

IBM VisualAge for C++ for Windows

S33H-5036-00

Building VisualAge for C++ Parts for Fun and Profit

Version 3.5

IBM

IBM VisualAge for C++ for Windows

S33H-5036-00

Building VisualAge for C++ Parts for Fun and Profit

Version 3.5

Note!

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

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.

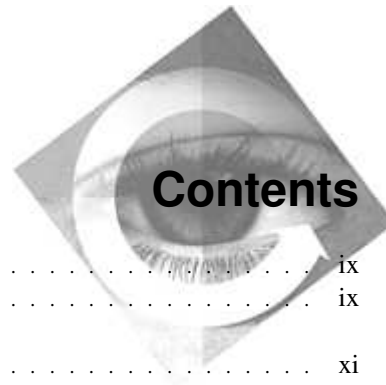
A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

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	ix
Trademarks	ix
About this Book	xi
Who Should Use This Book	xi
How to Use This Book	xi
Highlighting Conventions	xii
How to Read Syntax Diagrams	xii
Related Publications	xiii
Visual Builder Books	xiv
User Interface Programming and Design Books	xiv
Object-Oriented Programming and Design Books	xiv
How to Get Help	xv
Getting Help Inside VisualAge for C++	xv
Getting Help from the Command Line	xvi
Getting Help for a Keyword or Construct	xvi
Online Documents Available in VisualAge for C++	xvi
<hr/>	
Part 1. Introducing C++ Construction from Parts	1
Chapter 1. What Is C++ Construction from Parts?	3
The Origins of C++ Construction from Parts	4
The Benefits of Using Parts	5
What Is a C++ Part?	5
How Parts and Classes Are Related	6
How You Can Connect Parts	7
Sources of Parts	8
Chapter 2. Object Technology Overview	9
The Application Segmentation Paradigm	9
Separation of Model Objects from View Objects	11
Segmentation within the Model	13
Chapter 3. The Architecture of C++ Construction from Parts	17
The Origins of the C++ Construction from Parts Architecture	17
Architecture Characteristics	19
Part Interface Architecture	19
Access to Part Properties	19
Access to Part Behavior	22

Notification of Changes to Parts	22
Kinds of Parts Supported in Visual Builder	23

Part 2. Building Basic Parts 27

Chapter 4. Designing a C++ Part	29
Design Guidelines	29
Conventions to Consider	30
Naming Parts	30
Naming Actions, Attributes, and Events	30

Chapter 5. Implementing a C++ Part	31
Positioning a Part within the Class Hierarchy	31
Implementing Events	32
Implementing Attributes	33
Defining Get Member Functions	33
Defining Set Member Functions	34
Attribute Notification IDs	34
Implementing Actions	35
Implementing Constructors	36
Implementing Destructors	36
Implementing Assignment Operators	37
Creating Composers and Primitive Visual Parts	37
Implementation Checklists	38

Chapter 6. Describing Part Interfaces in Part Information Files	41
Part Information Syntax	42
VBBeginPartInfo Statement for a Part	43
VBParent Statement for a Part	43
VBIncludes Statement for a Part	43
VBPartDataFile Statement for a Part	44
VBLibFile Statement for a Part	44
VBComposerInfo Statement for a Part	44
VBConstraints Statement for a Part	46
VBConstructor Statement for a Part	47
VBEvent Statement for a Part	48
VBAttribute Statement for a Part	48
VBAction Statement for a Part	49
VBPreferredFeatures Statement for a Part	50
VB Statement for a Part	51
VBEndPartInfo Statement for a Part	51
Sample Part Information for IAddress	51
Class Information Syntax	52

VBBeginPartInfo Statement for a Class	53
VBParent Statement for a Class	53
VBIncludes Statement for a Class	53
VBPartDataFile Statement for a Class	54
VBComposerInfo Statement for a Class	54
VBConstraints Statement for a Class	54
VBConstructor Statement for a Class	55
VBAction Statement for a Class	55
VBAttribute Statement for a Class	56
VBPreferredFeatures Statement for a Class	57
VB Statement for a Class	57
VBEndPartInfo Statement for a Class	57
Sample Class Information for IRange	57
Function Group Information Syntax	58
VBBeginPartInfo Statement for Function Groups	59
VBIncludes Statement for Function Groups	59
VBPartDataFile Statement for Function Groups	59
VBComposerInfo Statement for Function Groups	60
VBConstraints Statement for Function Groups	60
VBAction Statement for Function Groups	60
VB Statement for Function Groups	61
VBPreferredFeatures Statement for Function Groups	61
VBEndPartInfo Statement for Function Groups	61
Sample Function Group Information	61
Enumeration Information Syntax	62
VBBeginEnumInfo Statement	62
VBIncludes Statement for an Enumeration	62
VBPartDataFile Statement for an Enumeration	63
VBEnumerators Statement	63
VB Statement for an Enumeration	63
VBEndEnumInfo Statement	64
Sample Alignment Enumeration for IEntryField	64
Type Definition Information Syntax	64
VBBeginTypedefInfo Statement	65
VBIncludes Statement for a Type Definition	65
VBPartDataFile Statement for a Type Definition	65
VB Statement for a Type Definition	66
VBEndTypedefInfo Statement	66
Sample Type Definition Information	66
Other Syntax Examples	66
 Chapter 7. Sharing Parts with Others	 67
Providing part files (.vbb)	67

Providing Part Information Files (.vbe)	68
---	----

Part 3. Understanding and Using Event Notification	69
---	----

Chapter 8. The IBM Class Notification Framework	71
Notifiers and Observers	71
Notification Protocol	73
IBM C++ Notification Class Hierarchy	74

Chapter 9. C++ Code to Enable Notification	77
Code for an Observer Class	77
Code for a Notifier Class	79
Sample Notification Flow	79

Part 4. Appendixes	81
-------------------------------------	----

Appendix A. IBM Open Class Library Conventions	83
File Extensions	83
File Names	84
Class Names, Function Names, and Data Member Names	84
Enumerations	85
Function Return Types	85
Function Arguments	86
Feature Names	86
Notification IDs	86
Other Standards	87

Appendix B. Code Listings	89
INotifier Header Code	90
IStandardNotifier Header Code	93
IObserver Header Code	96
INotificationEvent Header Code	99
IButton Header Code	102
Sample IAddress Part	107
IAddress Header Code (iadd.hpp)	107
IAddress Source Code (iadd.cpp)	110
IAddress Test Code (iadd.cxx)	117

Appendix C. VisualAge for C++ Classes by Include and Data File	121
---	-----

Glossary	131
---------------------------	-----

Bibliography 139

The IBM VisualAge for C++ Library 139

C and C++ Related Publications 139

 Non-IBM Publications 139

Index 141



Notices

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—a quick and easy way to create an application from reusable parts!

Reusable parts can be thought of as the building blocks of the applications you are constructing with Visual Builder. As with any construction project, you need to either buy or build the parts you need. This book describes how to design and construct basic building-block parts using the C++ programming language.

If you are familiar with VisualAge for Smalltalk, you already know about constructing applications from parts. The details of how to implement C++ parts is a little different than implementing IBM Smalltalk parts, but the overall concepts and architecture are the same.

Who Should Use This Book

This book is designed for experienced C++ programmers who need to build parts for use with Visual Builder. It can also be useful to Visual Builder users who need a deeper understanding of how Visual Builder works or who want an introduction to C++ parts.

How to Use This Book

The book is divided into the following parts:

- Part 1, “Introducing C++ Construction from Parts” on page 1, provides an introduction to the key concepts and architecture.
- Part 2, “Building Basic Parts” on page 27, contains the specifics on creating basic parts for the Visual Builder environment. After reading this section, you should understand how to implement C++ parts.
- Part 3, “Understanding and Using Event Notification” on page 69, provides information about advanced topics, including an overview of the IBM C++ notification framework and an explanation of how to connect parts together.
- The last part of this document consists of appendixes that provide conventions to consider when creating C++ parts, reference information for key IBM Class header files, and some sample parts.

Examples appear along the way to illustrate the principles of common interfaces among parts.

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 IAddress 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 Read Syntax Diagrams

This book uses syntax diagrams to show the syntax of some of the commands described.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The ►— symbol indicates the beginning of a command.

The —► symbol indicates that the command is continued on the next line.

The ►— symbol indicates that a command is continued from the previous line.

The —►◀ symbol indicates the end of a command.

Diagrams of syntactical units other than complete commands start with the ►— symbol and end with the —► symbol.

Note: In the following diagrams, COMMAND represents a command.

- Required items appear on the horizontal line (the main path).

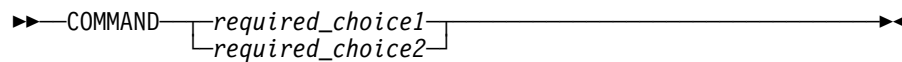
►—COMMAND—*required_item*—►◀

- Optional items appear below the main path.

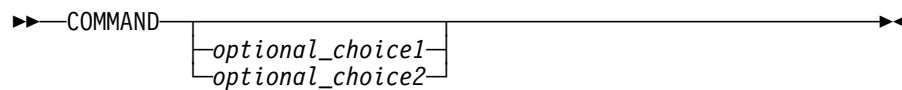
►—COMMAND—
 └─*optional_item*—┘—►◀

- If you can choose from two or more items, they appear vertically in a stack.

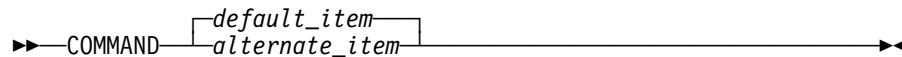
If you *must* choose one of the items, one item of the stack appears on the main path.



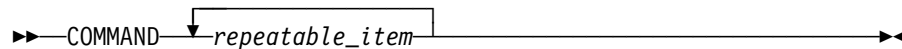
If choosing one of the items is optional, the entire stack appears below the main path.



The item that is the default appears above the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or you can repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `pragma`).

Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Note: The white space is not always required between items, but it is recommended that you include at least one blank between items unless otherwise specified.

Related Publications

This product provides a number of online guides and references that we hope you will find helpful as you develop applications. This information includes User's Guides, References, and How Do I help that gives you specific instructions for performing common tasks. You can get to this online information from the Information folder inside the main product folder. You can also get to it from the **Help** menu in any of the components of the product.

You can consult the following books for additional information about Visual Builder, user interface software design, and object-oriented software design:

Visual Builder Books

- *VisualAge for C++: Visual Builder User's Guide* (S33H-5034-00)
- *VisualAge for C++: Visual Builder Parts Reference* (S33H-5035-00)

User Interface Programming and Design Books

In addition to the specific Visual Builder books listed previously, the following books help you give a professional appearance to your part's user interfaces:

- *Object-Oriented User Interface Design – IBM Common User Access Guidelines*, Carmel, IN: Que Corporation, 1993. ISBN: 1-56529-170-0.
- *Volume 6A: Motif Programming Manual*, 2nd Edition, Sebastopol, CA: O'Reilly & Associates, Inc., 1993.
- *Volume 6B: Motif Reference Manual*, Sebastopol, CA: O'Reilly & Associates, Inc., 1993. ISBN: 1-56592-038-4.

Object-Oriented Programming and Design Books

In addition to the specific Visual Builder books listed previously, you might want to refer to a book about application design in the object-oriented and the cooperative-processing environments. Some of the available books are as follows:

- Booch, Grady, *Object-Oriented Design with Applications*, Redwood City, CA: Benjamin/Cummings Publishing Company, 1991. ISBN: 0-8053-0091-0.
- Coplien, James O., *Advanced C++ Programming Styles and Idioms*, Addison Wesley Publishing Company, 1992. ISBN: 0-201-54855-0.
- Cox, Brad J., *Object-Oriented Programming: An Evolutionary Approach*, Reading, MA: Addison Wesley Publishing Company, 1987. ISBN: 0201103931.
- Stroustrup, Bjarne, *The Design and Evolution of C++*, Murray Hill, NJ: Addison Wesley Publishing Company, 1994. ISBN: 0-201-54330-3.
- Tibbetts, John et al., *Building Cooperative Processing Applications Using SAA*, New York, NY: John Wiley & Sons, Inc., 1992. ISBN: 0-471-55485-5.
- Wirfs-Brock, Rebecca et al., *Designing Object-Oriented Software*, Englewood Cliffs, NJ: Prentice Hall, 1990. ISBN: 0-13-629825-7.

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++” 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 xvi.

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. Introducing C++ Construction from Parts

Just about any construction project that you can imagine involves assembling standard or customized basic parts into more complex parts. This process is repeated until the final product is complete. If you are building a birdhouse, these basic parts might be lumber, screws, wire, and paint. Some parts, such as screws and paint, can be used in their standard form. Other parts, such as lumber and wire, come in a standard form but need to be customized, or cut, before you use them.

If reusable software parts are available, building a software application can be conceptually similar to building a birdhouse. You can use the software parts as-is, as you would with the screws, or tailor the software parts to your exact needs, as you would with the lumber.

In both scenarios, you need to decide whether to build or buy the basic parts for your construction project. If you decide to build some basic software parts, this book guides you through the process (you will need to read about creating the parts of a birdhouse somewhere else). If you decide to buy the software parts, this book can help you to choose well-constructed software parts.

Chapter 1. What Is C++ Construction from Parts?	3
The Origins of C++ Construction from Parts	4
The Benefits of Using Parts	5
What Is a C++ Part?	5
How Parts and Classes Are Related	6
How You Can Connect Parts	7
Sources of Parts	8
 Chapter 2. Object Technology Overview	 9
The Application Segmentation Paradigm	9
Separation of Model Objects from View Objects	11
Segmentation within the Model	13
 Chapter 3. The Architecture of C++ Construction from Parts	 17
The Origins of the C++ Construction from Parts Architecture	17
Architecture Characteristics	19
Part Interface Architecture	19
Kinds of Parts Supported in Visual Builder	23

Chapter 1.

What Is C++ Construction from Parts?

C++ construction from parts is a technology for application development in which applications are built from existing, reusable software components called *parts*. Parts provide a wide range of capability, from very simple function through complete, highly sophisticated applications. Figure 1 shows a few examples.

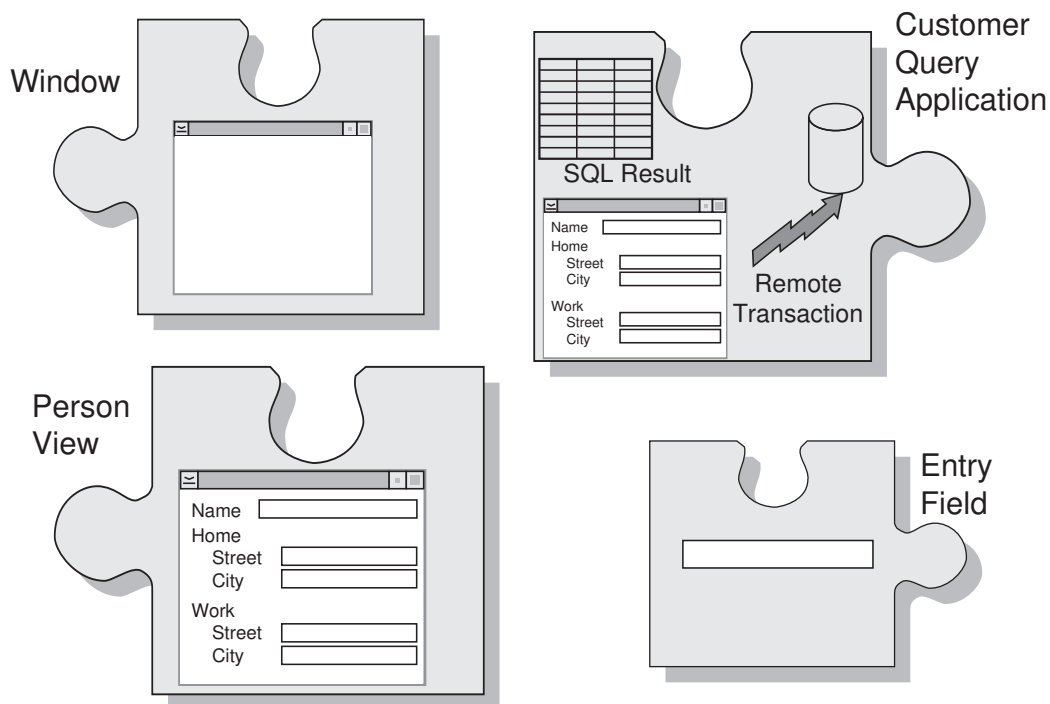


Figure 1. The Range of Primitive and Composite Parts

An entry field, a window, and a data array are examples of *primitive* parts. You combine primitive parts to form more complex *composite* parts, such as a person view. You can then extend this approach by combining primitive parts with composite parts to create entire applications, such as a customer query application.

In general, parts are either *visual* or *nonvisual*. In the previous example, the entry field, window, and person view are visual parts. The data array is a nonvisual part. For more information about types of parts, see "Kinds of Parts Supported in Visual Builder" on page 23.

The Origins of C++ Construction from Parts

The C++ construction from parts technology is just becoming popular in the software industry, but it is based on well-established techniques from other industries, such as manufacturing. Figure 2 compares the manufacturing process of constructing a computer system and the software process of constructing an application.

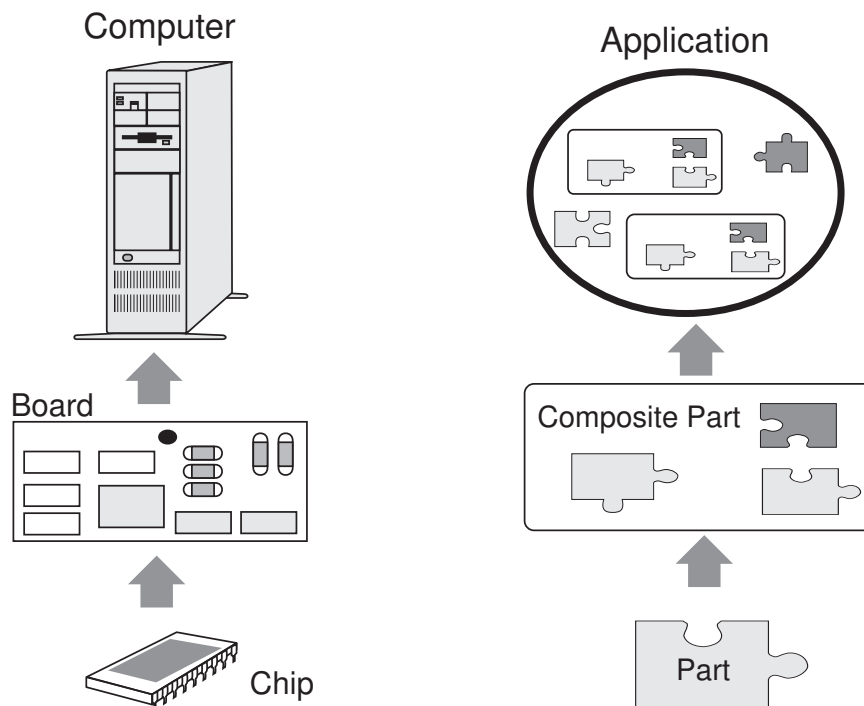


Figure 2. C++ Construction from Parts Technology

Just as electronic chips can be combined to form a functional board and functional boards can be combined to form a computer, software parts can be combined to form a composite part and composite parts can be combined to form an application.

To build a new computer today, you probably would not consider designing and constructing every single electronic and mechanical component from raw materials. Likewise, rather than always designing and developing new code for your applications, you can now use available standard parts. Now the software application development industry can realize the same benefits of reduced cycle time and increased quality that have become so prevalent in the manufacturing industry.

