options 278
- linking
 - See* connection
- linking application 279
- list box
 - centering with entry field, To-Do List 14
 - colors 142
 - connecting to Add push button, To-Do List 16
 - connecting to Remove push button, To-Do List 18

- list box (*continued*)
 - displaying objects, To-Do List 207
 - handler list 141
 - item types, To-Do List 211
 - matching width with entry field, To-Do List 12
 - placing in window 11
 - replacing and modifying, To-Do List 211
 - to display objects, introduction, To-Do List 207
- list, type, showing 35
- listing parts in composite part 146
- Lists category 46
- loaded vbb file, adding part 301
- loading part files 31
- long fly-over text, displaying in information area 260
- long hover help text, displaying in information area 260

M

- main() function
 - generating C++ source code 276
 - generation, source files created 276
 - yours, generating source code 20
- main() procedure, generating source code, To-Do List 214
- make file
 - additional libraries 278
 - attribute-to-attribute connections, OASearch 199
 - event-to-action connections, OASearch 202
 - generating 273
 - new connections, To-Do List 213
 - setting generation 36
- making connections 171
- manipulating
 - connections 189
 - parts 131
- Match height tool 136
- matching
 - part sizes 136
 - widths of list box and entry field 12
 - widths of push buttons, To-Do List 13
- member function
 - actions, connecting to exception events 188
 - changing settings when target 192
 - connect to features 179
 - features, connections between 179
 - when to connect attributes or events 179
- menu
 - adding menu bar, OASearch 244
 - adding menu choices, OASearch 246
 - adding separators, OASearch 246
 - adding to your application 243

- menu (*continued*)
 - and menu items, types 243
 - contextual, adding rows and columns on multicell canvas 222
 - contextual, deleting rows and columns on multicell canvas 223
 - OASearch Welcome window 243
- menu bar
 - adding to menu, OASearch 244
 - connecting to window, OASearch 245
- menu choice
 - displaying help for in information area 260
- menu choices, connecting actions, OASearch 247
- menu separators, adding, OASearch 246
- message box
 - adding to free-form surface, OASearch 203
 - passing exceptions 203
- methods for designing nonvisual parts 90
- minimum size group box 142
- missing parameters 178
- mnemonic for push button label 159
- modal window connections, OASearch 267
- modeless window connections, OASearch 268
- Models category 48
- modified To-Do List application, building and running 215
- modifying and replacing list box, To-Do List 211
- modifying code
 - getContractor action 109
 - parseName action 110
 - putContractor action 110
 - refreshID action 110
- modifying generated feature code 109
- modifying IVSequence* part, To-Do List 212
- modifying object factory part (IVBFactory*), To-Do List 212
- mouse pointer, unloading 125
- moving
 - dropping parts in multicell canvas 227
 - end points of connection 198
 - handler 141
 - part to different part file 121
 - part to different part file in Class Editor 51
 - part, free-form surface 134
 - through settings notebook 139
- MS/DOS command window (Windows), starting Visual Builder from 23
- multicell canvas
 - adding 219
 - adding extra rows and columns to hold group box 225
 - adding group box 225

- multicell canvas (*continued*)
 - adding parts 219
 - adding rows and columns using contextual menu 222
 - adding rows and columns using settings notebook 229
 - benefits 217
 - changing default minimum size, rows and columns 228
 - changing grid 222
 - changing settings 227
 - deleting rows and columns using contextual menu 223
 - deleting rows and columns using settings notebook 229
 - dropping and extending group box 226
 - extending part to span more than one cell 223
 - making rows or columns expandable 229
 - moving dropped parts 227
 - using to build windows 217
- multiple parts
 - parts, selecting 130
 - sizing 136

N

- name
 - changing for source feature 192
 - changing for target class 192
 - code generation files, specifying in Class Editor 54
 - FAT file names 36
- name of attribute 58
- naming convention, OACContractor part 91
- national language support
 - adjusting size for translated text using canvases 305
 - country-sensitive format for parts 306
 - DBCS support for Asian languages 306
 - enabling 303
 - resource files for translation 303
- navigating through settings notebook 139
- new part from base part 151
- new To-Do List application, running 215
- NLS
 - See* national language support
- no-event attribute 58
- no-set attribute 58
- nonvisual part
 - adding code 105
 - adding to container 236
 - adding to primary part, OASearch 263
 - connecting to variables, OASearch 267
 - creating 101
 - defining interface 101
 - designing 90
 - generating feature code 106