The Benefits of Using Parts

The benefits that you and your company can realize from using the C++ construction from parts technology to build applications include the following:

- Reduced application development cost through division of labor.

Application developers are able to focus their expertise on rapid development of superior solutions for their users by tailoring reusable parts and assembling them into applications. Meanwhile, part designers can concentrate on developing new and innovative parts to meet the needs of the application developers.

- Enhanced application quality and reliability.

Reusing existing parts reduces the chance of introducing errors when building applications. As parts are reused and refined, they become the solid building blocks for your applications.

- Reduced cycle time to respond to users' needs.

Building an application prototype from a library of pre-existing parts allows you to rapidly verify your users' requirements. You can then smoothly and quickly extend this prototype into a production application.

Your success in using this technology depends on the availability of easy-to-use construction tools, standard interface protocols to enable the tools and parts to interoperate, and an ever-growing library of standard, increasingly powerful parts to be reused.

What Is a C++ Part?

A C++ part is a software object implemented as a C++ class with some special characteristics:

- It supports a simple, standard interface protocol.

This protocol supports the interconnection of parts to form higher-function parts or entire applications. You can think of this protocol as being like the "innies" and "outies" on puzzle pieces that enable them to be interlocked into larger portions of the puzzle.

The part interface is composed of three distinct features: *attributes*, *actions*, and *events*. These features correspond to a natural way of viewing parts (and objects in general) in terms of what properties (*attributes*) they have, what behaviors (*actions*) they can perform, and what unsolicited information (*events*) they can notify other parts about. Figure 3 on page 6 shows an example of a part interface.

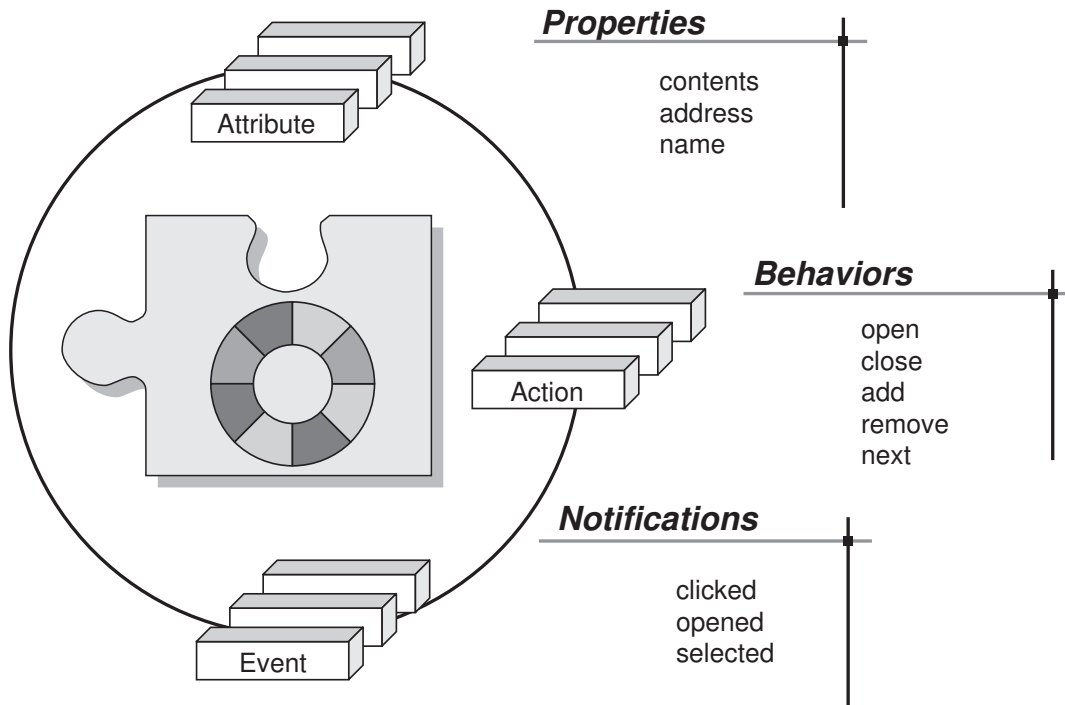


Figure 3. A Part Interface

- It can extend the functions of application building tools.

The part itself can extend the construction environment by providing tool functions specifically customized to the part. Examples of these tool functions are icons, automated view builders, and attribute value initialization. You can think of the picture on the top of real jigsaw puzzle pieces as a tool extension—it enhances the ability of the tool (you) to complete the job (putting this particular puzzle together).

How Parts and Classes Are Related

If you are familiar with object-oriented concepts, you have probably noticed that a part is very much like an object in object-oriented programming. In fact, parts and classes are closely related because C++ classes form the underlying software implementation of parts.

It might seem to you that we are not always talking about the same thing when we talk about a part. This is because the word *part* can mean different things at different times.

Part is most often used as shorthand for *part class*. A part class is nothing more than an object class with some special characteristics (such as a class method that defines the part interface of the part). A part class is used as a form for creating part instances. You can develop a new part, or enable an existing class to become a part, by supplying support for these characteristics in addition to the normal operations of the object class.

Part is also used as shorthand for *part instance*. A part instance is a particular object created from a part class. In C++, you might code an expression such as

```
BananaClass * aBananaInstance = new BananaClass();
```

to create a particular instance of a part class. In a visual programming environment, you might create a particular part instance by picking a Banana part class from a palette of part classes and dropping it on a free-form surface.

You can tell which kind of part we are talking about from the context in which the word appears. When we talk about parts on a palette or parts that you create by writing code, we are referring to part classes. When we talk about parts on a free-form surface or parts that are connected together to form an application, we are referring to part instances.

Chapter 3, “The Architecture of C++ Construction from Parts” on page 17 provides the blueprint for adding the characteristics that turn an object class into a part class. It sets the stage for you to build your own part classes.

How You Can Connect Parts

Visual Builder's Composition Editor enables you to connect the following kinds of parts:

- A visual part to a visual part
- A nonvisual part to a nonvisual part
- A visual part to a nonvisual part

You can also make the following kinds of connections:

- **Event-to-action connections** start an action when a certain event occurs. A variation of this, the attribute-event-to-action connection, starts an action when a certain attribute event (for example, attribute changing value) occurs.

The action can have parameters; these parameters are attributes of the connection. You can connect a single event to many actions. You can also specify the order in which the actions are processed. Connecting the clicked event from a push button to the clear action on an entry field is an example of an event-to-action connection. Connecting the street attribute event from an address to the enable

action on a **Save** push button is an example of an attribute-event-to-action connection.

- **Attribute-to-attribute connections** link two data values so that they always stay the same.

You can connect a single attribute to many attributes. Connecting the street attribute of an address part to the text (contents) attribute of an entry field part is an example of an attribute-to-attribute connection.

Sources of Parts

There are many possible sources for parts that you can combine to build new parts or complete applications.

Visual Builder is shipped with a collection of prefabricated parts. These parts are the basic visual and nonvisual building blocks of an application. Examples are entry fields, push buttons, and data arrays.

Software providers will be creating additional parts. These parts might provide additional system support, such as facsimile transmission or e-mail access, common business functions, such as charting or report writing, or industry-specific functions, such as hospital patient charting or wholesale distribution.

Finally, if none of the available parts fulfills your needs, you (or someone in your organization) can create new parts either by modifying existing parts to add functions, by modifying existing C++ classes to enable them as parts, or by building parts just as you would any other C++ class.



Chapter 2. Object Technology Overview

As the cost of processing power has decreased, enterprises have taken the opportunity to make people more effective. One way of increasing people's effectiveness is to make the computer system into an extension of their everyday business environment.

In the batch and transaction environments, you were presented with lists of functions that you could use. These functions did not necessarily correspond to the problem you were trying to solve. Rather, they were the application designer's idea of the solutions to the problem you were supposed to have.

With computer power moving to the desktop, a new approach to building applications has emerged. This approach provides you with the impression that the computer is able to deal with the things that are common to your business, such as calendars, notepads, invoices, bank accounts, or a wastebasket. Your desktop computer becomes an extension of your real world.

Designing applications to operate in this new environment can be challenging. Many books are available to guide you in this endeavor. We only touch lightly on the design issues here and suggest that you consult a book devoted to the subject if you need in-depth knowledge (see "Related Publications" on page xiii).

The Application Segmentation Paradigm

Figure 4 on page 10 shows the structure of an application developed using the guidelines presented in this book. This structure follows a common application segmentation paradigm in the cooperative processing environment, where an application is divided into three segments: user interface, business logic, and data access.

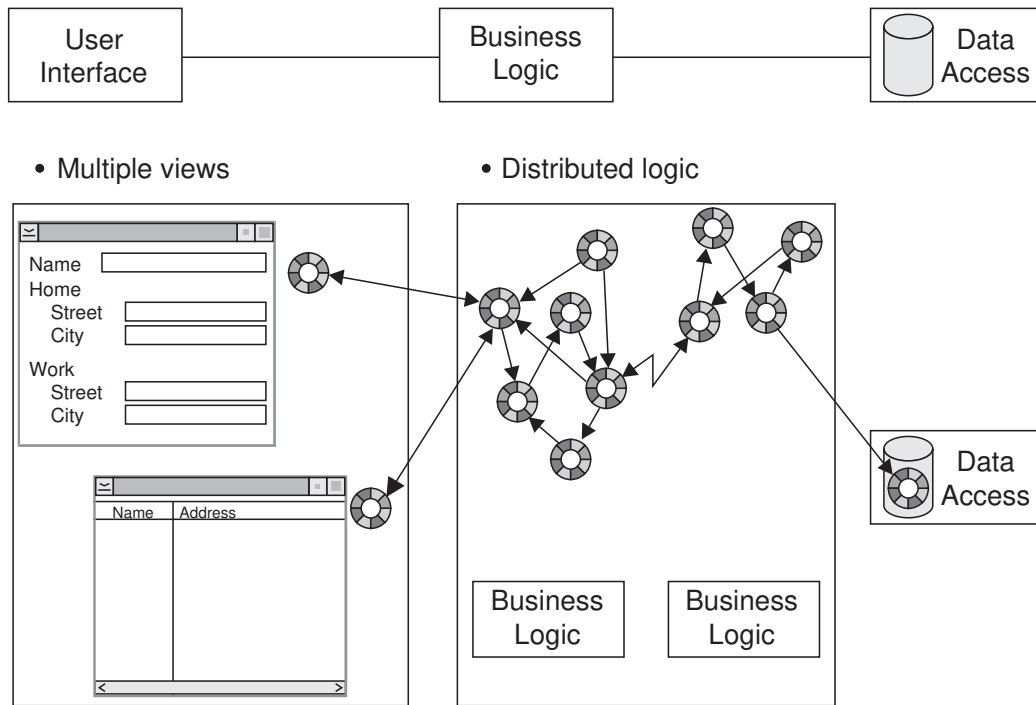


Figure 4. Overall Application Structure

The user interface segment defines how the user and the system interact. It presents information to the user and accepts input from the user on behalf of the business logic. Because it does not implement any of the business logic behavior, it can be updated, or completely replaced, without affecting the business logic. We refer to this user interface function as a *view*.

The business logic segment implements the real-world objects of the application. It defines the behaviors of these objects and their interrelationships without consideration for how they are presented to users or how users interact with them. We refer to the business logic segment as a *model*. The implementation of the model can be totally contained in a single computer, or it can be distributed among several computers using available inter-computer communication mechanisms to interconnect the distributed components of the model.

The third segment of an application is data access. From the application builder's perspective, this segment can be thought of as simply an extension of the model. Because of this, we do not discuss it in this book. You can refer to one of the books on cooperative processing environments (see "Object-Oriented Programming and Design Books" on page xiv) for a detailed discussion.

Segmenting an application in this way provides you several benefits, even outside the C++ construction from parts environment.

- It enhances parallel development.

Prototyping of the views can be done by user interface specialists working with end users. This activity can take place in parallel and is somewhat independent of the development of the underlying model.

- It supports connecting multiple views to the same model.

Users can access several concurrent views of the business model objects.

- It facilitates cooperative processing.

The business logic can be effectively distributed between the workstation and one or more servers.

Separation of Model Objects from View Objects

To segment your application into manageable chunks, first separate your view objects from your model objects. By segmenting them, you can provide several views of the same model object or objects. Figure 5 on page 12 shows two views (a detailed view and a tabular view) of a single model.

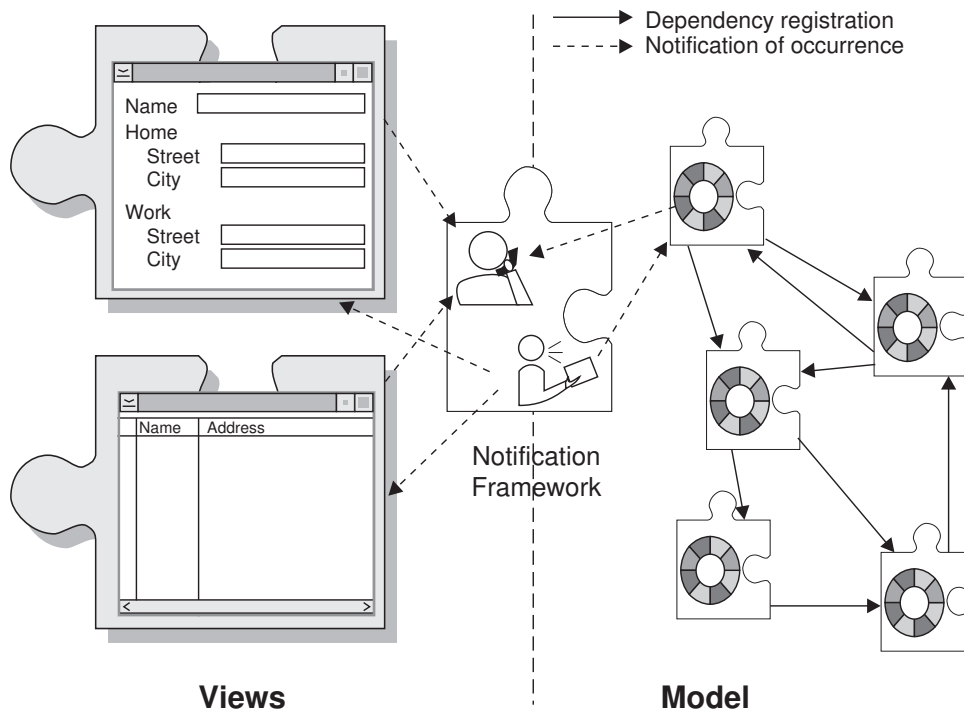


Figure 5. Connecting Views to a Model

To use this method of designing applications effectively, keep two important points in mind:

- Views can directly update models, but models cannot directly update views.
- Views contain only presentation and user-manipulation logic. Business logic exists only in the model objects.

The dependency manager (in C++, the notification framework) is used to communicate between views and models in place of sending direct messages. Systems that support the model-view or model-view-controller (MVC) application structure also support a dependency manager function. The specific implementation details may differ from system to system, but the overall concept remains constant.

The dependency manager maintains lists of objects that depend on the occurrence of specific events. It provides a set of interfaces so that objects can register their dependency on an event or remove their dependency from it.

An object can signal the occurrence of each of its events to the dependency manager. The dependency manager searches its lists and forwards the notification to the objects

that are dependent on the event. These objects can then take action based on the occurrence of the event.

To see how this works, consider a simple example. You might want to refer back to Figure 5 on page 12 as we go through the example.

In this example we define one event (`nameChanged`) that is associated with a change in the name of a person. Assume the upper-left object in the model is a person object. Because it maintains the data about a person, it has a dependency on the `nameChanged` event. Both view objects must also be notified when the `nameChanged` event occurs so they can update the content of the name field on the screen. Each of these objects registers a dependency on the `nameChanged` event with the dependency manager.

Now, suppose you type over the contents of the name field in the `detailView` object and then tab out of the field. The view object signals a notification that the `nameChanged` event has occurred. The dependency manager receives this notification message and looks for its list of `nameChanged` event dependents. It finds the list and forwards the notification to the objects that have registered their dependency on the event.

The person object receives the notification and updates its internal data. The `tabularView` object receives the notification and refreshes the display with the updated name.

Because the `detailView` object is also dependent on the `nameChanged` event, you might wonder why the notification was not forwarded back to it. Also, because the `tabularView` object signals a notification when the name changes, you might wonder why the program does not go into an endless loop sending this notification around. The answer to both these issues is that the dependency manager recognizes recursive notifications and discards them.

Segmentation within the Model

We have described one major approach to segmenting your application—dividing it into a model and one or more views of that model. You can further segment the model into several categories of objects. Figure 6 on page 14 shows these divisions using the analogy of an iceberg, because most models contain many more hidden (nonvisual) objects than visible real-world objects.

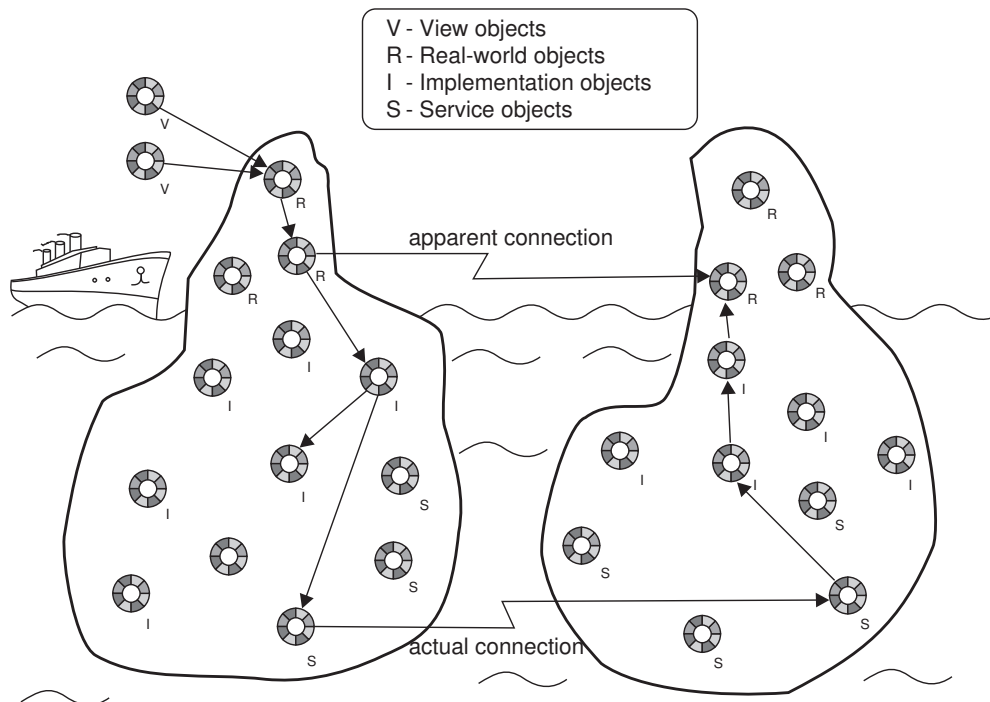


Figure 6. The Iceberg Model

The categories shown in the figure are as follows:

View objects (V). These objects create the user interface display. They implement the interface between users of the application and the application business model.

Real-world objects (R). These objects implement the physical objects of your business enterprise, such as a car, an invoice, a notepad, or a calendar. Their behavior closely models the behavior of these physical objects. They have a formally specified interface, which allows them to be widely used by application construction tools.

Implementation objects (I). These objects provide the internal implementation of the real-world objects. They generally correspond to more traditional computer-related entities, such as arrays, numbers, or abstract objects used to collect common behavior. These objects often do not have formally specified interfaces because they are usually designed to be used by programmers rather than by application builders.

Service objects (S). These objects provide access to external services, such as communication support, database access, or operating system functions. They insulate the real-world and implementation objects from the details of these external services. These objects are also generally designed for programmers, so they might not have formally specified interfaces.

As an example of this application segmentation, consider an application that shows you the contents of a customer file:

- The windows, entry fields, push buttons, and all other parts of the application's visual interface are view objects.
- The customer object is the real-world object, providing the data about a selected customer to the view objects.
- An array implementation object might be used internally by the customer object to hold the data.
- Service objects support querying a data store and returning the customer information, which insulates all the other objects from dependence upon the kind of data store used to hold the customer records or its location.



Chapter 3. The Architecture of C++ Construction from Parts

The C++ construction from parts architecture has been developed to meet the specific need of application developers to have an easier, more productive way to create high-quality applications in today's complex application development environment. Our ability to conceive this architecture was enabled by the evolution of application structure and by the emergence of application-building power tools.

This chapter describes the following:

- The origins of the C++ construction from parts architecture (page 17)
- Architecture characteristics (page 19)
- The part interface architecture (page 19)
- Kinds of parts supported in Visual Builder (page 23)

The Origins of the C++ Construction from Parts Architecture

In the past, applications were designed and implemented with a monolithic structure. In these applications, embedded logic usually governed the flow of control. Such applications controlled the user, rather than allowing the user to control the application. In addition, the monolithic structure led to application components that were highly customized for the application in which they were developed.

Figure 7 on page 18 shows how these applications evolved from the first generation, monolithic *batch* applications, to the second generation, online *transaction* applications. While this evolution did enable users to gain direct access to the applications through fixed-function terminals, it still left the applications in control of the users' interactions with the system.

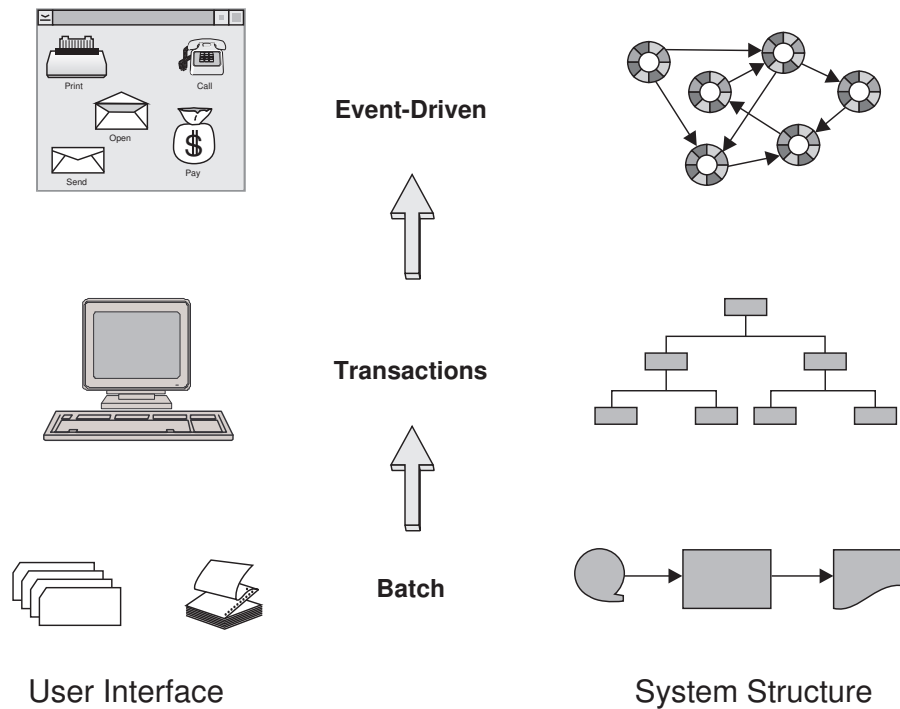


Figure 7. Evolution of Application Structure

As workstations became popular, another step in application structure evolution began. This evolutionary step came in response to the distributed nature of systems, the needs of the users to control system flow, and the desire of enterprises to reuse previously developed application components. While these three factors seem to be independent of each other, they drove toward a single solution. The application structure that developed in response to these factors represents the third generation in the evolutionary flow.

The third-generation application structure embodies the concept of *event-driven* application design and implementation. Event-driven applications allow the user to control the flow of operations. This structure also supports dividing applications into cooperating elements that can run on different systems. A natural outgrowth of this application structure is small, modularized components with increasingly standard interface protocols.

The C++ construction from parts architecture formalizes certain aspects of this application structure.

Architecture Characteristics

The C++ construction from parts architecture facilitates the development of parts to be used in this new environment. To be successful, this architecture must be both useful and easy to implement. It specifies the following:

- A structural paradigm for applications that is independent of implementation.

This maximizes the flexibility afforded to implementers. The only implementation constraints in the specification are those needed to provide reliable semantics for the interfaces.

- A standard interface protocol.

A small, simple protocol suite achieves greater acceptance by part builders and users than a large complex suite. This protocol supports communication among application parts and between application parts and the tools used to build the applications.

This standardized interface is called the *part interface*. Applications can be built from parts by connecting the part interface features.

The architecture specification supports both new and pre-existing object classes. You can apply the interface protocol to existing classes without making extensive code modifications.

In a C++ programming environment where applications are created using an editor or C++ browser, implementing the part interface of a part is sufficient. As more sophisticated tools, such as visual application builders, become available, parts can play a larger role in assisting the application developer to build applications.

Part Interface Architecture

As stated earlier, a part interface is composed of three clearly defined programming interface features: attributes, actions, and events. Figure 3 on page 6 shows a part interface. The part interface architecture specifies the general format of the programming interfaces, not the particular implementation behind the interface. For example, the protocol describes how to build an attribute interface, independent of the contents, address, name, or other properties that are specific to this part.

Access to Part Properties

Attributes provide access to the *properties* of a part. A property can be any of the following:

- An actual data object stored as a data member, such as the street in an address object

- An actual data object that is accessed via another object or the system, such as the contents of an entry field (the contents are stored within the system entry field control or widget)
- A computed data object that is a transformed version of an actual data object, such as the temperature in Fahrenheit when the actual data object is the temperature in Celsius
- A computed data object that is not stored, such as the sum of all numbers in an array or the profit that is computed by subtracting dealer cost from the retail price.

You can use the attribute interface to return the value of a property, to set the value of a property, and to notify other parts when the value of a property changes. You are not required to supply a complete attribute interface for a property. For example, a property might be read-only, in which case the part's attribute interface would not support the ability to set the property's value.

The attribute interface is represented as follows:

```
aType aQueryMember();
aSetMember(aType aValue);
static INotificationId const anEventId;
```

`aQueryMember` is the public member function to get the current value of the property; `aSetMember` is the public member function to set the value of the property to `aValue`; `aType` is the type of `aValue`; `anEventId` is the notification ID for the property change event.

The member function that sets the value of the property can use the following expression to notify dependent parts that the value of its property has changed:

```
notifyObservers(INotificationEvent(anEventId, *this, true, (void*)aValue));
```

`notifyObservers` is the member function that signals the event; `anEventId` is the notification ID for the property change event; `*this` is the notifier object; `true` indicates that the value of the attribute has changed; `aValue` is the event data. (For more information about events, see “Notification of Changes to Parts” on page 22.)

The following simpler call can be made if no event parameters are to be passed:

```
notifyObservers(INotificationEvent(anEventId, *this));
```

The member function that sets a property's value usually signals the value change, but any member function that is aware of the change can signal the event.

While a property is often represented as a data member of a part, it need not be; the property could be a computed value. What is important is that whenever the value of

the property changes, the change takes place using the set member function for the property. Changes made in any other way might not cause the event to be signalled.

A part implements the attribute interface protocol by implementing the member functions declared in the header files. For example, the following two member functions support the attribute interface protocol for the temperature property of the `IThermometer` part:

```
unsigned long IThermometer::temperature () const
{
    return iTemperature;
}

IThermometer& IThermometer::setTemperature (unsigned long newTemp)
{
    if (iTemperature != newTemp)
    {
        iTemperature = newTemp;
        notifyObservers(INotificationEvent(temperatureId, *this,
                                           true, (void*)newTemp));
    }
    return *this;
}
```

Because temperature has two common representations, Celsius and Fahrenheit, a more general solution would be to have an attribute for each representation. If the temperature was stored internally in Celsius, then you could rename the two member functions to `setTempInCelsius` and `tempInCelsius`. You could then implement two additional member functions, such as the following, that return and set the temperature in Fahrenheit:

```
unsigned long IThermometer::tempInFahrenheit () const
{
    return ((iTemperature * (9/5)) + 32);
}

IThermometer & IThermometer::setTempInFahrenheit (unsigned long newTemp)
{
    return setTempInCelsius ((newTemp - 32) * (5/9));
}
```

Notice that we did not introduce any additional data members when we added these two new member functions. There is still only one property (`iTemperature`) being maintained. However, now it is being maintained through two different attribute interfaces. This illustrates the design guideline for using a set member function (for example, `setTempInFahrenheit`) to change the value of a property. It also shows that a property is not always a data member.

Access to Part Behavior

An *action* provides access to the behavior of a part. Actions represent the tasks you can assign a part to do, such as open a window or add an object to a collection of objects.

The action interface is represented as follows:

```
aType aMemberFunction();
```

aMemberFunction is the public member function for the action to be performed.

A part implements the action interface by supplying a member function that responds to the behavior declared in the header file. For example, the following member function supports the action interface to set the default value of the city attribute in the IAddress class:

```
IAddress & IAddress :: setCityToDefault ()  
{  
    return setCity("Hometown");  
}
```

This example shows that actions can cause values of attributes to change. In fact, most Boolean attributes can be set to false using the disable member function. For example, the disableMouseClickedFocus member function in the IButton class causes the mouseClickedFocus attribute to be set to false.

Notification of Changes to Parts

By signalling events, a part can notify other parts that a state or value in its interface has changed. Events can be signalled when the state of a view part changes, such as when a push button is clicked or when a window is opened, as well as when the state of a model part changes, such as when the balance in a bank account becomes negative. Events can also be signalled when the value of a part's property changes, such as when money is deposited into or withdrawn from a bank account.

Notifications appear as messages broadcast to all parts that are *observers* of the event. Observers of an event are those parts that depend on the event's occurrence. The event interface is represented as follows:

```
static INotificationId const anEventId;
```

anEventId is the notification ID for the event.

Several different options are available to signal events. The first option is an example of using the event interface for attribute notification with event parameters:

```
notifyObservers(INotificationEvent(anEventId, *this, true, (void*)aValue));
```

- notifyObservers is the member function that causes the event notification.

- anEventId is the notification ID for the property change event.
- *this is the notifier object.
- true indicates that the value of the attribute has changed.
- aValue is the new value of the property.

The following simpler call can be made if no event parameters are to be passed:

```
notifyObservers(INotificationEvent(anEventId, *this));
```

Parts can also signal events when no attributes have changed, as follows:

```
notifyObservers(INotificationEvent(anEventId, *this, false, (void*)aValue));
```

- notifyObservers is the member function that signals the event.
- anEventId is the notification ID for the property change event.
- *this is the notifier object.
- false indicates that the value of the attribute has not changed.
- aValue is the value of the property.

The following simpler call can be made if no event parameters are to be passed:

```
notifyObservers(INotificationEvent(anEventId, *this, false));
```

To implement notification within a part, code the notifyObservers expression within one or more of its member functions. For example, the following setTempInCelsius member function notifies other parts when the temperature has changed and when the temperature is above the boiling point of water:

```
IThermometer& IThermometer::setTempInCelsius (unsigned long newTemp)
{
    if (iTemperature != newTemp)
    {
        iTemperature = newTemp;
        notifyObservers(INotificationEvent(temperatureId, *this,
                                           true, (void*)newTemp));
        if (iTemperature > 100)
        {
            notifyObservers(INotificationEvent(boilingId, *this));
        }
    }
    return *this;
}
```

Kinds of Parts Supported in Visual Builder

You can use many kinds of parts to construct applications. These different kinds of parts, categorized according to their characteristics, are shown in Figure 8 on page 24.

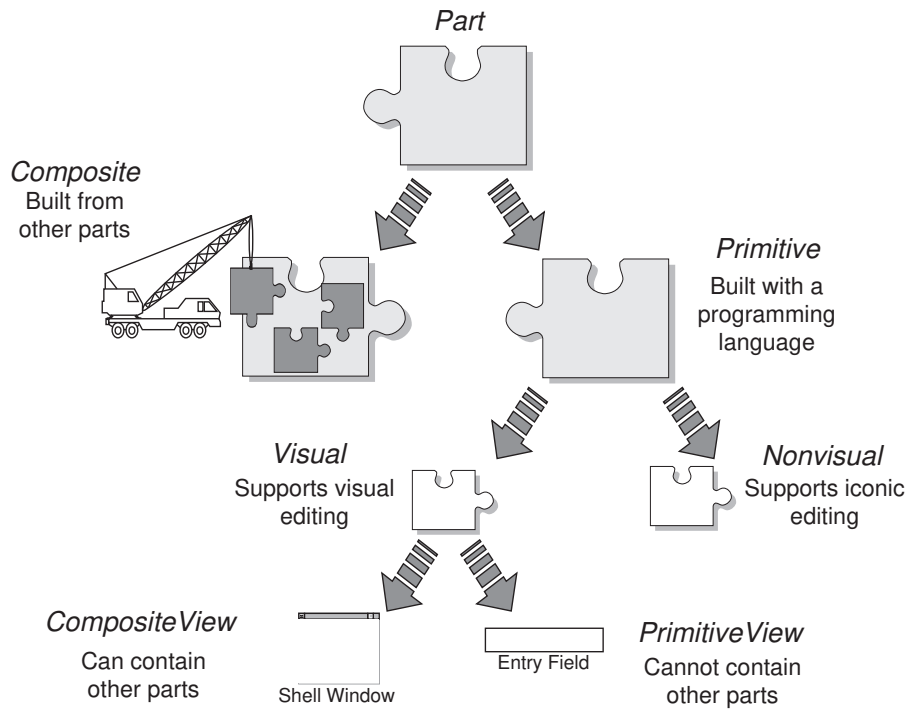


Figure 8. Kinds of Parts

As stated earlier, all parts are either *primitive parts*, which are the basic building blocks from which other parts are constructed, or *composite parts*, which are parts constructed from other parts. You must construct new primitive parts using a programming language because there are no similar parts to use in building them.

In turn, primitive parts can be either visual or nonvisual.

- *Visual parts* are elements of the application that the user can see at run time, such as view parts. They are components of a presentation surface, such as a window, an entry field, or a push button. The development-time representations of visual parts on the Composition Editor's free-form surface closely match their runtime visual forms. Users can edit these parts in the Composition Editor in their visual runtime forms (*visual editing*).
- *Nonvisual parts* are elements of the application that are not seen by the user at run time, such as model parts. On the Composition Editor's free-form surface, users can manipulate these parts only as icons (*iconic editing*). Examples of non-visual parts are business logic parts, data array parts, communication access protocol parts, and database query parts.

Parts that the user can see at run time but that support iconic editing only are treated as nonvisual parts at development time. A prompter message box part and a file selection dialog part are examples of this kind of nonvisual part.

Part 2. Building Basic Parts

The previous part describes the set of part interfaces that serve as the foundation for building primitive parts. This part expands on that foundation by describing how you can use those interfaces to build parts specifically for use with Visual Builder.

Whenever possible, use the Visual Builder development tools to build your parts. However, this may not always be the most practical way to create a new part. If you need a new primitive visual part (for example, a numeric entry field), or if you want to convert an existing C++ class into a part (for example, a dictionary), or if you need to develop a part with extensive model logic, you must use the underlying C++ development tools.

The three basic steps to construct a new part or enable an existing C++ class to be a part are the following:

1. Design your part.
2. Implement your part.
3. Describe the part interface.

Chapter 4. Designing a C++ Part	29
Design Guidelines	29
Conventions to Consider	30
 Chapter 5. Implementing a C++ Part	31
Positioning a Part within the Class Hierarchy	31
Implementing Events	32
Implementing Attributes	33
Implementing Actions	35
Implementing Constructors	36
Implementing Destructors	36
Implementing Assignment Operators	37
Creating Composers and Primitive Visual Parts	37
Implementation Checklists	38
 Chapter 6. Describing Part Interfaces in Part Information Files	41
Part Information Syntax	42
Class Information Syntax	52
Function Group Information Syntax	58
Enumeration Information Syntax	62
Type Definition Information Syntax	64
Other Syntax Examples	66

Chapter 7. Sharing Parts with Others	67
Providing part files (.vbb)	67
Providing Part Information Files (.vbe)	68



Chapter 4. Designing a C++ Part

This chapter highlights some factors to consider when designing a C++ part. Before creating a new primitive part, answer the following questions:

- Is the part visual or nonvisual? (See “Kinds of Parts Supported in Visual Builder” on page 23.)
- Can it be created as a composite part?
- Do you have a good model of the part and its responsibilities?
- Is it a real-world implementation or a service part? (See “Segmentation within the Model” on page 13.)

Design Guidelines

Designing a good part is very similar to designing a good class. In fact, a good part can be used as a traditional class in applications that are not otherwise being built using C++ construction from parts. Consider the following when designing your part:

- Keep it simple.
- Keep the number of actions, events, and attributes to a reasonable size. In practice, 10–20 part features per part is a good target.
- Minimize the dependencies on other parts and classes. Do not make nonvisual parts dependent upon visual parts.
- Specify several actions with a small number of parameters rather than a single action with many parameters. When possible, provide default parameter values.
- Minimize the number of connections that need to be made when using the Composition Editor.

To design a new part, do the following:

1. Determine the attributes (properties) of the part.
2. Determine the events (notifications) that the part will signal.
3. Determine the actions (behaviors) for the part.
4. After determining the part interface, investigate the available parts to see if one already exists or to determine which class to use as a base. Determine if any classes can be converted to parts.

Conventions to Consider

This section gives you some high-level conventions to follow when creating parts. For more information about conventions, see Appendix A, “IBM Open Class Library Conventions” on page 83.

Naming Parts

Because the names of C++ classes come from a flat name-space, developers of parts must ensure that their class names are unlikely to duplicate the class names used by other developers. Using a prefix on your class names is a good way to reduce the chances of duplicating a class name. All IBM Open Class names in the global name space begin with the letter “I” for IBM.

Naming Actions, Attributes, and Events

A *part feature* is an element of a part’s interface. It is used as a collective term for a part action, attribute, or event.

If you follow these simple conventions in choosing your feature names, it is easier for users of your parts to recognize the function of a feature:

- Name actions with phrases that indicate activities to be performed, together with an optional receiver of that activity. Examples of feature names for actions are `startTimer`, `openWindow`, `hide`, and `setFocus`.
- Name attributes with phrases that indicate the physical property they represent. Examples of feature names for attributes are `height`, `buttonLabel`, and `contents`.
- Name events with phrases that indicate activities that either have happened or are about to happen. Examples of feature names for events are `clicked`, `aboutToCloseWindow`, and `timeExpired`.

Note: Do not use feature names that start with *avl* or *vb*. These are reserved for use by Visual Builder.

The main place that users see your action, attribute, and event names is on the Connections pop-up menu of the Composition Editor. Because features are shown on this pop-up menu in alphabetical order, the phrasing you use for a feature name is the only way to distinguish between actions, attributes, and events.

It is important to choose unique names for your new actions, attributes, or events. This prevents you from unintentionally overriding an inherited part feature. If you intend to replace an existing part feature that your part inherits, then your new name must be the same as the name of the part feature you are replacing. The scope within which your feature name must be unique is your part class and all its base classes in the class hierarchy.