- nonvisual part (*continued*)
 - generating feature code, OACContractor part 106
 - oanonvis.vbb, OASearch 90
 - ToDoItem part 208
- notebook
 - adding 239
 - adding pages 240
 - adding parts 237
 - constructing 231
 - controls, IBM Open Class Library 231
 - generic settings 144
 - layout 239
 - page, adding parts 241
 - page, setting up 240
 - settings for multiple parts 139
 - settings, opening 138
 - tab, setting up 240

O

- OACContract part's attributes and actions 92
- OACContractor part 91
- OACContractor part's attributes 91
- OAMain, primary view, OASearch 94
- oanonvis.vbb, nonvisual parts, OASearch 90
- OASearch application
 - adding basic visual parts to GUI 158
 - adding menu bar 244
 - adding menu choices to menus 246
 - adding message box 203
 - adding nonvisual parts to primary part, OASearch 263
 - adding object factory parts 265
 - adding static parts 264
 - adding variable parts 266
 - adding visual parts as dynamic instances 264
 - attribute-to-attribute connections 199
 - combining all visual parts 263
 - completing menu bar 269
 - connecting menu bar to window 245
 - connecting menu choices 247
 - connecting nonvisual parts to variables 267
 - connecting to object factory parts 266
 - connecting variable part 199
 - connections from Edit submenu 269
 - connections from View submenu 269
 - constructing user interface 157
 - designing nonvisual parts 90
 - designing parts 90
 - designing visual parts 94
 - enabling push buttons 201

- OASearch application (*continued*)
 - enabling push buttons when entry field contains value 200
 - enabling window to be cleared of all entry values 204
 - event-to-action connections 202
 - making connections 199
 - modal window connections 267
 - modeless window connections 268
 - naming convention, OACContractor part 91
 - nonvisual parts in oanonvis.vbb 90
 - OACContract part's attributes and actions 92
 - OACContractor part 91
 - OACContractor part's attributes 91
 - OASkill part's attributes 93
 - OASkillBase part's actions 93
 - object factory parts, adding and setting, introduction 265
 - object factory parts, setting 265
 - Opportunities Abound Contract Information window 98
 - Opportunities Abound Contractor Information window 96
 - Opportunities Abound Databases window 94
 - Opportunities Abound General Information window 95
 - Opportunities Abound Skill Information window 98
 - primary view, OAMain 94
 - query windows 96
 - resetting resource library 270
 - simulated database 90
 - steps for building 99
 - visual parts in oawin.vbb 90
 - Welcome window with menus 243
- OASkill part's attributes 93
- OASkillBase part's actions 93
- OASkillView* part, adding container part 232
- oawin.vbb, visual parts, OASearch 90
- object factory
 - description 264
 - dynamic windows, OAMain part in OASearch 264
 - object types, To-Do List 212
 - part (IVBFactory*), modifying in To-Do List 212
 - part (IVBFactory*), placing in To-Do List 212
 - part on free-form surface, To-Do List 212
 - parts, adding and setting, OASearch, introduction 265
 - parts, adding with variable parts, OASearch 266
 - parts, adding, OASearch 265
 - parts, connecting to variables, OASearch 267
 - parts, connecting to, OASearch 266
 - parts, setting, OASearch 265
- object types
 - in sequence, To-Do List 213
 - object factory can create, To-Do List 212

- object-oriented
 - application, high-level steps for creating 84
 - approach to design 84
- objects, displaying in list box, To-Do List 207
- Open Class Library (IBM), container and notebook controls 231
- opening
 - multiple parts 120
 - one part 119
 - parts 119
 - settings notebook for multiple parts 139
 - settings notebook for part 138
- Opportunities Abound Contract Information window, OASearch 98
- Opportunities Abound Contractor Information window, OASearch 96
- Opportunities Abound Databases window, OASearch 94
- Opportunities Abound General Information window, OASearch 95
- Opportunities Abound Skill Information window, OASearch 98
- order, tab, setting 148
- OS/2
 - deselecting connections 198
 - preparing resource DLL 294
 - selecting multiple connections 197
- Other category 48
- overlying parts 127
- overview
 - adding and removing categories and parts from parts palette 293
 - connection types 163
 - creating parts 101
 - GUI construction 157
 - visual construction 113