Chapter 5. Implementing a C++ Part

This chapter explains how to implement a C++ part. It explains how to create the header and code files for a C++ class that supports the parts architecture. No matter which kind of part you are building, you need to read the concepts and techniques introduced early in the chapter to understand later material. The key items covered in this chapter follow:

- Positioning the part within the class hierarchy
- Implementing events (notification) (page 32)
- Implementing attributes (properties) (page 33)
- Implementing actions (behaviors) (page 35)

Positioning a Part within the Class Hierarchy

The first step associated with implementing a part is positioning the part class in the C++ class hierarchy.

Place your nonvisual part, as shown in Table 1, under the `IStandardNotifier` class hierarchy. Inserted in this location, your part inherits certain default behavior from `IStandardNotifier` and the `INotifier` protocol.

Table 1. Class Hierarchy for Nonvisual Parts

Class	Responsibility
<code>IBase</code>	Base class
<code>IVBase</code>	Virtual base class
<code>INotifier</code>	Notification protocol
<code>IStandardNotifier</code>	Implementation of notification protocol
<i>New nonvisual part</i>	

An example from the `IAddress` header file follows:

```
class IAddress : public IStandardNotifier
```

In some cases, you might want your nonvisual part to be abstract. Be aware that your users cannot drop abstract nonvisual parts onto the Composition Editor's free-form surface. If you do not provide concrete parts derived from this abstract part, your users must derive their own concrete parts. For more information about the syntax to define a part as abstract, see "VBComposerInfo Statement for a Part" on page 44.

Once you have found your part's position in the class hierarchy, you are ready to begin the actual building.

Creating a C++ class for a part is not much different from creating any other C++ class. There are just a few additional guidelines to keep in mind for those member functions that support your part's part interface.

Implementing Events

Events might be occurring inside your part that you want to signal to other parts. The mechanism for signalling this information to other parts is event notification.

In a simple event notification, dependent parts are notified only that the event occurred. An example of this type of event notification is a push button visual part notifying all observer parts that it has been selected. The public section of the `IButton` class header file contains the following definition for the `buttonClickId` notification ID:

```
static INotificationId const
    buttonClickId;
```

In addition, the `IButton` code file contains the following:

```
const INotificationId IButton::buttonClickId="IButton::buttonClick";
```

The notification string ID contains the class name followed by the event name. `IButton` can notify others of this event by using `notifyObservers` with the `buttonClickId` notification event ID:

```
notifyObservers(INotificationEvent(buttonClickId, *this, false));
```

This function call signals that something happened (the button was clicked) but provides no additional information. The `false` parameter on the `INotificationEvent` constructor indicates that no attributes have changed.

A part can provide additional information about the event by passing parameters with the notification. An example of this use of an event notification is a push button part that represents a key on a calculator notifying all observer parts that it has been selected. In addition to notifying other parts that it has been selected, the push button part can also pass the value of its label, as follows:

```
notifyObservers(INotificationEvent(buttonClickId, *this,
    false, (void *)5));
```

This notifies other parts that push button 5 was selected.

Event notification is also used to inform other parts that attributes have changed. An example of this type of event notification is the street changing in the `IAddress` part. The part can do this with the following code:

```

IString eventData(iStreet);
notifyObservers(INotificationEvent(streetId, *this,
                                   true, (void*)&eventData));

```

Implementing Attributes

Each property that your part exposes through its attribute interface has one or two corresponding member functions to support the attribute interface protocol for accessing the property. The public member function that retrieves the value of a property is called the *get member function*. The public member function that sets the value of a property is called the *set member function*. In addition, you need to define a public static notification event ID for notification of changes to the attribute. An example of the definition of the street attribute from the public section of the header file for the `IAddress` class follows:

```

virtual IString
    street () const;
virtual IAddress
    &setStreet (const IString& aStreet);
static INotificationId const
    streetId;

```

Get and set member functions usually come in pairs. The exception is when a property is read-only (such as a property that represents the serial number of the computer you are currently using). In this case, the property has only a get member function.

Always use the get and set member functions to access the value of a property so that the associated behaviors are performed. In particular, if you update the value of a property without triggering event notification, the application might fail to operate correctly. Define property data members as private to ensure that other classes do not access the properties directly.

Defining Get Member Functions

Get member functions return the value of a part's property. They are always accessed using a public member function without parameters.

The simplest get member function returns the data member that holds the value of a property. The `street` member function of the `IAddress` class is an example of a simple get member function, as follows:

```

IString IAddress::street () const
{
    return iStreet;
}

```

Because this get member function does not change any values within the object, it is defined as `const`.

Defining Set Member Functions

Set member functions modify the value of a part's property and notify dependent objects that the value has changed. Set member functions are always accessed using a member function with one parameter—the value to be set into the property. Normally, this parameter is defined as `const` because most set member functions do not change the parameter.

A simple set member function sets the `IStreet` data member and signals a notification using `notifyObservers`. The following set member function, taken from the `IAddress` example part, does just that:

```
IAddress& IAddress::setStreet (const IString& aStreet)
{
    if (iStreet != aStreet)
    {
        iStreet = aStreet;
        IString eventData(iStreet);
        notifyObservers(INotificationEvent(streetId, *this,
                                           true, (void*)&eventData));
    } /* endif */
    return *this;
}
```

An even simpler set member function can be implemented that does not signal a notification when its property is changed. You might use such a set member function when you know that a group of properties is always changed together. In this case, only one set member function out of the group would actually signal the event. This couples the event signalling with the entire sequence of set member function calls.

Set member functions can also perform other operations, such as computing values for many properties based on the value supplied or signalling an additional notification when the value of a property crosses a threshold value.

Signal events only when the value of the property has changed. In addition, providing the new value of the attribute in the notification event can improve the overall system performance.

Attribute Notification IDs

You define notification IDs using public static data members. An example from the public section of the `IAddress` class header file follows:

```
static INotificationId const
    streetId,
    cityId,
    stateId,
    zipId;
```

This defines `streetId`, `cityId`, `stateId` and `zipId` as notification IDs. In addition, the `IAddress` code file contains the following:

```
const INotificationId IAddress::streetId="IAddress::street";
const INotificationId IAddress::cityId="IAddress::city";
const INotificationId IAddress::stateId="IAddress::state";
const INotificationId IAddress::zipId="IAddress::zip";
```

The notification string ID contains the class name followed by the attribute name.

Implementing Actions

Each behavior that your part exposes in its action interface has a corresponding public member function to support the action protocol for that behavior. An example of the `setCityToDefault` public member function from the `IAddress` class header file follows:

```
virtual IAddress
&setCityToDefault ();
```

Just like other C++ member functions, actions can have parameters and a return value. You can specify the return value and parameters as part of the action interface (see Chapter 6, “Describing Part Interfaces in Part Information Files” on page 41).

The only thing unique to Visual Builder about these member functions is that they should not directly access data members for properties; instead, use the attribute’s get and set member functions to access the data members. You need to do this because the get and set member functions often have additional behavior beyond simply accessing the value of the data member.

For example, the `setCityToDefault` member function of the `IAddress` class uses the following `setCity` to set the city to `Hometown` instead of setting it directly:

```
IAddress& IAddress::setCityToDefault ()
{
    return setCity("Hometown");
}
```

Consider providing an action to reset each attribute to a default value when implementing a part. For Boolean attributes, provide a public member function (for example, a disable action) that causes the attribute to be set to `false` and a public set member function (for example, an enable action) with a default parameter equal to `true`. You can use the set member function (for example, `enableMouseClickedFocus`) with a default value as the set member function for the Boolean attribute and an action member function. An example from the `IButton` class for the `mouseClickedFocus` Boolean attribute follows:

```

IButton
&enableMouseClickedFocus (Boolean turnOn = true),
&disableMouseClickedFocus ();

```

Implementing Constructors

Nonvisual parts should have a default constructor. An example of the IAddress class default constructor follows:

```
IAddress();
```

The implementation of the standard constructor for the IAddress class follows:

```

IAddress::IAddress() : IStandardNotifier (),
    iStreet("101 Main Street"),
    iCity("Hometown"),
    iState("NC"),
    iZip("27511")
{
}

```

For most nonvisual parts, supply a copy constructor. An example follows:

```
IAddress (const IAddress& partCopy);
```

The implementation of the copy constructor for IAddress follows:

```

IAddress::IAddress (const IAddress& partCopy)
: IStandardNotifier (partCopy),
  iStreet(partCopy.street()),
  iCity(partCopy.city()),
  iState(partCopy.state()),
  iZip(partCopy.zip())
{
}

```

Implementing Destructors

For all visual and nonvisual parts, specify a virtual destructor. An example follows:

```

virtual
~IAddress ();

```

The implementation of the IAddress destructor follows:

```

IAddress::~~IAddress()
{
}

```

Implementing Assignment Operators

To ensure that attribute changes are signalled, specify an assignment operator for non-visual parts with attributes, as follows:

```
IAddress& operator= (const IAddress& aIAddress);
```

The implementation of the IAddress class assignment operator follows:

```
IAddress& IAddress::operator= (const IAddress& aIAddress)
{
    if (this == &aIAddress) {
        return *this;
    } /* endif */
    IStandardNotifier::operator=(aIAddress);
    setStreet(aIAddress.street());
    setCity(aIAddress.city());
    setState(aIAddress.state());
    setZip(aIAddress.zip());
    return *this;
}
```

Note the following in the previous example:

- The assignment operator checks to ensure that the new value is not the current object.
- The part's base class is called to ensure that the base class' data is assigned correctly.
- To ensure notification of attribute changes, the attribute set member functions are used to change the value of the attributes.

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 about importing parts, see the *Visual Builder User's Guide*.

Implementation Checklists

The following checklists contain the items required to implement a new part or to convert an existing class to a part. Because parts are implemented as classes, you can convert existing classes to parts and still use them as classes.

In the header file, make the following changes to support parts:

1. For nonvisual parts, publicly inherit from IStandardNotifier or a derived class.
For visual parts, publicly inherit from IWindow or a derived class.
2. Define the constructors and a virtual destructor.
3. If appropriate, define the assignment operator for nonvisual parts.
4. Define a public notification ID for each event.
5. Define a public notification ID for each attribute.
6. Define a public get member function with no parameters so users can obtain the value of each attribute.
7. If the attribute can be changed, define a public set member function with a single parameter containing the new value. If this attribute is Boolean, set the default to true.
8. Define any public action member functions. Consider reset or default actions for attributes, including disable and enable actions for Boolean attributes.

In the code or inline file, make the following changes in the code logic to support parts:

1. Code each event notification ID using a string containing the class name and event name.
2. Code each attribute notification ID using a string containing the class name and attribute name.
3. Code the constructors and a virtual destructor.
4. If defined, code the assignment operator.
5. Code the public get member functions for each attribute.
6. Code the public set member functions for each attribute. Notify observers when the value changes.
7. Call get and set member functions to return and change attribute values in actions. In addition, notify observers of events specified by the part.



Chapter 6. Describing Part Interfaces in Part Information Files

You describe the interface information that Visual Builder needs by using the Part Interface Editor or by creating files containing the interface information and importing these files into Visual Builder. This chapter describes the format of the interface information and is divided into the following sections:

- Part information syntax (page 42)
- Class information syntax (page 52)
- Function group information syntax (page 58)
- Enumeration information syntax (page 62)
- Type definition information syntax (page 64)

You can include the statements describing the interface in a C++ header file or in a separate file (sometimes called a *part information file*, or .vbe file). All interface information code lines begin with //VB in column 1. Between statements, lines that do not start with //VB are ignored. You can arrange statements on a single line or continue them on multiple lines by using the VB statement.

Rules for entering part information

The following rules apply to all interface information:

1. All part, class, function group, enumeration, and type definition names must be unique.
2. A single file can contain information about multiple parts, classes, function groups, enumerations, and type definitions.
3. You cannot begin one type of information until you have ended another. For example, you cannot put a VBBeginEnumInfo statement between VBBeginPartInfo and VBEndPartInfo statements. If you do, you will not be able to import the part information.
4. The interface information about a specific part, class, function group, enumeration, and type definition must be contained in a single file.
5. In some cases, you must account for missing keywords by including a comma (,) in the statement with no value. For example, you can use the VBComposerInfo statement to specify different types of support information, only one of which is required. The following statements are acceptable:

```
//VBComposerInfo: visual  
  
//VBComposerInfo: visual,803,cppov33r,userprimitive  
  
//VBComposerInfo: visual,,,userprimitive  
The following statement is not acceptable:  
//VBComposerInfo: visual,userprimitive
```

VB statements for IBM use only

You might find the following VB statements used in part information files shipped by IBM as samples. Do not use these statements in your own part information files; you could seriously compromise your part data.

- VBComposerClass
- VBComposerTextSetting
- VBFlagAttribute
- VBFlagInfo
- VBGeneratorClass
- VBSettingsPages

Part Information Syntax

This section describes the interface information for nonvisual parts that Visual Builder uses. Syntax descriptions appear in the recommended order of occurrence in a file. The following rules apply to the interface information for parts:

1. All feature names (attributes, events, actions) within a part hierarchy must be unique. If duplicate feature names exist, the information for the derived part is used and the information for the base part is ignored for that specific feature.
2. All part names must be unique and must be valid C++ class names.
3. The name on the VBBeginPartInfo statement and VBEndPartInfo statement must match.
4. Attribute, event and action information statements can appear more than once for a specific part, but all other information statements must appear only once.

For information about how to read these syntax diagrams, see “How to Read Syntax Diagrams” on page xii.

VBBeginPartInfo Statement for a Part

The first statement describing the interface is the VBBeginPartInfo statement. This statement specifies the part name and the part description.

►►—//VBBeginPartInfo:—*part_name*—┐
└┐, "*description*"└┐→

part_name

Valid and unique C++ class name that implements the part interface.

description

Optional part description.

VBParent Statement for a Part

The VBParent statement describes the part's base part. The attributes, actions, and events defined by the base part are inherited. The part class hierarchy must include the IStandardNotifier class for nonvisual parts.

►—┐
└┐//VBParent:—*parent_name*└┐→

parent_name

Valid and unique C++ class name.

VBIncludes Statement for a Part

The VBIncludes statement describes the include file that contains the declaration of the C++ class that implements the part interface.

►—//VBIncludes:—<*include_file*>—┐
└┐*include_define*└┐→

include_file

Header file required by the compiler to use this part. If you want to specify a link dependency in your application make file, use double quotation marks (" ") to delimit the file name instead of the angle brackets (< >).

include_define

Optional statement that associates a variable with the include statement, enabling generated code to test for duplicate include statements.

VBPartDataFile Statement for a Part

The optional VBPartDataFile statement specifies the file in which to save the part information. If you do not use this statement, the imported file name is used with a .vbb extension.

→ `//VBPartDataFile:—file_name—` →

file_name

File name in which to save the part interface information.

VBLibFile Statement for a Part

The optional VBLibFile statement specifies the library file that will contain the part when it is compiled. Visual Builder uses this data as follows:

- It inserts a #pragma statement in the part's generated header code.
- It adds the file name to the dependency list in the make file generated for applications using the part.

→ `//VBLibFile:—file_name—` →

file_name

Name of the file that will contain the compiled nonvisual part.

VBComposerInfo Statement for a Part

The part VBComposerInfo statement specifies the information needed to support visual and nonvisual parts. This information can include the following options:

- Icon resource ID and icon resource DLL name.

You must include either both of these options or neither of them. In addition, if you specify any other options on this statement, you must include the commas that would be used to delimit them. If you do not specify any options, omit the commas.

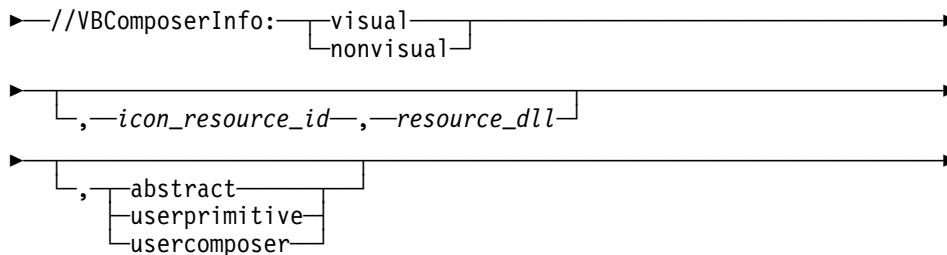
- Additional information about how the part can be used.

An *abstract* part is like an abstract class. It contains functions and data that other parts inherit. You cannot place an abstract part on the free-form surface.

A *user primitive* part does not accept child parts.

A *user composer* part can accept child windows.

Both user primitive and user composer parts are parts that you create, not parts that Visual Builder provides. For more information on implementing these types of parts, refer to the *Visual Builder User's Guide*.



icon_resource_id

Optional resource number of the icon to be used to represent the part. If you specify this, you must also specify the resource DLL name.

resource_dll

Resource DLL name containing the icon to be used. Do not include the .dll extension.

abstract

Optional keyword to indicate that the part is abstract. This keyword is mutually exclusive of the `userprimitive` and `usercomposer` keywords.

userprimitive

Optional keyword to indicate that the part is a primitive part that you created. This keyword is mutually exclusive of the `abstract` and `usercomposer` keywords.

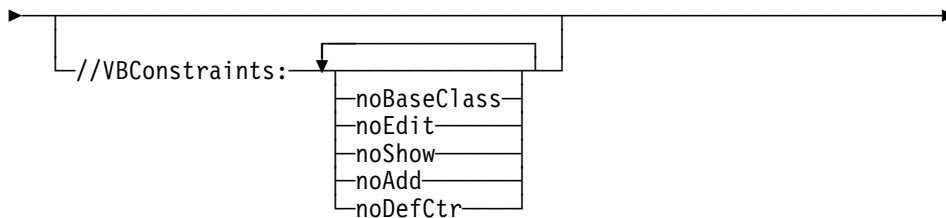
usercomposer

Optional keyword to indicate that the part is a composer part that you created. This keyword is mutually exclusive of the `abstract` and `userprimitive` keywords.

If you are implementing a user primitive or user composer part that does not take a style flag as input to its constructor, you must use `VBAAttribute` and `VBConstructor` statements to prevent problems with your part at run time. For more information, see “VBConstructor Statement for a Part” on page 47.

VBConstraints Statement for a Part

The optional VBConstraints statement enables you to limit the use of your part.



noBaseClass

Optional keyword to prevent the part from being used as a base class by another part or class.

noEdit

Optional keyword to prevent users from opening the part in a Visual Builder editor.

noShow

Optional keyword to prevent the part from being listed on the Visual Builder window.

noAdd

Optional keyword to prevent the part from being added to the Composition Editor's free-form surface.

noDefCtr

Optional keyword indicating that the part requires at least one input parameter to be constructed. If this keyword is specified, supply constructor information in a VBConstructor statement.

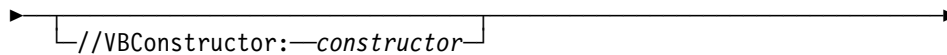
Using this keyword enforces the following restrictions:

- An attribute of this type cannot be torn off on the free-form surface.
- Users cannot create a variable of this type. To use this part as a variable, users must specify a pointer to this part as the variable type.
- This type cannot be returned as an action result. Instead, users must specify a pointer to this part as the return type of the action.

VBConstructor Statement for a Part

The optional VBConstructor statement specifies the form of the part's constructor. If you do not use this statement, Visual Builder expects a constructor as follows:

- If the part is nonvisual, a standard C++ default constructor.
- If the part is visual, a constructor of the same form as a public constructor for IWindow. IBM Open Class provides several constructors that differ in the default values provided for the input arguments.



```
//VBConstructor:—constructor
```

constructor

C++ public constructor definition.

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
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. For more information about .cpv and .hvp files, see the *Visual Builder User's Guide*.

Visual Builder expects the constructor for visual parts to include a style flag. If your part's constructor does not take a style flag as an input parameter, you must include VBConstructor and VBAttribute statements for the part. The following VBAttribute statement prevents the Styles setting page from being enabled for the part. Use this statement to prevent problems at run time.

```
//VBAttribute: style,
//VB:          "override to remove style from the part"
//VB:          integer,,, NOCONNECT NOSETTING
```

VBEvent Statement for a Part

The optional VBEvent statement specifies the event information. This information includes the event name, event description, event notification ID, parameter name, and parameter type. Use this statement to describe events implemented by the part that are useful in making connections. Do not use the VBEvent statement to describe events that occur as a result of attribute changes; specify this information on the VBAttribute statement.

```
►--//VBEvent:—event_name—, —————, —notificationID—►
                        └"description"┘
└—————,parameter_name—————┘
      └————,parameter_type——┘
```

event_name

Event name to be used by the Visual Builder user interface.

description

Optional event description.

notificationID

C++ public notification ID.

parameter_name

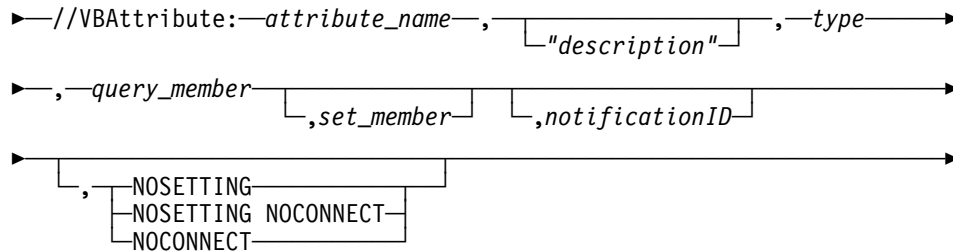
Optional event parameter name.

parameter_type

Optional event parameter type.

VBAttribute Statement for a Part

The optional VBAttribute statement specifies the attribute information. This information includes the attribute name, attribute description, attribute type (class name), get member function declaration, set member function declaration, and attribute notification ID. Use this statement to describe attributes implemented by the part that are useful in making connections.



attribute_name

Attribute name to be used by the Visual Builder user interface.

description

Optional attribute description.

type

Attribute type.

query_member

C++ public get member function that returns the attribute value. Specify this as a forward declaration.

set_member

Optional C++ public set member function that changes the attribute value. Specify this as a forward declaration.

notificationID

Optional C++ public attribute notification ID.

NOSETTING

Optional keyword to prevent the attribute's appearance on a generic settings page.

NOSETTING NOCONNECT

Optional keyword combination to hide a feature defined by a base part.

NOCONNECT

Optional keyword to disable connections to this feature.

VBAAction Statement for a Part

The optional VBAAction statement specifies the action information. This information includes the action name, action description, return type, and the C++ action member that implements the action. Use this statement to describe actions implemented by the part that are useful in making connections. To describe member functions that support attributes, use the VBAAttribute statement.

```

//VBAAction:—action_name—, —"description"—, —return_type—, —action_member—, —NOCONNECT—

```

action_name

Action name to be used by the Visual Builder user interface. Connections to this action are allowed unless the NOCONNECT option has been specified.

description

Optional action description.

return_type

Optional return type that specifies the type of the return value from the action and allows the user to make connections to this value.

action_member

C++ public member function that implements the action. The parameter names are used in the Visual Builder user interface to make connections to parameters.

NOCONNECT

Optional keyword to disable connections to this feature.

VBPreferredFeatures Statement for a Part

The optional VBPreferredFeatures statement specifies the preferred part features. If you do not use this statement, the base part's preferred list is used. If you do use this statement, the base part's preferred list is not inherited, and you must specify the complete list of preferred features for this part.

```

//VBPreferredFeatures: —, —action_name—  

—attribute_name—  

—event_name—

```

action_name

Preferred action name.

attribute_name

Preferred attribute name.

event_name

Preferred event name.

VB Statement for a Part

The optional VB statement allows interface information to be continued on the next statement line. This can be useful when a single statement exceeds a reasonable length. For example, you can use this statement to keep each line under 80 characters.

► `//VB:—` ►

VBEndPartInfo Statement for a Part

The VBEndPartInfo statement specifies the end of the part interface information.

► `//VBEndPartInfo:—part_name` ►

part_name

Valid and unique C++ class name.

Sample Part Information for IAddress

The following is an example of interface information for the IAddress nonvisual part:

```
//VBBeginPartInfo: IAddress, "IBM sample address"
//VBPParent: IStandardNotifier
//VBIncludes: <iadd.hpp> _IADD_
//VBPartDataFile: 'VBSample.vbb'
//VBComposerInfo: nonvisual, 10058, dde4vr30
//
//VBAttribute: city,
//VB:         "Query the city (IString) attribute.",
//VB:         IString,
//VB:         virtual IString city() const,
//VB:         virtual IAddress& setCity(const IString& city),
//VB:         cityId
//VBAttribute: state,
//VB:         "Query the state (IString) attribute.",
//VB:         IString,
//VB:         virtual IString state() const,
//VB:         virtual IAddress& setState(const IString& state),
//VB:         stateId
//VBAttribute: street,
//VB:         "Query the street (IString) attribute.",
//VB:         IString,
//VB:         virtual IString street() const,
//VB:         virtual IAddress& setStreet(const IString& street),
//VB:         streetId
//VBAttribute: zip,
//VB:         "Query the zip (IString) attribute.",
//VB:         IString,
//VB:         virtual IString zip() const,
//VB:         virtual IAddress& setZip(const IString& zip),
```

```

//VB:      zipId
//VBAction: operator !=
//VB:      ,, Boolean,
//VB:      Boolean operator !=(const IAddress* aValue) const
//VBAction: operator ==
//VB:      ,, Boolean,
//VB:      Boolean operator ==(const IAddress* aValue) const
//VBAction: setCityToDefault
//VB:      ,"Perform the setCityToDefault action.",,
//VB:      virtual IAddress& setCityToDefault()
//VBAction: setStateToDefault
//VB:      ,"Perform the setStateToDefault action.",,
//VB:      virtual IAddress& setStateToDefault()
//VBAction: setStreetToDefault
//VB:      ,"Perform the setStreetToDefault action.",,
//VB:      virtual IAddress& setStreetToDefault()
//VBAction: setToDefault
//VB:      ,"Perform the setToDefault action.",,
//VB:      virtual IAddress& setToDefault()
//VBAction: setZipToDefault
//VB:      ,"Perform the setZipToDefault action.",,
//VB:      virtual IAddress& setZipToDefault()
//VBPreferredFeatures: this, city, state, street, zip
//VBEndPartInfo: IAddress

```

Class Information Syntax

This section describes the interface information for classes that Visual Builder uses. Syntax descriptions appear in the recommended order of occurrence in a part information file. Many of the statements are similar to the part interface statements documented in the previous section. The following rules apply to the interface information for classes:

1. All feature names (attributes, actions) within a class hierarchy must be unique. If duplicate feature names exist, the information for the derived class is used and the information for the base class is ignored for that specific feature.
2. All class names must be unique and be valid C++ class names.
3. The names on the VBBeginPartInfo statement and VBEndPartInfo statement must match.
4. Attribute and action information statements can appear more than once for a specific part, but all other information statements must appear only once.

For information about how to read these syntax diagrams, see “How to Read Syntax Diagrams” on page xii.

VBBeginPartInfo Statement for a Class

The first statement describing the interface is the VBBeginPartInfo statement. This statement specifies the class name and the class description.

```
►►//VBBeginPartInfo:—class_name—┐—————→  
                                └┐"description"└┐
```

class_name

Valid and unique C++ class name that implements the interface.

description

Optional class description.

VBParent Statement for a Class

The optional VBParent statement describes the class' base class. The actions defined by the base class are inherited.

```
►┐—————→  
└┐//VBParent:—parent_name└┐
```

parent_name

Valid and unique C++ class name.

VBIncludes Statement for a Class

The VBIncludes statement describes the include file that contains the declaration of the C++ class that implements the interface.

```
►—//VBIncludes:—<include_file>—┐—————→  
                                └┐include_define└┐
```

include_file

Header file required by the compiler to use this class. If you want to specify a link dependency in your application make file, use double quotation marks (" ") to delimit the file name instead of the angle brackets (< >).

include_define

Optional statement that associates a variable with the include statement, enabling generated code to test for duplicate include statements.

VBPartDataFile Statement for a Class

The optional VBPartDataFile statement specifies the file in which to save the class interface information. If you do not use this statement, the imported file name is used with a .vbb extension.

→ `//VBPartDataFile:—file_name` →

file_name

File name in which to save the class interface information.

VBComposerInfo Statement for a Class

The class VBComposerInfo statement specifies the composer information needed to support classes. This information includes the icon resource ID and icon resource DLL name.

→ `//VBComposerInfo:—class` →
→ `—, —icon_resource_id—, —resource_dll` `—, —abstract` →

icon_resource_id

Resource number of the icon to be used to represent the class. If you specify this, you must also specify the resource DLL name.

resource_dll

Resource DLL name containing the icon to be used. Do not include the .dll extension.

abstract

Optional keyword to indicate that the class is abstract. Abstract classes cannot be dropped on the free-form surface.

VBConstraints Statement for a Class

The optional VBConstraints statement enables you to limit the use of your class.

→ `//VBConstraints:` →
→ `noBaseClass`
→ `noEdit`
→ `noShow`
→ `noAdd`
→ `noDefCtr` →

noBaseClass

Optional keyword to prevent the class from being used as a base class by another part or class.

noEdit

Optional keyword to prevent users from opening the class in a Visual Builder editor.

noShow

Optional keyword to prevent the class from being listed on the Visual Builder window.

noAdd

Optional keyword to prevent the class from being added to the Composition Editor's free-form surface.

noDefCtr

Optional keyword to indicate that the class requires at least one input parameter to be constructed. If this keyword is specified, supply constructor information in a `VBConstructor` statement.

Using this keyword enforces the following restrictions:

- An attribute of this type cannot be torn off on the free-form surface.
- Users cannot create a variable of this type. To use this class as a variable, users must specify a pointer to this class as the variable type.
- This type cannot be returned as an action result. Instead, users must specify a pointer to this class as the return type of the action.

VBConstructor Statement for a Class

The optional VBConstructor statement specifies the class constructor information.

```

//VBConstructor:—constructor

```

constructor

C++ public constructor definition. The parameter names are used in the Visual Builder settings pages to specify the parameters. Each parameter name must match an attribute name.

VBAction Statement for a Class

The optional VBAction statement specifies the action information. This information includes the action name, action description, return type, and the action member that implements the action.

```

▶//VBAction:—action_name—, [ "description" ],
▶ [ return_type ], —action_member [ , NOCONNECT ]

```

action_name

Action name to be used by the Visual Builder user interface. Connections to this action are allowed unless the NOCONNECT option has been specified.

description

Optional action description.

return_type

Optional return type.

action_member

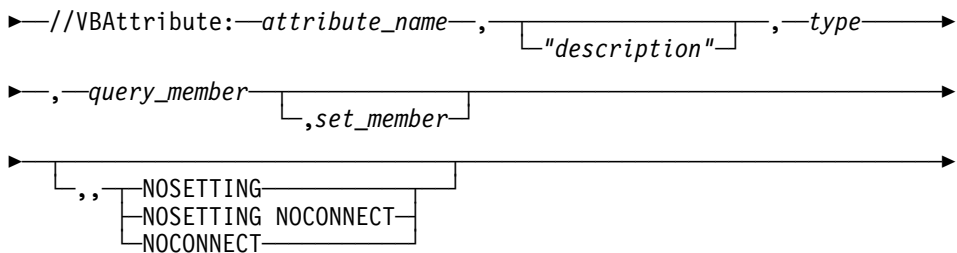
C++ public member function that implements the action. The parameter names are used in the Visual Builder user interface to make connections to parameters.

NOCONNECT

Optional keyword to disable connections to this feature.

VBAAttribute Statement for a Class

The optional VBAAttribute statement specifies the attribute information. This information includes the attribute name, attribute description, attribute type (class name), get member function declaration, set member function declaration, and attribute notification ID.

**attribute_name**

Attribute name to be used by the Visual Builder user interface.

description

Optional attribute description.

type

Attribute type.

query_member

C++ public get member function that returns the attribute value. Specify this in the form of a forward declaration.

set_member

Optional C++ public set member function that changes the attribute value. Specify this in the form of a forward declaration.

NOSETTING

Optional keyword to prevent the attribute's appearance on a generic settings page.

NOSETTING NOCONNECT

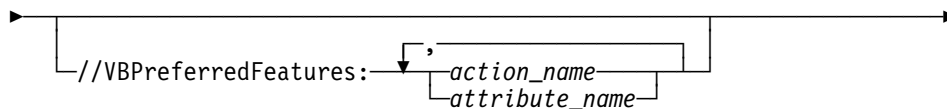
Optional keyword combination to hide a feature defined by a base class.

NOCONNECT

Optional keyword to disable connections to this feature.

VBPreferredFeatures Statement for a Class

The optional VBPreferredFeatures statement specifies the preferred part features. If you do not use this statement, the base class' preferred list is used. If you do use this statement, the base class' preferred list is not inherited, and you must specify the complete list of preferred features for this class.



action_name

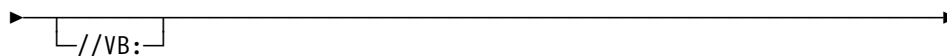
Preferred action name.

attribute_name

Preferred attribute name.

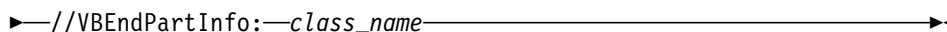
VB Statement for a Class

The optional VB statement allows the information to be continued on the next statement line. This can be useful when a single statement exceeds a reasonable length. For example, you can use this statement to keep each line under 80 characters.



VBEndPartInfo Statement for a Class

The VBEndPartInfo statement specifies the end of the class interface information.



class_name

Valid and unique C++ class name.

Sample Class Information for IRange

The following is an example of interface information for the IRange class:

```

//VBBeginPartInfo: IRange, "IBM range of coordinate values"
//VBParent: IPair
//VBIncludes: <ipoint.hpp> _IPOINT_
//VBPartDataFile: 'VBBase.vbb'
//VBComposerInfo: class, 204, dde4vr30
//
//VBAttribute: lowerBound,
//VB:         "Returns the lower bound of the range.",
//VB:         Coord,
//VB:         Coord lowerBound() const,
//VB:         IRange& setLowerBound(Coord lowerBound)
//VBAttribute: upperBound,
//VB:         "Returns the upper bound of the range.",
//VB:         Coord,
//VB:         Coord upperBound() const,
//VB:         IRange& setUpperBound(Coord upperBound)
//VBAction: includes,
//VB:         "Returns true if the range contains the specified coordinate value.",
//VB:         Boolean,
//VB:         Boolean includes(Coord aValue) const
//VBPreferredFeatures: this, lowerBound, upperBound
//VPEndPartInfo: IRange

```

Function Group Information Syntax

This section describes the interface information for a group of related C functions that you want to use within Visual Builder. Syntax descriptions appear in the recommended order of occurrence in a file. The following rules apply to the interface information for function groups:

1. All action names within a function group must be unique.
2. All function group names must be unique.
3. The name on the VBBeginPartInfo statement and VPEndPartInfo statement must match.
4. Action information statements can appear more than once for a specific part, but all other information statements must appear only once.

For information about how to read these syntax diagrams, see “How to Read Syntax Diagrams” on page xii.

VBBeginPartInfo Statement for Function Groups

The first statement describing a group of functions is the VBBeginPartInfo statement. This statement specifies the function group name and the function group description.

►►—//VBBeginPartInfo:—*function_group_name*—┐
└┐, "*description*"└┐→

function_group_name

A unique name that identifies a set of related functions as a group.

description

Optional description.

VBIncludes Statement for Function Groups

The VBIncludes statement describes the include file that contains the definition of the function group implemented in the interface.

►►—//VBIncludes:—<*include_file*>—┐
└┐*include_define*└┐→

include_file

Header file required by the compiler to use this function group. If you want to specify a link dependency in your application make file, use double quotation marks (" ") to delimit the file name instead of the angle brackets (< >).

include_define

Optional statement that associates a variable with the include statement, enabling generated code to test for duplicate include statements.

VBPartDataFile Statement for Function Groups

The optional VBPartDataFile statement specifies the file in which to save the function information. If you do not use this statement, the imported file name is used with a .vbb extension.

►—┐
└┐//VBPartDataFile:—*file_name*└┐→

file_name

File name in which to save the function group interface information.

VBComposerInfo Statement for Function Groups

The VBComposerInfo statement specifies the composer information needed to support a group of functions. This information includes the icon resource ID and icon resource DLL name.

```
► //VBComposerInfo:—functions—►  
└─, —icon_resource_id—, —resource_dll—┘
```

icon_resource_id

Resource number of the icon to be used to represent the function group. If you specify this, you must also specify the resource DLL name.

resource_dll

Resource DLL name containing the icon to be used. Do not include the .dll extension.

VBConstraints Statement for Function Groups

The optional VBConstraints statement enables you to limit the use of your function group.

```
► └─ //VBConstraints: ─┘  
└─ ┌─ noShow ─┘  
   └─ noAdd ─┘
```

noShow

Optional keyword to prevent the function group from being listed on the Visual Builder window.

noAdd

Optional keyword to prevent the function group from being added to the Composition Editor's free-form surface.

VBAction Statement for Function Groups

The optional VBAction statement specifies the action information. This information includes the action name, action description, return type, and the C++ function definition.

```
► //VBAction:—action_name—, └─ "description" ─┘, ─►  
└─ └─ return_type ─┘, —function_definition—►
```


action_name

Action name to be used by the Visual Builder user interface.

description

Optional action description.

return_type

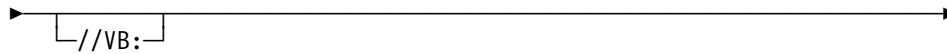
Optional return type.

function_definition

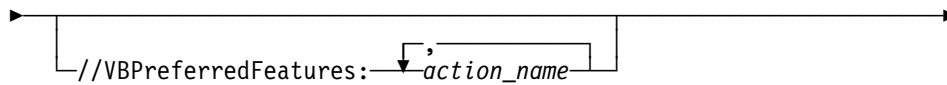
C public function that implements the action. The parameter names are used in the Visual Builder user interface to make connections to parameters.

VB Statement for Function Groups

The optional VB statement allows the information to be continued on the next statement line. This can be useful when a single statement exceeds a reasonable length. For example, you can use this statement to keep each line under 80 characters.

**VBPreferredFeatures Statement for Function Groups**

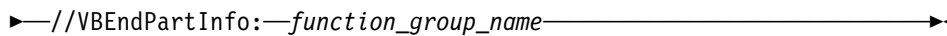
The optional VBPreferredFeatures statement specifies the preferred actions.

**action_name**

Preferred action name.

VBEndPartInfo Statement for Function Groups

The VBEndPartInfo statement specifies the end of the interface information.

**function_group_name**

Function group name matching the previous VBBeginPartInfo statement.

Sample Function Group Information

The following is an example of a function group interface:

```
//
// process function group information
//
//VBBeginPartInfo: ProcessFunctions, "Process functions"
//VBIncludes: <process.h>
//VBPartDataFile: 'Process.vbb'
//VBComposerInfo: functions, 10701, dde4vr30
//VBAction: abort, "Abort program.",,
//VB:      void _LNK_CONV abort( )
//VBAction: exit, "Exit program.",,
//VB:      void _LNK_CONV exit( int exitCode)
//VBAction: system, "Execute system command.",,
//VB:      void _LNK_CONV system( const char * commandString)
//VBEndPartInfo: ProcessFunctions
```

Enumeration Information Syntax

This section describes the interface information for enumerations that Visual Builder uses. Syntax descriptions appear in the recommended order of occurrence in a file.