P

- packaging application 279
- page
 - Action 69
 - Color 142
 - Control settings 139
 - Font 142
 - General settings 139
 - Handlers 140
 - Promote, features 73
 - Promote, promoting features in Part Interface Editor 74
 - Size/Position 142
 - Styles settings 140
- page and tab for notebook, setting up 240
- pages
 - adding notebook 240
 - settings, descriptions 139
- parameter connections 169
- parameter names, changing in Part Interface Editor 72
- parameters
 - connections 169
 - defaults 177
 - missing 178
 - not valid 178
 - value, supplying using connection 175
 - value, supplying using constant value 177
 - values, supplying for incomplete connections 175
- parseName action, modifying code 110
- part
 - add when editing 300
 - adding code 105
 - adding to GUI, OASearch 158
 - adding to notebook page 241
 - adding to parts palette 299
 - adding to parts palette, introduction 293
 - aligning static text, entry field, list box, To-Do List 14
 - aligning, free-form surface 136
 - aligning, To-Do List 12
 - arranging 159
 - arranging, introduction 134
 - base class, displaying in Class Editor 51
 - browsing features of 172
 - changing depth order 146
 - changing font 142
 - changing settings 138
 - changing tab order 149
 - Composers, creating own 290
 - Composers, toggling grid, showing and hiding 135
 - composing with Composition Editor 40
 - composite, adding variable 160
 - concepts 4
 - connecting, To-Do List 16
 - constructor, modifying in Class Editor 51
 - container, adding 231
 - container, adding to OASkillView* part 232
 - copying by dragging 131
 - copying from one part file to another 121
 - copying with clipboard 132
 - country-sensitive format 306
 - creating new, basic steps 117
 - creating, overview 101
 - defining 101
 - defining interface 101

part (*continued*)

- deleting from part file 123
- deleting from parts palette, steps 302
- deleting, free-form surface 132
- derived, using inheritance to build 86
- description, entering in Class Editor 50
- deselecting 131
- deselecting all 114
- designing for single purpose 83
- designing, OASearch 90
- displaying names 113
- displaying pop-up menu 131
- dragging and dropping, To-Do List 13
- editing text strings 133
- exporting information 116
- extending to span more than one cell on multicell canvas 223
- features, promoting 153
- fonts 142
- generating feature code 106
- generating feature code, OA Contractor part 106
- generating source code 273
- guidelines for placing on free-form surface 127
- importing 289
- information, importing 114
- interface, defining using part information files 287
- IVSequence*, free-form surface, To-Do List 212
- IVSequence*, placing and modifying, To-Do List 212
- learning to use 113
- listing in composite part 146
- manipulating 131
- moving dropped parts on multicell canvas 227
- moving to different part file 121
- moving to different part file in Class Editor 51
- moving, free-form surface 134
- multiple, changing tab and depth order 147
- new visual, To-Do List 8
- new, from base part 151
- nonvisual, adding to container 236
- nonvisual, OASearch 90
- notebook, adding 237
- OACContract, attributes 92
- OACContractor 91
- OASearch 93
- OASkill, attributes 93
- OASkillBase, actions 93
- object factory (IVBFactory*), placing and modifying, To-Do List 212
- object factory, connecting to variables, OASearch 267
- object factory, on free-form surface, To-Do List 212

part (*continued*)