For information about how to read these syntax diagrams, see “How to Read Syntax Diagrams” on page xii.

VBBeginEnumInfo Statement

The first statement describing the interface is the VBBeginEnumInfo statement. This statement specifies the enumeration name and description. The *enum_name* must be a valid and unique C++ enumeration name.

►►—//VBBeginEnumInfo:—*enum_name*—┐, "*description*"┘—————►

enum_name

Fully qualified C++ enumeration name.

description

Optional enumeration description.

VBIncludes Statement for an Enumeration

The VBIncludes statement describes the include file that contains the definition of the enumeration.

►—//VBIncludes:—<*include_file*>—┐*include_define*┘—————►

include_file

Header file that defines the enumeration. If you want to specify a link dependency in your application make file, use double quotation marks (" ") to delimit the file name instead of the angle brackets (< >).

include_define

Optional statement that associates a variable with the include statement, enabling generated code to test for duplicate include statements.

VBPartDataFile Statement for an Enumeration

The optional VBPartDataFile statement specifies the file in which to save the enumeration information. If you do not use this statement, the imported file name is used with a .vbb extension.

► `//VBPartDataFile:—file_name—` →

file_name

File name in which to save the enumeration interface information.

VBEnumerators Statement

The VBEnumerators statement describes the enumerators for the enumeration.

► `//VBEnumerators:—enumerator—=value—` →

enumerator

Enumerator name.

value

Value assigned to this enumerator.

VB Statement for an Enumeration

The optional VB statement allows the information to be continued on the next statement line. This can be useful when a single statement exceeds a reasonable length. For example, you can use this statement to keep each line under 80 characters.

► `//VB:—` →

VBEndEnumInfo Statement

The VBEndEnumInfo statement specifies the end of the interface information.

►—//VBEndEnumInfo:—*enum_name*—————►◄

enum_name

Enumeration name matching the previous VBBeginEnumInfo statement.

Sample Alignment Enumeration for IEntryField

The IEntryField class defined in the ientryfd.hpp header file contains the following nested public Alignment enumeration:

```
#ifndef _IENTRYFD_
#define _IENTRYFD_
class IEntryField : public ITextControl {
public:
    enum Alignment {
        left,
        center,
        right
    };
};
#endif
```

An example of how you could define this Alignment enumeration in an interface file follows:

```
//VBBeginEnumInfo: IEntryField::Alignment
//VBIncludes: <ientryfd.hpp> _IENTRYFD_
//VBPartDataFile: 'VBBase.vbb'
//VBEnumerators: left
//VB:             ,center
//VB:             ,right
//VBEndEnumInfo: IEntryField::Alignment
```

Type Definition Information Syntax

This section describes the interface information for type definitions that Visual Builder uses. Syntax descriptions appear in the recommended order of occurrence in a file.

For information about how to read these syntax diagrams, see “How to Read Syntax Diagrams” on page xii.

VBBeginTypedefInfo Statement

The first statement describing the interface is the VBBeginTypedefInfo statement. This statement specifies the type definition name and the type definition description.

►► `//VBBeginTypedefInfo:—typedef_name—` `, "description"` ►

typedef_name

Fully qualified C++ type definition name.

description

Optional type definition description.

VBIncludes Statement for a Type Definition

The VBIncludes statement describes the include file that contains the type definition.

► `//VBIncludes:—<include_file>—` `include_define` ►

include_file

Header file that contains the type definition. If you want to specify a link dependency in your application make file, use double quotation marks (" ") to delimit the file name instead of the angle brackets (< >).

include_define

Optional statement that associates a variable with the include statement, enabling generated code to test for duplicate include statements.

VBPartDataFile Statement for a Type Definition

The optional VBPartDataFile statement specifies the file in which to save the type definition information. If you do not use this statement, the imported file name is used with a .vbb extension.

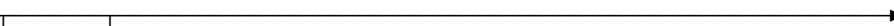
► `//VBPartDataFile:—file_name—` ►

file_name

File name in which to save the type definition interface information.

VB Statement for a Type Definition

The optional VB statement allows the information to be continued on the next statement line. This can be useful when a single statement exceeds a reasonable length. For example, you can use this statement to keep each line under 80 characters.

► `//VB:` 

VBEndTypedefInfo Statement

The VBEndTypedefInfo statement specifies the end of the interface information.

► `//VBEndTypedefInfo:—typedef_name` 

typedef_name

Type definition name matching the previous VBBeginTypedefInfo statement.

Sample Type Definition Information

The type definition `typedef int IBoolean;` in the `isynonym.hpp` file could be defined by the following .vbe file:

```
//VBBeginTypedefInfo: IBoolean
//VBIncludes: <isynonym.hpp> _ISYNONYM_
//VBPartDataFile: 'VBBase.vbb'
//VBEndTypedefInfo: IBoolean
```

Other Syntax Examples

An example of the `buttonClickEvent` event in the `IButton` part information file follows:

```
//VBEvent: buttonClickEvent,
//VB:      "Notification ID provided to observers when button is clicked"
//VB:      buttonClickId
```

An example of the `importFromFile` action with an `IString` parameter from the `IMLE` part information file follows:

```
//VBAction: importFromFile,
//VB:      "Inserts the contents of a file into the MLE"
//VB:      unsigned long,
//VB:      virtual unsigned long importFromFile(const char* fileName,
//VB:      EOLFormat type = cfText)
```



Chapter 7. 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 68.

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

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 3. Understanding and Using Event Notification

This part expands upon the basic concept of event notification discussed in Chapter 5, “Implementing a C++ Part.”

Chapter 8. The IBM Class Notification Framework	71
Notifiers and Observers	71
Notification Protocol	73
IBM C++ Notification Class Hierarchy	74
 Chapter 9. C++ Code to Enable Notification	 77
Code for an Observer Class	77
Code for a Notifier Class	79
Sample Notification Flow	79



Chapter 8. The IBM Class Notification Framework

This chapter provides an overview of the IBM class notification framework. You use this framework to implement event and attribute notification for visual and nonvisual parts. Developers coding to the IBM Open Class Library can also use it.

The notification framework is different from the previously existing event handler framework. Handlers are capable of stopping the dispatching of events to the remaining handlers in the chain. This is unsatisfactory for a notification framework, where registered observer objects must always be notified of an event regardless of how the event was handled.

The notification framework contains the following entities:

- Notifier objects that support the notifier protocol defined by the `INotifier` class
- Observer objects that support the observer protocol defined by the `IObserver` class
- Notification IDs, which are defined for parts that have been enabled for event notification
- Notification event objects defined by the `INotificationEvent` class

For examples of how the notification framework can be implemented in code, see Chapter 9, “C++ Code to Enable Notification” on page 77.

Notifiers and Observers

Notifier objects enable other objects in the system to register dependence upon the state of the notifier objects' properties. To register dependence, objects add an observer object to the notifier object by using the following function in the `IObserver` class:

```
virtual IObserver  
    &handleNotificationsFor (INotifier& aNotifier,  
                           const IEventData& userData = IEventData()),
```

The `IObserver` class also supports removing an observer from a notifier via the following:

```
virtual IObserver  
    &stopHandlingNotificationsFor (INotifier& aNotifier );
```

Notifier objects are responsible for publishing their supported notification events, managing the list of observers, and notifying observers when an event occurs. To notify observers of attribute changes or events, notifiers use the following member function defined by the `INotifier` class:

```
virtual INotifier  
    &notifyObservers (const INotificationEvent& anEvent) = 0;
```

The `INotifier` abstract base class defines the notifier protocol and requires its derived classes to completely implement its interface. To ensure that all notifier objects can coexist, no data is stored in any notifier object.

A notifier adds observers to an observer list and uses this list to notify observers in a first-in, first-notified manner.

The `IObserver` class defines the protocol that accepts event signals from the notifier object by overriding the member function in the `IObserver` class as follows:

```
virtual IObserver  
    &dispatchNotificationEvent (const INotificationEvent&)=0;
```

Because a single list of observers is kept for each notifier, all observers in the list get called when any notification occurs within the notifier. Each observer must test to determine if a given notification event should be processed. Normally, this is done by checking `notificationId` in an `INotificationEvent` object.

Notifier objects publish the notification events that they support by providing a series of unique identifiers in their interface. These *notification IDs* are string objects that are defined in the notifier. The string is in the form of the class name followed by the event name, such as `IStaticText::backgroundColor`. Each notification event provides a unique public static notification ID.

Events are typically a notification of changes in the attributes or intrinsic data that can be accessed in a notifier object. Attributes can represent any logical property of a part, such as the balance of an account, the size of a shipment, or the label of a push button.

A *notification event* is the data provided to an observer object when a change occurs in the attributes of an object. Included in this data is the identity of the attribute being changed and the part in which the change has occurred. Also, some of the data supplied to the observer can be the actual data being changed in the notifier object.

A notification event can also include observer-specific data. The caller that registers the observer with a notifier provides this data as the `userData` parameter on the following call in the `IObserver` class:

```
virtual IObserver
    &handleNotificationsFor (INotifier&          aNotifier,
                           const IEventData& userData = IEventData()),
```

The notifier passes this data to that observer anytime it notifies the observer of an event. To support the use of existing classes in a part-building tool, it is highly desirable to be able to derive from these classes and add all required notification behavior in the derived class. You do this by multiply inheriting from the base class and the `INotifier` class. You can then update the derived class to provide the required notifier behavior and the appropriate notification IDs. Ideally, the class can also provide the required notification behavior. Whether this can actually be done depends on the design of the base class.

Notification Protocol

Concrete classes that inherit from the `INotifier` class implement its protocol. This includes the following:

- Enabling, disabling, and querying the ability to signal events.

In general, notifiers are created disabled and must be enabled before they can signal events. This allows notifier objects to delay the setup to support notification until the notifier is enabled. (It also allows the Visual Builder connection objects to initialize themselves and related connection objects.)

The following member functions in the `INotifier` class enable you to enable and disable notification:

```
virtual INotifier
    &enableNotification    (Boolean enabled = true) = 0,
    &disableNotification  () = 0;
```

- Managing the collection of observers, including adding and removing observers. These are defined by the following protected members in `INotifier`:

```
virtual INotifier
    &addObserver          (IObserver&          anObserver,
                         const IEventData& userData) = 0,
    &removeObserver      (const IObserver& anObserver) = 0,
    &removeAllObservers  () = 0;
```

- Within the notifier object, calling the following member function every time an event of interest occurs:

```
    notifyObservers(const INotificationEvent&)
```

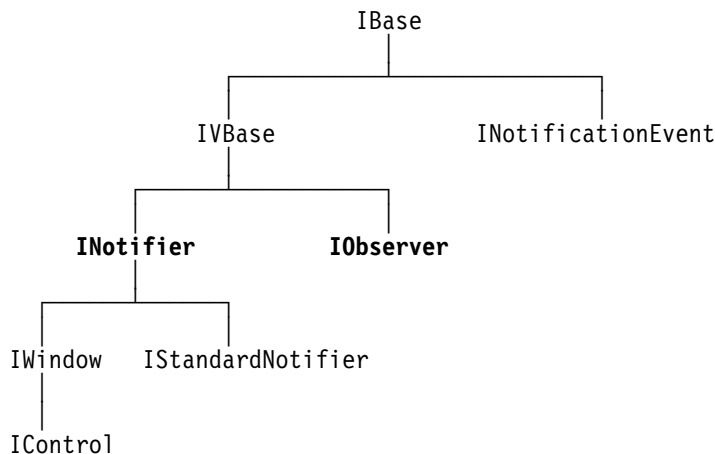
While the classes providing notification must call this function, in many cases it makes sense that the responsibility be delegated to another class. For instance, in the IBM Open Class User Interface Class Library, this responsibility is typically delegated to handler style objects.

The protected member `INotifier::addObserver` accepts a piece of typeless data as a `const IEventData&` that is forwarded to the `IObserver` instance with any notification request. This enables a piece of data to be maintained for each instance of an observer. (One concern in the window handler framework today is that data cannot be stored in a handler object because the handler might be handling many windows.)

The `IStandardNotifier` class provides the concrete implementation of the notifier protocol and provides the base support for nonvisual parts. The notifier protocol is also supported in the subclasses of `IWindow`. These classes inherit from a notifier class that supports registration of and notification to observer objects. The notification under the `IWindow` classes occurs primarily using the existing handler classes.

Note: Notification does not work across multiple threads.

IBM C++ Notification Class Hierarchy



Within this partial hierarchy, note the following:

- The `INotifier` abstract class defines the notifier protocol.
- The `IObserver` abstract class defines the observer protocol.
- The `INotificationEvent` class implements the notification event object.
- The `IStandardNotifier` and `IWindow` classes are concrete implementations of the notifier protocol.
- Nonvisual parts would normally be derived from `IStandardNotifier`.

- Visual parts would normally be derived from `IWindow` or `IControl`.



Chapter 9. C++ Code to Enable Notification

Normally, parts are connected by using the Composition Editor; however, developers can also create code that uses the notification framework. This chapter describes a sample program that uses the notification framework to test the `IAccess` class. The following example shows how the `streetChangedAction` member function (action) in the `IAccessTest` class is run when the `street` attribute changes in the address object.

Code for an Observer Class

The first step is to code the connection or observer class to be used. The `StreetConnection` class in the `IAccessTest` example follows:

```
//*****  
// StreetConnection Event-to-Action Connection Class  
//*****  
class StreetConnection : public IObserver  
{  
public:  
    StreetConnection(IAccessTest * newTestTarget)  
    { iTestTarget = newTestTarget;          //Save source } ;  
    ~StreetConnection () {};  
  
protected:  
    IObserver &dispatchNotificationEvent (const INotificationEvent& anEvent)  
    {  
        if (IAccess::streetId == anEvent.notificationId())  
        {  
            try {iTestTarget->streetChangedAction();}  
            catch (IException& exc) {}  
        }  
        return *this;  
    } ;  
    IAccessTest * iTestTarget;  
} ;
```

The key points in the previous sample code follow:

- The `StreetConnection` class is publicly derived from the `IObserver` class.
- The connection target is required on the `StreetConnection` constructor and is saved in `iTestTarget` to be used later in the `dispatchNotificationEvent` member function.

- The dispatchNotificationEvent member function is defined, and the code checks if this event is the street-changed event as follows. streetId is the notification ID for the street attribute.

```
if (IAddress::streetId == anEvent.notificationId())
```

This is followed by the code (action) to execute if the test is true:

```
try {iTestTarget->streetChangedAction();}
catch (IException& exc) {}:
```

```
//*****
// main - Application entry point *
//*****
int main(int argc, char **argv) //Main procedure with no parameters
{
    IApplication::current(). //Get current
        setArgs(argc, argv); // and set command line parameters

    IAddress* sourceNotifier;
    IAddressTest* testTarget;
    testTarget = new IAddressTest ();
    sourceNotifier = new IAddress ();

    {
        StreetConnection * iStreet = new StreetConnection (testTarget);
        iStreet->handleNotificationsFor(*sourceNotifier);
    }
    {
        CityConnection * iCity = new CityConnection (testTarget);
        iCity->handleNotificationsFor(*sourceNotifier);
    }
    {
        StateConnection * iState = new StateConnection (testTarget);
        iState->handleNotificationsFor(*sourceNotifier);
    }
    {
        ZipConnection * iZip = new ZipConnection (testTarget);
        iZip->handleNotificationsFor(*sourceNotifier);
    }
    sourceNotifier->enableNotification();
    sourceNotifier->setStreet("");
    sourceNotifier->setCity("");
    sourceNotifier->setState("");
    sourceNotifier->setZip("");

    delete sourceNotifier;
    delete testTarget;
} /* end main */
```

Code for a Notifier Class

The following code creates a notifier object called `sourceNotifier` and an observer object called `testTarget` that is called when events in `sourceNotifier` are signalled:

```
testTarget = new IAddressTest ();
sourceNotifier = new IAddress ();
```

The following code creates a `StreetConnection` object and adds it as an observer to the `sourceNotifier`:

```
StreetConnection * iStreet = new StreetConnection (testTarget);
iStreet->handleNotificationsFor(*sourceNotifier);
```

The following code enables notification in `sourceNotifier` and changes the value of the street attribute in the `sourceNotifier` object:

```
sourceNotifier->enableNotification();
sourceNotifier->setStreet("");
```

Sample Notification Flow

Calling the `setStreet` member function in the `IAddress` class starts a flow of control that results in the calling of the `streetChangedAction` in the `IAddressTest` class.

Follow this flow starting with the `setStreet` member function in `IAddress`:

```
IAddress& IAddress::setStreet (const IString& aStreet)
{
    if (iStreet != aStreet)
    {
        iStreet = aStreet;
        IString eventData(iStreet);
        notifyObservers(INotificationEvent(streetId, *this,
                                           true, (void*)&eventData));
    } /* endif */
    return *this;
}
```

Because a street connection observer has been added to the address object and event notification has been enabled, running `notifyObservers` in `setStreet` causes `dispatchNotificationEvent` in `StreetConnection` to run, as follows:

```
IObserver &dispatchNotificationEvent (const INotificationEvent& anEvent)
{
    if (IAddress::streetId == anEvent.notificationId())
    {
        try {iTestTarget->streetChangedAction();}
        catch (IException& exc) {}
    }
    return *this;
} ;
```

When `notifyObservers` is called, all observers are called. Each observer must determine if this notification event should be processed. So the `dispatchNotificationEvent` in `StreetConnection` compares the notification ID, determines that this is a `streetId` notification event, and calls `streetChangedAction` in `IAddressTest`:

```
class IAddressTest : public IStandardNotifier
{
    streetChangedAction()
    { printf("The Street Attribute has been changed\n"); }
};
```

This completes the flow from the street notification (`streetId`) to the target action, `streetChangedAction`.

Part 4. Appendixes

The appendixes contain information for further reference.

Appendix A. IBM Open Class Library Conventions	83
File Extensions	83
File Names	84
Class Names, Function Names, and Data Member Names	84
Enumerations	85
Function Return Types	85
Function Arguments	86
Feature Names	86
Notification IDs	86
Other Standards	87
 Appendix B. Code Listings	 89
INotifier Header Code	90
IStandardNotifier Header Code	93
IObserver Header Code	96
INotificationEvent Header Code	99
IButton Header Code	102
Sample IAddress Part	107
 Appendix C. VisualAge for C++ Classes by Include and Data File	 121



Appendix A. IBM Open Class Library Conventions

This appendix introduces you to the conventions used in the IBM Open Class Library, as follows:

- File extensions
- File names
- Class, function, and data member names
- Enumerations
- Function return types
- Function arguments
- Feature names (attributes, actions, events), for Visual Builder only
- Notification IDs
- Other standards

File Extensions

.c	C code file or C++ template code file.
.cpp	C++ code file.
.def	Import module definition file.
.dll	Dynamic link library file.
.h	C header file or C++ template header file.
.hpp	C++ header file.
.inl	C++ inline functions file.
.lib	Library file.
.mak	Make file.
.rc	Resource file.
.rsp	Import linker automatic response file.
.vbb	Visual Builder class (binary) file.
.vbe	Part information (external) file.

File Names

All files provided by the IBM Open Class Library begin with the letter “i,” such as `iapp.hpp`. File names have a maximum of eight characters, including the “i.” Following is a list of the file name extensions that are used:

xxxxxxx.hpp IBM Open Class Library header file.

xxxxxxx.inl IBM Open Class Library inline functions.

The file name generally indicates the class or classes it contains. For example, the `iapp.hpp` file contains the `IApplication` and `ICurrentApplication` classes.