- object factory, setting, OASearch 265
- opening 119
- opening multiple ones 120
- opening one 119
- opening settings notebook 138
- opening settings notebook for multiple parts 139
- overlying 127
- pasting with clipboard 132
- performing operations, Parts List window 148
- placing in To-Do List application window 10
- placing on free-form surface, introduction 124
- placing when not on parts palette 125
- positioning on grid 134
- primitive visual, creating own 290
- removing from parts palette, just added 302
- renaming in part file 124
- renaming, free-form surface 133
- resizing, To-Do List 12
- reusable, identifying 85
- reusing (inheritance) 86
- same, adding several copies 125
- saving changes on parts palette 302
- selected, adding to Visual Builder window 299
- selecting all 114
- selecting multiple 130
- selecting single 130
- selecting, introduction 129
- size and position 142
- sizes, matching 136
- sizing more than one 136
- sizing, free-form surface 136
- spacing in bounding box 137
- static text, placing another, To-Do List window 11
- static text, placing in To-Do List window 10
- sub-, spacing in Composers parts 137
- supplementary composite, as subpart 128
- tearing-off attribute 156
- unique icon for parts palette 298
- unique icon, Class Editor 53
- variable, adding, OASearch 266
- variable, connecting in OASearch 199
- variables, connecting to object factory parts, OASearch 267
- Visual Builder categories, table 128
- Visual Builder window 113
- visual, as dynamic instances, OASearch 264
- visual, changing 159
- visual, designing, OASearch 94
- visual, integrating into single application 263

- part code generation, source files created 275
- part features, promoting from Composition Editor 154
- part files
 - deselecting all 33
 - loading 31
 - seeing where located 34
 - selecting all 33
 - unloading 32
- part information files
 - creating 287
 - creating and importing, Direct-to-SOM objects 310
 - using to define part interface 287
- part interface
 - adding variables 162
 - defining using Part Interface Editor 102
- Part Interface Editor
 - Action page 69
 - adding attributes 58
 - Attribute page 57
 - defining part interface 102
 - deleting an action 72
 - Event page 63
 - introduction 56
 - Preferred page 75
 - Promote page 73
 - promoting features 74
 - setting defaults for actions 72
 - setting defaults for attributes 61
 - setting defaults for events 67
- parts
 - connecting to fill container 236
 - placing on free-form surface 124
- Parts List window, performing operations on parts 148
- parts palette
 - adding categories and parts, introduction 293
 - adding category, steps 296
 - adding part 299
 - adding part, vbb file loaded 301
 - Buttons category 45
 - Composers category 47
 - Data entry category 45
 - Frame extensions category 46
 - introduction 44
 - Lists category 46
 - Models category 48
 - Other category 48
 - preparing icons 293
 - saving changes 302
 - Sliders category 47
 - sticky 125
- parts palette (*continued*)
 - unique icon for part 298
- parts, sharing with others 283
- passing exceptions to message box 203
- pasting part with clipboard 132
- placing
 - another static text, To-Do List window 11
 - entry field in To-Do List window 11
 - IVSequence* part on free-form surface, To-Do List 212
 - IVSequence* part, To-Do List 212
 - list box in To-Do List window 11
 - object factory part (IVBFactory*) on free-form surface, To-Do List 212
 - object factory part (IVBFactory*), To-Do List 212
 - parts in To-Do List application window 10
 - parts on free-form surface 124
 - parts on free-form surface, introduction 124
 - parts when not on parts palette 125
 - push buttons, To-Do List window 12
 - static text part in To-Do List window 10
- pop-up menu, displaying 131
- populating collection in container 236
- portability publications 325
- position and size of part 142
- positioning parts
 - (aligning), free-form surface 136
 - (arranging) parts, introduction 134
 - parts on grid 134
- preferred features
 - adding 76
 - removing 77
 - removing all 77
 - showing inherited only 78
- preparing
 - generated files for compilation 277
 - icons for parts palette 293
 - resource DLL for OS/2 294
 - resource DLL, Windows 295
 - source code generation 273
- preparing generated files for compilation 277
- prerequisite knowledge xiii
- primary part, adding nonvisual parts, OASearch 263
- primary view, OAMain, OASearch 94
- primitive visual parts, creating own 290
- project, WorkFrame, starting Visual Builder 24
- Promote page
 - features 73
 - promoting features in Part Interface Editor 74
- Promote page, changing promoted features 74

- promoted features
 - changing 74
 - deleting 75
 - updating 74
- promoting
 - part's features 153
 - part's features from Composition Editor 154
- promoting features in Part Interface Editor 74
- publications, portability 325
- publications, related 325
- push button
 - Add, connecting to list box, To-Do List 16
 - aligning, To-Do List 15
 - centering, To-Do List 15
 - default 160
 - enabling in OASearch 201
 - enabling when entry field contains value, OASearch 200
 - Help, adding 257
 - label, mnemonic 159
 - matching widths, To-Do List 13
 - placing in To-Do List window 12
 - Remove, connecting to list box, To-Do List 18
 - spacing evenly, To-Do List 16
- push buttons for connection settings window 193
- putContractor action, modifying code 110
- putting parts on free-form surface, introduction 124

Q

- query windows, OASearch 96

R

- radio buttons in groups 150
- rci, resource file 303
- ready event 64
- ready event, example 65
- rearranging connections 196
- red-green-blue values 142
- redoing changes 157
- refreshID action, modifying code 110
- refreshing display 37
- related publications, VisualAge for C++ 325
- removing
 - all preferred features 77
 - categories and parts from parts palette, introduction 293
 - category or part just added 302
 - handler 141
 - preferred feature 77

- renaming
 - part in part file 124
 - part on free-form surface 133
- reordering connections 194
- replacing
 - and modifying list box, To-Do List 211
 - IListBox with ICollectionViewListBox, To-Do List 211
- resetting resource library, OASearch 270
- resizable windows, creating 217
- resource DLL
 - preparing, OS/2 294
 - preparing, Windows 295
- resource file
 - rci 303
 - translation 303
- resource header file (h) 304
- resource ID
 - starting, Class Editor 53
 - starting, guidelines 305
- reusable part
 - inheritance 86
 - parts, identifying 85
 - small components 89
- RGB values 142
- running
 - application, To-Do List 21
 - modified To-Do List application 215
 - new To-Do List application 215

S

- sample application
 - adding and setting object factory parts, introduction 265
 - adding basic visual parts to GUI 158
 - adding menu bar 244
 - adding menu choices to menus 246
 - adding message box 203
 - adding nonvisual parts to primary part 263
 - adding object factory parts 265
 - adding static parts 264
 - adding variable parts 266
 - adding visual parts as dynamic instances 264
 - attribute-to-attribute connections 199
 - combining all visual parts 263
 - completing menu bar 269
 - connecting menu bar to window 245
 - connecting menu choices 247
 - connecting nonvisual parts to variables 267
 - connecting object factory parts to variables 267
 - connecting to object factory parts 266

- sample application (*continued*)
 - connecting variable part 199
 - connections from Edit submenu 269
 - connections from View submenu 269
 - constructing interface 157
 - designing nonvisual parts 90
 - designing visual parts 94
 - enabling push buttons 201
 - enabling push buttons when entry field contains value 200
 - enabling window to be cleared of all entry values 204
 - event-to-action connections 202
 - modal windows, connections for 267
 - modeless windows, connections 268
 - naming convention, OACContractor part 91
 - nonvisual parts in oanonvis.vbb 90
 - OACContract part's attributes and actions 92
 - OACContractor part 91
 - OACContractor part's attributes 91
 - OASearch Welcome window with menus 243
 - OASkill part's attributes 93
 - OASkillBase part's actions 93
 - Opportunities Abound Contract Information window 98
 - Opportunities Abound Contractor Information window 96
 - Opportunities Abound Databases window 94
 - Opportunities Abound General Information window 95
 - Opportunities Abound Skill Information window 98
 - primary view, OAMain 94
 - query windows 96
 - resetting resource library 270
 - setting object factory part 265
 - simulated database 90
 - steps for building 99
 - visual parts in oawin.vbb 90
- sample help source code 252
- selecting
 - all part files 33
 - all parts 114
 - connections 197
 - multiple connections, in OS/2 197
 - multiple connections, in Windows 197
 - multiple part 130
 - parts, introduction 129
 - single connection 197
 - single part 130
- set member function, attribute 59
- setting
 - group 150
 - object factory parts, OASearch 265
- setting (*continued*)
 - object factory parts, OASearch, introduction 265
 - tab stop 150
 - working directory 36
- setting container titles 232
- setting defaults
 - actions, Part Interface Editor 72
 - attributes, Part Interface Editor 61
 - events, Part Interface Editor 67
- setting tab
 - order 148
 - stop for each entry field 150
- setting up
 - containers 232
 - icon container columns 234
 - page and tab for notebook 240
 - string container columns 235
- settings
 - activating changes 144
 - changing 159
 - changing for attribute-to-attribute connections 190
 - changing for multicell canvas 227
 - changing, action as target 191
 - changing, custom logic as target 194
 - changing, member function as target 192
 - for connection, changing 189
 - part, changing 138
- settings notebook
 - adding rows and columns on multicell canvas 229
 - deleting rows and columns on multicell canvas 229
 - generic 144
 - navigating 139
 - opening for multiple parts 139
 - opening for part 138
- settings pages, descriptions 139
- sharing parts with others 283
- showing
 - Composers part (toggle to hiding) 135
 - connections 195
 - full file names 34
 - grid 135
 - inherited preferred features only 78
 - part names 113
 - type list 35
- simulated database, OASearch 90
- single part, selecting 130
- size and position of part 142
- Size/Position page 142
- sizes, part, matching 136