Class Names, Function Names, and Data Member Names

Class names are mixed case, with the first letter of each word capitalized, as in `ICurrentApplication`. All class names in the global name space begin with the letter “I.”

Function names and data member names are also mixed case, except the first letter is always lowercase, as in the `autoSize` data member. Here are some more general rules about class and function names:

- Acronyms are uppercase, as in `IDBCSBuffer` (DBCS is the acronym for double-byte character set). Other acronyms are GUI (graphical user interface) and DDE (dynamic data exchange).
- Abbreviations are mixed case, as `IPresSpaceHandle`, which is the class for presentation space handles.
- Functions that query begin with a prefix that implies a query is being conducted, such as *is* or *has*. The `IDragItem` class, for example, has the `isCopyable` function, which queries whether an object can be copied.
- Functions that render an object as a different type begin with the *as* prefix, as in `asUnsignedLong`, which renders an object as an unsigned long.
- Functions that provide enabling or disabling capabilities begin with the *enable* or *disable* prefix, respectively. The `IEntryField` class, for example, provides the `enableAutoScroll` function, which enables automatic scrolling.
- Functions that set something begin with the *set* prefix. The `setDefaultStyle` function, to follow the preceding example, is used to set the default style for a class.
- Functions that get something, however, have no *get* prefix. For example, many classes use the `defaultStyle` function to get the default style for that class.
- Functions that act on objects are verbs, such as *copy* and *move*.

- Function names and arguments are virtually self-explanatory. The following example would move the IWindow object `aWindow` to the position specified by the IPoint object `aPoint`.

```
aWindow.moveTo( aPoint );
```

- Many functions that toggle the state of an object are provided with an optional Boolean argument that you can use to perform the opposite action of the function. This allows the result of a prior query function to be used as an input argument, such as the following:

```
Boolean initialVisibility = isVisible();
hide();
/* Do some hidden work */
show(initialVisibility);
```

Enumerations

The conventions for enumeration types and enumerators follow:

- The first character of each enumeration name is uppercase, such as `Severity`. If two words are joined, each begins with an uppercase letter, such as `ExceptionType`.
- Enumerators use the same naming conventions as functions; they begin with lowercase letters, but if two words are joined, the second begins with an uppercase letter, such as `accessError`.

Function Return Types

The return types for the various types of functions follow:

- A testing function typically returns a Boolean (true or false), as follows:

```
Boolean isValid() const
```

- Other accessor functions typically return an object, as follows:

```
ISize size() const;           //Returns an object
IWindow* owner();             //Returns a pointer to an object
```

- Functions that act on an object return an object reference, as follows:

```
IWindow& hide();
```

This enables the chaining of function calls, as follows:

```
aWindow.moveTo(IPoint(10,10)).show();
```

Function Arguments

Function arguments are usually passed using the following conventions:

- Built-in types (ints, doubles) and enumerations are passed in by value.
- Objects are passed by reference (a `const` reference if the argument is not modified by the function).
- Optional objects are passed by pointer. For example, a 0 pointer can be passed.
- `IWindow` objects are usually passed by pointer.
- `IContainerObjects` are usually passed by pointer.
- Strings are passed as `const char *`. This enables you to pass either an `IString` or a literal character array.

Feature Names

Give actions, attributes, and events meaningful names, and start them with a lower-case letter to avoid confusion with part or class names. If you follow these simple conventions in choosing your feature names, it is easier for users of your parts to recognize the function of a feature:

- Name actions with phrases that indicate activities to be performed, together with an optional receiver of that activity. Examples of feature names for actions are `startTimer`, `openWindow`, `hide`, and `setFocus`.
- Name attributes with phrases that indicate the physical property they represent. Examples of feature names for attributes are `height`, `buttonLabel`, and `contents`.
- Name events with phrases that indicate activities that either have happened or are about to happen. Examples of feature names for events are `clicked`, `aboutToCloseWindow`, and `timeExpired`.

Note: Do not use feature names that start with *avl* or *vb*. These are reserved for use by Visual Builder.

Notification IDs

Notification IDs are defined in the header files as public static data members. The notification ID name is the attribute or event name followed by “Id.” The following is an example of the notification ID for the `buttonClick` event in the `IButton` class:

```
INotificationId const  
    buttonClickId;
```

The notification ID is implemented in the code files as strings with the class name followed by the attribute or event name. An example from IButton code file follows:

```
const INotificationId IButton::buttonClickId="IButton::buttonClick";
```

Other Standards

The following are additional standards followed by the IBM Open Class Library:

- Header files are wrapped to ensure that files are not included more than once. An example from the ibutton.hpp file follows:

```
#ifndef _IBUTTON_  
#define _IBUTTON_
```

- All functions that can be inlined are placed in separate .inl files with a user option (I_NO_INLINES) to determine whether they should be inlined into the application code. An example from the inotifev.hpp file follows:

```
#ifndef I_NO_INLINES  
#include <inotifev.inl>  
#endif
```

If you do not want to inline these functions, then define I_NO_INLINES.

- isynonym.hpp contains the names of the types and values that are in the global name space but do not begin with the letter “I.” If you have collisions with other libraries, you can change the names in isynonym.hpp.



Appendix B. Code Listings

This chapter presents sample code listings for the following IBM Open Class Library header files:

- INotifier (page 90)
- IStandardNotifier (page 93)
- IObserver (page 96)
- INotificationEvent (page 99)
- IButton (page 102)

This chapter also presents the following for a sample IAddress part:

- Header code (.hpp) (page 107)
- Source code (.cpp) (page 110)
- Test code (page 117)

There might be differences between these printed samples and the sample code shipped with Visual Builder. Where differences exist, use the shipped sample code.

INotifier Header Code

This abstract class contains the notification protocol.

```
#ifndef _INOTIFY_
#define _INOTIFY_
/*****
* FILE NAME: inotify.hpp
*
* DESCRIPTION:
* Declaration of the class:
* INotifier - Abstract Notifier protocol.
*
* COPYRIGHT:
* Licensed Materials - Property of IBM
* (C) Copyright IBM Corporation 1992, 1993, 1995
* All Rights Reserved
* US Government Users Restricted Rights - Use, duplication, or disclosure
* restricted by GSA ADP Schedule Contract with IBM Corp.
*****/
#ifndef _IVBASE_
#include <ivbase.hpp>
#endif
/*-----*/
/* Align classes on a four-byte boundary */
/*-----*/
#pragma pack(4)

// Forward declarations
class IObserver;
class IObserverList;
class INotificationEvent;
class IEventData;

typedef const char* const
    INotificationId;

class INotifier : public IVBase {
typedef IVBase
    Inherited;
```

INotifier public

public:

```
/*----- Constructors/Destructor -----*/
| This class is an abstract base class, so objects cannot be constructed.
|
|-----*/

/*----- Activation -----*/
| The following functions affect the ability of an INotifier to notify
| of events of interest:
|   enableNotification - Causes the notifier to send notifications to any
|                       observer objects added.
|   disableNotification - Causes the notifier to stop sending notifications to
|                       all observer objects added.
|   isEnabledForNotification - Returns true if a notifier is sending
|                             notifications to its observers.
|
|-----*/

virtual INotifier
    &enableNotification      ( Boolean enabled = true ) = 0,
    &disableNotification    ( ) = 0;

virtual Boolean
    isEnabledForNotification ( ) const = 0;

/*----- Observer Notification -----*/
| The following function is used to notify observers of a change in a notifier:
|   notifyObservers      - Notifies all observers in the notifier's collection.
|                       Each observer receives a notification event
|                       describing the identity of the notifier, the
|                       notification ID, and optional data provided by the
|                       specific instance of the notifier.
|
|-----*/

virtual INotifier
    &notifyObservers        ( const INotificationEvent& anEvent ) = 0;
```

INotifier protected and private

```
protected:
/*----- Observer Addition/Removal -----*/
| The following functions add and remove observers from the notifier's
| collection:
|   addObserver      - Adds an observer to the notifier's collection.
|   removeObserver   - Removes an observer from the notifier's collection.
|   removeAllObservers - Removes all observers from the notifier's collection.
|-----*/
virtual INotifier
  &addObserver      ( IObserver&      anObserver,
                     const IEventData& userData) = 0,
  &removeObserver   ( const IObserver& anObserver) = 0,
  &removeAllObservers ( ) = 0;

/*----- ObserverList -----*/
|   observerList      Returns the collection of IObservers.
|-----*/

virtual IObserverList
  &observerList      ( ) const = 0;

/*----- Observer Notification -----*/
| The following function is used to notify observers of a change in a notifier:
|   notifyObservers   - Notifies all observers in the notifier's collection.
|                       Each observer receives a notification event
|                       describing the identity of the notifier, the
|                       notification ID, and optional data provided by the
|                       specific instance of the notifier.
|-----*/
virtual INotifier
  &notifyObservers (const INotificationId& nId) = 0;

private:

friend class IObserver;
};

/*----- Resume compiler default packing -----*/
#pragma pack()

/*----- Inline Functions -----*/

#endif /* _INOTIFY_ */
```

IStandardNotifier Header Code

The IStandardNotifier class implements the notification (INotifier) protocol. Non-visual parts are derived from this class.

```
#ifndef _ISTDNTFY_
#define _ISTDNTFY_
/*****
 * FILE NAME: istdntfy.hpp
 *
 * DESCRIPTION:
 *   Declaration of the class:
 *   IStandardNotifier - Concrete implementation of the INotifier protocol.
 *
 * COPYRIGHT:
 *   Licensed Materials - Property of IBM
 *   (C) Copyright IBM Corporation 1992, 1993, 1995
 *   All Rights Reserved
 *   US Government Users Restricted Rights - Use, duplication, or disclosure
 *   restricted by GSA ADP Schedule Contract with IBM Corp.
 *****/
#ifndef _INOTIFY_
#include <inotify.hpp>
#endif
#ifndef _IEVTDATA_
#include <ievtdata.hpp>
#endif
/*-----*/
/* Align classes on a four-byte boundary */
/*-----*/
#pragma pack(4)

class IObservableList;
class IObservable;
class INotificationEvent;

class IStandardNotifier : public INotifier {
typedef IStandardNotifier
    Inherited;
/*****
 * An IStandardNotifier provides a concrete implementation of the INotifier
 * protocol. If you want to create classes that provide notification,
 * you can do so by inheriting from IStandardNotifier. Alternatively, you can
 * inherit from INotifier directly and provide the notifier protocol yourself.
 *****/
}
```

IStandardNotifier public

public:

```
/*----- Constructors/Destructor -----*/
| The only constructor for an IStandardNotifier object is the default
| constructor that accepts no parameters.
|-----*/
IStandardNotifier ( );
IStandardNotifier (const IStandardNotifier& partCopy);
IStandardNotifier& operator= (const IStandardNotifier& aPart);

virtual
~IStandardNotifier ( );

/*----- Activation -----*/
| The following functions affect the ability of a part to notify observers
| of events of interest:
|   enableNotification - Causes the part to send notifications to any
|                       observer objects added.
|   disableNotification - Causes the part to stop sending notifications to
|                       all observer objects added.
|-----*/
virtual IStandardNotifier
  &enableNotification      ( Boolean enable = true),
  &disableNotification    ( );

virtual Boolean
  isEnabledForNotification ( ) const;

/*----- Observer Notification -----*/
| The following function is used to notify observers of a change in a notifier:
|   notifyObservers      - Notifies all observers in a part's collection.
|-----*/
virtual IStandardNotifier
  &notifyObservers      ( const INotificationEvent& anEvent);

/*----- Notification Event Descriptions -----*/
| These INotificationId strings are used for all notifications that
| IStandardNotifier provides to its observers:
|   deleteId - Notification identifier provided to observers when the part
|             object is deleted.
|-----*/
static INotificationId const
  deleteId;
```

IStandardNotifier protected and private

```
protected:
/*----- Observer Addition/Removal -----*/
| The following functions add and remove observers from the notifiers
| collection:
|   addObserver      - Adds an observer to the part's collection.
|   removeObserver   - Removes an observer from the part's collection.
|   removeAllObservers - Removes all observers from the part's collection.
|-----*/
virtual IStandardNotifier
&addObserver      ( IObserver&      anObserver,
                    const IEventData& userData = IEventData(0)),
&removeObserver   ( const IObserver& anObserver),
&removeAllObservers ( );

/*----- ObserverList -----*/
|   observerList      - Returns the collection of IObservers
|-----*/
IObserverList
&observerList      ( ) const;

/*----- Observer Notification -----*/
| The following function is used to notify observers of a change in a notifier:
|   notifyObservers    - Notifies all observers in a part's collection.
|-----*/
virtual IStandardNotifier
&notifyObservers (const INotificationId& nId);

private:
IObserverList
*observers;
Boolean
enabled;

};

/*-----*/
/* Resume compiler default packing */
/*-----*/
#pragma pack()

#endif /* _ISTDNTFY_ */
```

IObserver Header Code

IObserver is an abstract class that can be subclassed and added to notifiers to receive event notification.

```
#ifndef _IOBSERVER_
#define _IOBSERVER_
/*****
* FILE NAME: iobserver.hpp
*
* DESCRIPTION:
* Declaration of the class:
* IObserver - Abstract Observer protocol.
*
* COPYRIGHT:
* Licensed Materials - Property of IBM
* (C) Copyright IBM Corporation 1992, 1993, 1995
* All Rights Reserved
* US Government Users Restricted Rights - Use, duplication, or disclosure
* restricted by GSA ADP Schedule Contract with IBM Corp.
*
*****/

#ifndef _IVBASE_
#include <ivbase.hpp>
#endif

#ifndef _IEVTDATA_
#include <ievtdata.hpp>
#endif

/*-----*/
/* Align classes on a four-byte boundary */
/*-----*/
#pragma pack(4)

// Forward declarations
class IObserver;
class INotificationEvent;
class INotifier;

class IObserver : public IVBase {
typedef IVBase
    Inherited;
```

IObserver public

```
public:
/*----- Constructors/Destructor -----
| This class is an abstract class, so objects cannot be constructed.
|
|-----*/
~IObserver ();

/*----- Notifier Attachment -----
| These functions permit attaching and detaching the observer object to/from
| a given notifier
|   handleNotificationsFor      - Attaches the observer to the argument
|                               INotifier object.
|   stopHandlingNotificationsFor - Detaches the observer from the argument
|                               INotifier object.
|-----*/
virtual IObserver
  &handleNotificationsFor ( INotifier&      aNotifier,
                           const IEventData& userData = IEventData()),
  &stopHandlingNotificationsFor ( INotifier&      aNotifier );
```

IObserver protected and private

```
protected:
/*----- Overrides -----*/
| The following function must be overridden in a subclass.
|   dispatchNotificationEvent - An object of a class that inherits from
|                               INotifier calls this function to notify an observer
|                               of a change in itself. The notification event
|                               also includes notification-specific information.
|-----*/
virtual IObserver
    &dispatchNotificationEvent ( const INotificationEvent&)=0;

private:

friend class INotifier;
friend class IObserverList;
};

/*-----*/
/* Resume compiler default packing */
/*-----*/
#pragma pack()

/*----- Inline Functions -----*/

#endif /* _IOBSERVER_ */
```

INotificationEvent Header Code

The class INotificationEvent provides the details of a notification event to an observer object as shown in the following example:

```
#ifndef _INOTIFEV_
#define _INOTIFEV_
/*****
 * FILE NAME: inotifev.hpp
 *
 * DESCRIPTION:
 *   Declaration of the class:
 *   INotificationEvent - The details of a notification to an IObserver
 *   object.
 *
 * COPYRIGHT:
 *   Licensed Materials - Property of IBM
 *   (C) Copyright IBM Corporation 1992, 1993, 1995
 *   All Rights Reserved
 *   US Government Users Restricted Rights - Use, duplication, or disclosure
 *   restricted by GSA ADP Schedule Contract with IBM Corp.
 *****/
#endif

#include <inotify.hpp>
#endif

#ifndef _IEVTDATA_
#include <ievtdata.hpp>
#endif

/*-----*/
/* Align classes on a four-byte boundary */
/*-----*/
#pragma pack(4)

class INotificationEvent : public IBase {
typedef IBase
    Inherited;
/*****
 * The class INotificationEvent provides the details of a notification event
 * to an observer object. Included in the event is the notification ID and
 * the notifier object. Optionally included in the event is notifier-specific
 * data (see the notifier for details) and observer-specific data provided
 * to the notifier when the observer was added to the notifier.
 *****/
}
```

INotificationEvent public

public:

enum EventType { attribute=1, event};

/*----- Constructors/Destructor -----
-----*/

```
INotificationEvent ( const INotificationId&   anId,  
                    INotifier&               aNotifier,  
                    Boolean                  notifierAttrChanged=true,  
                    const IEventData&        eventData=IEventData(),  
                    const IEventData&        observerData=IEventData());
```

```
INotificationEvent (const INotificationEvent& anEvent);
```

/*----- Accessors -----
-----*/

```
INotificationEvent  
&setNotifierAttrChanged ( Boolean          changed=true),  
&setEventData           ( const IEventData& eventData ),  
&setObserverData        ( const IEventData& observerData);
```

```
INotificationId  
    notificationId      ( ) const;  
INotifier  
    &notifier           ( ) const;  
Boolean  
    pnotifierAttrChanged ( ) const;  
IEventData  
    eventData          ( ) const,  
    observerData       ( ) const;  
//
```


INotificationEvent protected and private

private:

```
INotificationId
    evtId;
INotifier
    *evtNotifier;
Boolean
    attrChanged;
IEventData
    evtData,
    obsData;
};
```

```
/*-----*/
/* Resume compiler default packing */
/*-----*/
#pragma pack()

/*----- Inline Functions -----*/
#ifndef I_NO_INLINES
    #include <inotifev.inl>
#endif
#endif /* _INOTIFEV_ */
```

IButton Header Code

IButton is an abstract class for button controls.

```
#ifndef _IBUTTON_
#define _IBUTTON_
/*****
* FILE NAME: ibutton.hpp
*
* DESCRIPTION:
* Declaration of the class:
* IButton - The IButton class is the abstract base class for button controls.
*
* COPYRIGHT:
* Licensed Materials - Property of IBM
* (C) Copyright IBM Corporation 1992, 1993, 1995
* All Rights Reserved
* US Government Users Restricted Rights - Use, duplication, or disclosure
* restricted by GSA ADP Schedule Contract with IBM Corp.
*****/
#ifndef _ITEXTCTL_
#include <itextctl.hpp>
#endif

#ifndef _IBUTTON1_
#include <ibutton1.hpp>
#endif

/*-----*/
/* Align classes on a four-byte boundary */
/*-----*/
#pragma pack(4)

// Forward declarations for other classes:
class IColor;

class IButton : public ITextControl {
typedef ITextControl
    Inherited;
/*****
* The IButton class is the abstract base class for button controls. This
* class contains the common functions for all button controls. Actual button
* controls are created by deriving from this base class.
*****/
```

```

public:
/*----- Style -----
    The following functions provide a means to set and query button styles:

    Style - Nested class that provides static members that define the set of
            valid button styles. These styles can be used in conjunction
            with the styles defined by the nested classes IWindow::Style and
            IControl::Style. For example, you could define an instance of
            the IButton::Style class and initialize it as follows:
                IButton::Style
                style = IControl::tabStop;
            An object of this type is provided when the button is created. A
            customizable default is used if no styles are specified. Once
            the object is constructed, you can use IButton, IWindow, and IControl
            member functions to set or query the object's style.

            The declaration of the IButton::Style nested class is generated
            by the INESTEDBITFLAGCLASSDEF2 macro.

    The valid button styles are:
        noPointerFocus - Buttons with this style do not set the focus to
                        themselves when the user clicks on them using the mouse.
                        This enables the cursor to stay on a control for which
                        information is required, rather than moving to the
                        button. This has no effect on keyboard interaction.
-----*/
INESTEDBITFLAGCLASSDEF2(Style, IButton, IWindow, IControl);
                        // style class definition
static const Style
    noPointerFocus;

/*----- Constructor/Destructor -----
| Instances of this class cannot be created. |
-----*/
    IButton ( );
virtual
    ~IButton ( );