- sizing
 - adjusting for translated text using canvases 305
 - more than one part 136
 - parts, free-form surface 136
- sizing tools on tool bar 44
- Sliders category 47
- SOM objects
 - bypassing limitations 313
 - creating and importing part information file 310
 - Direct-to- 309
 - introduction 309
 - Visual Builder application 312
- source code
 - C++, generating for individual parts 274
 - C++, generating for main() function 276
 - generating for parts and applications 273
 - generating for your main() function 20
 - generation, preparing 273
 - sample help 252
 - yours, generating for visual part 20
- source code generation
 - concepts 5
 - main() procedure, To-Do List 214
 - visual part, To-Do List 214
- source feature, changing name 192
- source files
 - created, generation of main() function 276
- source files, created, part code generation 275
- source of connection
 - changing 198
 - determining 171
- spacing
 - grid 135
 - parts in bounding box 137
 - push buttons evenly, To-Do List 16
 - subparts in Composers parts 137
- spacing part
- special notices xi
- specifying
 - additional libraries in make file 278
 - container type and layout 233
 - debug options for compiler and linker programs 278
 - default push button 160
 - defaults for parameters 177
 - files for application build 55
 - fonts 142
 - grid spacing 135
 - item types in list box, To-Do List 211
 - library file in Class Editor 52
 - names of code generation files in Ckass Editor 54
- specifying (*continued*)
 - notebook layout 239
 - object types sequence, To-Do List 213
 - object types, object factory can create, To-Do List 212
 - option to generate Browser information 278
 - parts with country-sensitive format 306
 - push button label, mnemonic 159
 - size and position of part 142
 - starting resource ID in Class Editor 53
 - starting resource IDs, guidelines 305
 - unique icon for part added to parts palette 298
 - unique icon for part in Class Editor 53
 - you constructor code in Class Editor 52
 - your destructor code in Class Editor 52
- square bracket symbols 161
- starting
 - Visual Builder for To-Do List 8
 - Visual Builder, C/C++ window (OS/2) 23
 - Visual Builder, introduction 23
 - Visual Builder, MS/DOS command window (Windows) 23
 - Visual Builder, VisualAge C++ folder (Windows) 24
 - Visual Builder, VisualAge C++ Tools folder (OS/2) 24
 - Visual Builder, WorkFrame project 24
- starting resource ID, Class Editor 53
- starting resource IDs, guidelines 305
- static parts, adding, OASearch 264
- steps
 - (high-level) for creating object-oriented application 84
 - creating new parts 117
 - creating To-Do List application 7
 - developing code outside Visual Builder 112
- sticky selected 125
- storing parts in files, vbb file 31
- string container columns, setting up 235
- style guidelines for groups and tab stops 150
- Styles settings page 140
- submenu
 - Edit, connections from, OASearch 269
 - View, connections from, OASearch 269
- subpart
 - fly-over help 258
 - hover help 258
 - spacing in Composers parts 137
- subpart, fly-over help 258
- subpart, hover help 258
- summary of connection types 170
- supplementary composite parts, as subparts 128
- supplying
 - initial field values using code strings 143

supplying (*continued*)

- parameter value using connection 175
- parameter value using constant value 177
- parameter values for incomplete connections 175

symbols

- editors, fast-path 39
- square bracket 161

T

tab and depth order for multiple parts, changing 147

tab and page for notebook, setting up 240

tab order

- changing 149
- setting 148

tab stop

- setting 150
- setting for entry field 150

tab stops, style guidelines 150

table

- connection types you can change 199
- optimal sizes for components 89
- types of connections, summary 170
- Visual Builder categories 128

target class, changing name 192

target member function, changing 193

target of connection

- changing 198
- determining 171

tear-off attribute, example 161

tearing off attribute 156

text strings, editing 133

tips for using Visual Builder 315

title, changing, To-Do List application window 10

To-Do List

- aligning parts 12
- aligning static text, entry field, list box 14
- aligning, push buttons 15
- building application 21
- building modified application 215
- building new application 215
- centering entry field and list box 14
- centering push buttons 15
- changing title in application window 10
- connecting Add push button to list box 16
- connecting parts 16
- connecting Remove push button to list box 18
- creating new visual part 8
- creating, introduction 7
- displaying objects 207

To-Do List (*continued*)

- dragging and dropping parts 13
- exiting Composition Editor 22
- finished 7
- generating C++ code 20
- generating source code for visual part 214
- generating source code, main() procedure 214
- ICollectionViewComboBox to display objects, introduction, To-Do List 207
- ICollectionViewListBox to display objects, introduction, To-Do List 207
- item types in list box 211
- list boxes to display objects, To-Do List 207
- making new connections 213
- matching widths of list box and entry field 12
- matching widths of push buttons 13
- modifying IVSequence* part 212
- modifying list box 211
- modifying object factory part (IVBFactory*) 212
- object factory part (IVBFactory*) on free-form surface 212
- object types, object factory can create 212
- placing another static text part in window 11
- placing entry field in window 11
- placing IVSequence* part 212
- placing IVSequence* part on free-form surface 212
- placing list box in window 11
- placing object factory part (IVBFactory*) 212
- placing parts in application window 10
- placing push buttons in window 12
- placing static text part in window 10
- replacing IListBox with ICollectionViewListBox 211
- replacing list box 211
- resizing application window 13
- resizing parts 12
- running application 21
- running modified application 215
- running new application 215
- spacing push buttons evenly, To-Do List 16
- specifying type of objects in sequence 213
- starting Visual Builder 8
- steps for creating 7
- ToDoItem nonvisual part 208
- ToDoList part, copying 208
- ToDoItem nonvisual part, creating 208
- toggling (showing and hiding) grid 135
- toggling grid, showing and hiding Composers part 135
- tool bar
 - Align bottom tool 137
 - Align center tool 136

tool bar (*continued*)

- Align left tool 136
- Align middle tool 136
- Align right tool 137
- Align top tool 136
- alignment tools 43
- connection tools 42
- Distribute horizontally tool 137
- Distribute vertically tool 137
- distribution tools 44
- grid tools 43
- introduction 41
- Match height tool 136
- sizing tools 44

tools

- Align bottom 137
- Align center 136
- Align left 136
- Align middle 136
- Align right 137
- Align top 136
- alignment 43
- connection 42
- Distribute horizontally 137
- Distribute vertically 137
- distribution 44
- grid 43
- Match height 136
- sizing tools 44

Tools folder, VisualAge C++ (OS/2), starting Visual Builder 24

tracing Visual Builder connections, setting up 278

trademarks xi

translated text, canvases to adjust size 305

translation, resource files 303

type list, showing 35

types

- connection, introduction 163
- items in list box, To-Do List 211
- menus and menu items 243
- objects in sequence, To-Do List 213
- objects the object factory can create, To-Do List 212
- variable's, changing 161

U

undoing changes 157

unique icon for part added to parts palette 298

unloading mouse pointer 125

unloading part files 32

updating

- actions in Part Interface Editor 72
- attribute 61
- event 67
- promoted feature 74
- using existing C++ code 101
- using inheritance 86
- using parts, learning 113

V

values, setting for container 232

variable

- adding to composite part 160
- adding to free-form surface 160
- connecting nonvisual parts to, OASearch 267
- part, connecting in OASearch 199
- parts, connecting object factory parts to, OASearch 267
- type, changing 161

variables, adding to part interface 162

vbb files

- adding part, vbb file loaded 301
- copying part from one part file to another 121
- deleting part 123
- moving part to different part file 121
- oanonvis, nonvisual parts, OASearch 90
- oawin, visual parts, OASearch 90
- renaming part 124
- storing parts in files 31

View submenu, connections from, OASearch 269

Visual Builder

- adding code created outside of 112
- application, Direct-to-SOM objects 312
- application, readying for others 273
- benefits 3
- creating To-Do List application, introduction 7
- definition 3
- developing basic application 79
- editors, introducing 39
- exiting, To-Do List 22
- hints and tips for using 315
- key concepts 4
- setting up to generate HTML documentation 274
- setting up to generate make files 273
- starting for To-Do List 8
- starting from C/C++ window 23
- starting from MS/DOS command window (Windows) 23
- starting, introduction 23
- starting, VisualAge C++ folder (Windows) 24

- Visual Builder (*continued*)
 - starting, VisualAge C++ Tools folder (OS/2) 24
 - starting, WorkFrame project 24
 - table of categories 128
 - using existing C and C++ code 287
- Visual Builder window
 - adding selected part 299
 - introduction 29
 - parts 113
- visual construction, overview 113
- visual part
 - adding as dynamic instances, OASearch 264
 - adding basic to GUI, OASearch 158
 - changing labels 159
 - changing settings 159
 - designing, OASearch 94
 - generating source code, To-Do List 214
 - integrating into single application 263
 - new, creating, To-Do List 8
 - oawin.vbb, visual parts, OASearch 90
 - primitive, creating own 290
 - yours, generating source code 20
- VisualAge C++ folder (Windows), starting Visual Builder 24
- VisualAge C++ Tools folder (OS/2), starting Visual Builder 24
- VisualAge for C++ publications 325
- window (*continued*)
 - Opportunities Abound Skill Information, OASearch 98
 - query, OASearch 96
 - resizable, creating 217
 - using multicell canvas to build 217
 - Visual Builder window 29
 - Visual Builder, adding selected part 299
 - Visual Builder, parts 113
- Windows
 - deselecting connections 198
 - preparing resource DLL 295
 - selecting multiple connections 197
- WorkFrame project, starting Visual Builder 24
- WorkFrame, compiling and linking applications 278
- working directory, setting 36
- working with parts, Visual Builder window 113
- writing portable help 250

W

- Welcome window with menus, OASearch 243
- window
 - application help 254
 - application, resizing, To-Do List 13
 - C/C++ (OS/2), starting Visual Builder from 23
 - connecting menu bar, OASearch 245
 - enabling to be cleared of all entry values, OASearch 204
 - help for factory-generated frame windows 256
 - modal, connections for, OASearch 267
 - modeless, connections, OASearch 268
 - MS/DOS command window (Windows), starting Visual Builder from 23
 - OASearch Welcome window with menus 243
 - Opportunities Abound Contract Information, OASearch 98
 - Opportunities Abound Contractor Information, OASearch 96
 - Opportunities Abound Databases, OASearch 94
 - Opportunities Abound General Information, OASearch 95

Communicating Your Comments to IBM

IBM VisualAge for C++ for Windows
Visual Builder User's Guide
Version 3.5

Publication No. S33H-5034-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
 - United States and Canada: 416-448-6161
 - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
 - Internet: torrcf@vnet.ibm.com
 - IBMLink: [toribm\(torrcf\)](mailto:toribm(torrcf)@vnet.ibm.com)
 - IBM/PROFS: [torolab4\(torrcf\)](mailto:torolab4(torrcf)@vnet.ibm.com)
 - IBMMAIL: [ibmmail\(caibmwt9\)](mailto:ibmmail(caibmwt9)@vnet.ibm.com)

Readers' Comments — We'd Like to Hear from You

IBM VisualAge for C++ for Windows
Visual Builder User's Guide
Version 3.5

Publication No. S33H-5034-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
S33H-5034-00



Cut or Fold
Along Line

Fold and Tape

Please do not staple

Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape

Please do not staple

Fold and Tape

S33H-5034-00

Cut or Fold
Along Line

IBM®

Part Number: 33H5034

Printed in U.S.A.

S33H-5034-00



33H5034