```

```

/*----- Mouse Focus -----
| The following functions are used to set and query whether the button can
| receive the input focus when clicked with the mouse pointer:
|   enableMouseClickedFocus - Enables the button to receive the focus when the
|   user clicks on the button using the mouse.
|   disableMouseClickedFocus - Prevents the button from receiving the focus
|   when the user clicks on the button using the
|   mouse.
|   allowsMouseClickedFocus - Queries whether the button can receive the focus.
|-----*/
#ifndef IC_MOTIF_FLAGNOP
IButton
    &enableMouseClickedFocus ( Boolean turnOn = true ),
    &disableMouseClickedFocus ( );
Boolean
    allowsMouseClickedFocus ( ) const;
#endif // end of IC_MOTIF_FLAGNOP

/*----- Highlighted State -----
| These operations test and set a button's highlight state. A highlighted
| button has the same appearance as if the mouse selection button (mouse
| button 1) was pressed while the mouse pointer was over the button control:
|   isHighlighted - Returns true if the button's highlight state is set.
|   highlight     - Sets the button's highlight state.
|   unhighlight   - Turns off the button's highlight state.
|-----*/
#ifndef IC_MOTIF_FLAGNOP
Boolean
    isHighlighted ( ) const;

virtual IButton
    &highlight ( ),
    &unhighlight ( );
#endif // end of IC_MOTIF_FLAGNOP

/*----- Click the Button -----
|   click - Simulates the user clicking on the button control using the
|           mouse selection button.
|-----*/
virtual IButton
    &click ( );

```

```

#ifdef IC_PM
//ifndef IC_MOTIF
/*----- Color Functions -----*/
| foregroundColor      - Returns the foreground color value of the button
|                      - or the default if no color for the area has been
|                      - set.
| backgroundColor      - Returns the background color value of the button
|                      - or the default if no color for the area has been
|                      - set.
| disabledForegroundColor - Returns the disabled foreground color value of
|                      - the button or the default if no color for the
|                      - area has been set.
| hiliteForegroundColor - Returns the hilite foreground color value of the
|                      - button or the default if no color for the area
|                      - has been set.
| hiliteBackgroundColor - Returns the hilite background color value of the
|                      - button or the default if no color for the area
|                      - has been set.
|-----*/
virtual IColor
| foregroundColor      () const,
| backgroundColor      () const,
| disabledForegroundColor () const,
| hiliteForegroundColor () const,
| hiliteBackgroundColor () const;
//endif
#endif

/*----- Notification Event Descriptions -----*/
| These INotificationId strings are used for all notifications that IButton
| provides to its observers:
|   buttonClickId      - Notification identifier provided to observers when the
|                       - button control is clicked by the user.
|-----*/

// Attribute Change Notifications
static INotificationId const
| buttonClickId;

/*----- Overrides -----*/
| This class overrides the following inherited functions:
|   setText - Sets the text for the button and notifies a parent canvas to
|             - update the layout for its children, if appropriate.
|-----*/

virtual IButton
| &setText      ( const char* text ),
| &setText      ( const IResourceId& text );

```

```

protected:
private:
/*----- Private -----*/
    IButton      ( const IButton& );
IButton
    &operator=    ( const IButton& );

public:
/*----- Obsolete data and Functions -----*/
| The following enumerations are defined:
|   ColorArea - Used to replace the color for a particular region.
|   Values are:
|       foreground      - Sets the color of the foreground text.
|       disabledForeground - Sets the foreground color for disabled
|                           text.
|       background      - Sets the color of the background of
|                           the button window.
|       highlightForeground - Sets the foreground color for
|                           highlighted text.
|       border          - Sets the color of the border that
|                           surrounds the button window.
|   setColor - Changes the color of the given region.
|   color    - Returns the color of the given region.
|-----*/
enum ColorArea {
    foreground,
    background,
    disabledForeground,
    highlightForeground,
    border
};
IButton
    &setColor      ( ColorArea value, const IColor& color );
IColor
    color         ( ColorArea value ) const;
}; // class IButton

INESTEDBITFLAGCLASSFUNCS(Style, IButton);
// global style functions

/*-----*/
/* Resume compiler default packing */
/*-----*/
#pragma pack()

#endif /* _IBUTTON_ */

```

Sample IAddress Part

The IAddress nonvisual part contains several attributes, including street, city, state and zip code.

IAddress Header Code (iadd.hpp)

```
#ifndef _IADD_
#define _IADD_
/*****
* FILE NAME: iadd.hpp
*
* DESCRIPTION:
* Declaration of the class:
* IAddress - Address Class
*
* COPYRIGHT:
* Licensed Materials - Property of IBM
* (C) Copyright IBM Corporation 1994, 1995
* All Rights Reserved
* US Government Users Restricted Rights - Use, duplication, or disclosure
* restricted by GSA ADP Schedule Contract with IBM Corp.
*****/
#include <istring.hpp>
#include <istdntfy.hpp>
/*-----*/
/* Align classes on a four-byte boundary */
/*-----*/
#pragma pack(4)

class IAddress : public IStandardNotifier
{
typedef IStandardNotifier
    Inherited;
public:
/*----- PUBLIC -----*/
/*----- Constructors/Destructor -----*/
/*-----*/
    IAddress ();
    IAddress (const IAddress& partCopy);
virtual
    ~IAddress ();
    IAddress& operator= (const IAddress& aIAddress);
```

```

/*----- Attributes -----*/
| The following members support attributes for this class:
|   street          - Returns the street attribute.
|   city            - Returns the city attribute.
|   state           - Returns the state attribute.
|   zip             - Returns the zip attribute.
|   setStreet       - Sets the street attribute.
|   setCity         - Sets the city attribute.
|   setState        - Sets the state attribute.
|   setZip          - Sets the zip attribute.
|-----*/
virtual IString
    street () const,
    city () const,
    state () const,
    zip () const;

virtual IAddress
    &setStreet (const IString& aStreet),
    &setCity (const IString& aCity),
    &setState (const IString& aState),
    &setZip (const IString& aZip);

/*----- Actions -----*/
| These operations or services provided by this class:
|   setStreetToDefault - Sets street to a default value.
|   setCityToDefault   - Sets city to a default value.
|   setStateToDefault  - Sets state to a default value.
|   setZipToDefault    - Sets zip to a default value.
|   setToDefault       - Sets all attributes to their default values.
|-----*/
virtual IAddress
    &setStreetToDefault (),
    &setCityToDefault (),
    &setStateToDefault (),
    &setZipToDefault (),
    &setToDefault ();

```



```

/*----- Notification Event Descriptions -----*/
| These INotificationId strings are used for all notifications that IWindow
| provides to its observers:
|     streetId          - Notification identifier provided to observers
|                       when the street attribute changes.
|     cityId            - Notification identifier provided to observers
|                       when the city attribute changes.
|     stateId           - Notification identifier provided to observers
|                       when the state attribute changes.
|     zipId             - Notification identifier provided to observers
|                       when the zip attribute changes.
|-----*/
static INotificationId const
    streetId,
    cityId,
    stateId,
    zipId;

private:
/*----- PRIVATE -----*/
    IString iStreet;
    IString iCity;
    IString iState;
    IString iZip;
};

/*-----*/
/* Resume compiler default packing */
/*-----*/
#pragma pack()

#endif

```

IAddress Source Code (iadd.cpp)

```

/*****
* FILE NAME: iadd.cpp
*
* DESCRIPTION:
*   Class implementation of the class:
*   IAddress - Address Class
*
* COPYRIGHT:
*   Licensed Materials - Property of IBM
*   (C) Copyright IBM Corporation 1994, 1995
*   All Rights Reserved
*   US Government Users Restricted Rights - Use, duplication, or disclosure
*   restricted by GSA ADP Schedule Contract with IBM Corp.
*****/

#ifndef _IADD_
#include <iadd.hpp>                //IAddress class header
#endif

#ifndef _INOTIFEV_
#include <inotifev.hpp>
#endif

const INotificationId IAddress::streetId="IAddress::street";
const INotificationId IAddress::cityId="IAddress::city";
const INotificationId IAddress::stateId="IAddress::state";
const INotificationId IAddress::zipId="IAddress::zip";
```

```

/*-----
| IAddress::IAddress
| Standard constructor.
|-----*/
IAddress::IAddress() : Inherited (),
    iStreet("101 Main Street"),
    iCity("Hometown"),
    iState("NC"),
    iZip("27511")
{
}

/*-----
| IAddress::IAddress
| Standard copy constructor
|-----*/
IAddress::IAddress (const IAddress& partCopy)
    : Inherited (partCopy),
    iStreet(partCopy.street()),
    iCity(partCopy.city()),
    iState(partCopy.state()),
    iZip(partCopy.zip())
{
}

/*-----
| IAddress::~IAddress
| Empty destructor here for page tuning
|-----*/
IAddress::~IAddress()
{
}

```

```

/*-----
| IAddress::IAddress
| Standard operator=
|-----*/
IAddress& IAddress::operator= (const IAddress& aIAddress)
{
    if (this == &aIAddress) {
        return *this;
    } /* endif */
    Inherited::operator=(aIAddress);
    setStreet(aIAddress.street());
    setCity(aIAddress.city());
    setState(aIAddress.state());
    setZip(aIAddress.zip());
    return *this;
}

/*-----
| IAddress::street
| Returns the street attribute
|-----*/
IString IAddress::street () const
{
    return iStreet;
}

/*-----
| IAddress::setStreet
| Sets the street attribute
|-----*/
IAddress& IAddress::setStreet (const IString& aStreet)
{
    if (iStreet != aStreet)
    {
        iStreet = aStreet;
        IString eventData(iStreet);
        notifyObservers(INotificationEvent(streetId, *this,
            true, (void*)&eventData));
    } /* endif */
    return *this;
}

```

```

/*-----
| IAddress::city
| Returns the city attribute
|-----*/
IString IAddress::city () const
{
    return iCity;
}

/*-----
| IAddress::setCity
| Sets the city attribute
|-----*/
IAddress& IAddress::setCity (const IString& aCity)
{
    if (iCity != aCity)
    {
        iCity = aCity;
        IString eventData(iCity);
        notifyObservers(INotificationEvent(cityId, *this,
                       true, (void*)&eventData));
    } /* endif */
    return *this;
}

```

```

/*-----
| IAddress::state
| Returns the state attribute
|-----*/
IString IAddress::state () const
{
    return iState;
}

/*-----
| IAddress::setState
| Sets the state attribute
|-----*/
IAddress& IAddress::setState (const IString& aState)
{
    if (iState != aState)
    {
        iState = aState;
        IString eventData(iState);
        notifyObservers(INotificationEvent(stateId, *this,
                     true, (void*)&eventData));
    } /* endif */
    return *this;
}

```

```

/*-----
| IAddress::zip
| Returns the zip attribute
|-----*/
IString IAddress::zip () const
{
    return iZip;
}

/*-----
| IAddress::setZip
| Sets the zip attribute
|-----*/
IAddress& IAddress::setZip (const IString& aZip)
{
    if (iZip != aZip)
    {
        iZip = aZip;
        IString eventData(iZip);
        notifyObservers(INotificationEvent(zipId, *this,
                     true, (void*)&eventData));
    } /* endif */
    return *this;
}

/*-----
| IAddress::setStreetToDefault
| Performs the setStreetToDefault action
|-----*/
IAddress& IAddress::setStreetToDefault ()
{
    setStreet("101 Main Street");
    return *this;
}

/*-----
| IAddress::setCityToDefault
| Performs the setCityToDefault action
|-----*/
IAddress& IAddress::setCityToDefault ()
{
    setCity("Hometown");
    return *this;
}

```

```

/*-----
| IAddress::setStateToDefault
| Performs the setStateToDefault action
|-----*/
IAddress& IAddress::setStateToDefault ()
{
    setState("NC");
    return *this;
}

/*-----
| IAddress::setZipToDefault
| Performs the setZipToDefault action
|-----*/
IAddress& IAddress::setZipToDefault ()
{
    setZip("27511");
    return *this;
}

/*-----
| IAddress::setDefault
| Performs the setToDefault action
|-----*/
IAddress& IAddress::setDefault ()
{
    setStreetToDefault();
    setCityToDefault();
    setStateToDefault();
    setZipToDefault();
    return *this;
}

```


IAddress Test Code (iadd.cxx)

```
/******  
* FILE NAME: iadd.cxx *  
* * *  
* COPYRIGHT: *  
*   Licensed Materials - Property of IBM *  
*   (C) Copyright IBM Corporation 1994, 1995 *  
*   All Rights Reserved *  
*   US Government Users Restricted Rights - Use, duplication, or disclosure *  
*   restricted by GSA ADP Schedule Contract with IBM Corp. *  
*****/  
  
#include <iadd.hpp>  
#include <stdio.h>  
#ifndef _IOBSERV_  
    #include <iobservr.hpp>  
#endif  
#ifndef _INOTIFEV_  
    #include <inotifev.hpp>  
#endif  
#ifndef _IAPP_  
    #include <iapp.hpp>  
#endif  
  
class IAddressTest : public Inherited  
{  
public:  
    IAddressTest () {}  
    ~IAddressTest () {}  
  
    streetChangedAction()  
    { printf("The Street Attribute has been changed\n"); }  
  
    cityChangedAction()  
    { printf("The City Attribute has been changed\n"); }  
  
    stateChangedAction()  
    { printf("The State Attribute has been changed\n"); }  
  
    zipChangedAction()  
    { printf("The Zip Attribute has been changed\n"); }  
};
```

```

//*****
// StreetConnection Event-to-Action Connection Class
//*****
class StreetConnection : public IObserver
{
public:
    StreetConnection(IAddressTest * newTestTarget)
    { iTestTarget = newTestTarget;          //Save source } ;
    ~StreetConnection () {} ;

protected:
    IObserver& dispatchNotificationEvent (const INotificationEvent& anEvent)
    {
        if (IAddress::streetId == anEvent.notificationId())
        {
            try {iTestTarget->streetChangedAction();}
            catch (IException& exc) {}
        }
        return *this;
    } ;
    IAddressTest * iTestTarget;
} ;

//*****
// CityConnection Event-to-Action Connection Class
//*****
class CityConnection : public IObserver
{
public:
    CityConnection(IAddressTest * newTestTarget)
    { iTestTarget = newTestTarget;          //Save source } ;
    ~CityConnection () {} ;

protected:
    IObserver& dispatchNotificationEvent (const INotificationEvent& anEvent)
    {
        if (IAddress::cityId == anEvent.notificationId())
        {
            try {iTestTarget->cityChangedAction();}
            catch (IException& exc) {}
        }
        return *this;
    } ;
    IAddressTest * iTestTarget;
} ;

```

```

//*****
// StateConnection Event-to-Action Connection Class
//*****
class StateConnection : public IObserver
{
public:
    StateConnection(IAddressTest * newTestTarget)
    { iTestTarget = newTestTarget;          //Save source } ;
    ~StateConnection () {} ;

protected:
    IObserver& dispatchNotificationEvent (const INotificationEvent& anEvent)
    {
        if (IAddress::stateId == anEvent.notificationId())
        {
            try {iTestTarget->stateChangedAction();}
            catch (IException& exc) {}
        }
        return *this;
    } ;
    IAddressTest * iTestTarget;
} ;

//*****
// ZipConnection Event-to-Action Connection Class
//*****
class ZipConnection : public IObserver
{
public:
    ZipConnection(IAddressTest * newTestTarget)
    { iTestTarget = newTestTarget;          //Save source } ;
    ~ZipConnection () {} ;

protected:
    IObserver& dispatchNotificationEvent (const INotificationEvent& anEvent)
    {
        if (IAddress::zipId == anEvent.notificationId())
        {
            try {iTestTarget->zipChangedAction();}
            catch (IException& exc) {}
        }
        return *this;
    } ;
    IAddressTest * iTestTarget;
} ;

```

```

//*****
// main - Application entry point *
//*****
int main(int argc, char **argv) //Main procedure with no parameters
{
    IApplication::current(). //Get current
        setArgs(argc, argv); // and set command line parameters

    IAddress * sourceNotifier;
    IAddressTest * testTarget;
    testTarget = new IAddressTest ();
    sourceNotifier = new IAddress ();

    {
        StreetConnection * iStreet = new StreetConnection (testTarget);
        iStreet->handleNotificationsFor(*sourceNotifier);
    }
    {
        CityConnection * iCity = new CityConnection (testTarget);
        iCity->handleNotificationsFor(*sourceNotifier);
    }
    {
        StateConnection * iState = new StateConnection (testTarget);
        iState->handleNotificationsFor(*sourceNotifier);
    }
    {
        ZipConnection * iZip = new ZipConnection (testTarget);
        iZip->handleNotificationsFor(*sourceNotifier);
    }
    sourceNotifier->enableNotification();
    sourceNotifier->setStreet("");
    sourceNotifier->setCity("");
    sourceNotifier->setState("");
    sourceNotifier->setZip("");

    delete sourceNotifier;
    delete testTarget;
} /* end main */

```

Appendix C. VisualAge for C++ Classes by Include and Data File

Table 2 (Page 1 of 9). Class name, include and data files

Name	Header file	Data file
floatSamples	float.h	vbsample
IOString	i0string.hpp	
I3StateCheckBox	i3statbx.hpp	
<i>IABag</i>	iabag.h	vbcc
<i>IAccelerator</i>	iaccel.hpp	
<i>IAcceleratorKey</i>	iaccelky.hpp	
<i>IAcceleratorTable</i>	iacceltb.hpp	
<i>ICollection</i>	iacllct.h	
<i>IAddress</i>	iadd.hpp	vbsample
<i>IADeque</i>	iadeque.h	vbcc
<i>IAEqualityCollection</i>	iaequal.h	vbcc
<i>IAEqualityKeyCollection</i>	iaekey.h	vbcc
<i>IAEqualityKeySortedCollection</i>	iaeqsrt.h	vbcc
<i>IAEqualitySequence</i>	iaeqseq.h	vbcc
<i>IAEqualitySortedCollection</i>	iaeqsrt.h	vbcc
<i>IAHeap</i>	iaheap.h	vbcc
<i>IAKeyBag</i>	iakeybag.h	vbcc
<i>IAKeyCollection</i>	iakey.h	vbcc
<i>IAKeySet</i>	iakeset.h	vbcc
<i>IAKeySortedBag</i>	iaksbag.h	vbcc
<i>IAKeySortedCollection</i>	iaksrt.h	vbcc
<i>IAKeySortedSet</i>	iakssset.h	vbcc
<i>IAMap</i>	iamap.h	vbcc
IAnimatedButton	ianimbut.hpp	
<i>IAOrderedCollection</i>	iaorder.h	
<i>IApplication</i>	iapp.hpp	
<i>IPriorityQueue</i>	iaprioqu.h	vbcc
<i>IQueue</i>	iaqueue.h	vbcc
<i>IRelation</i>	iarel.h	vbcc
<i>ISequence</i>	iaseq.h	
<i>ISequentialCollection</i>	iasqntl.h	
<i>ISet</i>	iaset.h	vbcc
<i>ISortedBag</i>	iasrtbag.h	vbcc
<i>ISortedCollection</i>	iasrt.h	vbcc
<i>ISortedMap</i>	iasrtmap.h	vbcc
<i>ISortedRelation</i>	iasrtrel.h	vbcc
<i>ISortedSet</i>	iasrtset.h	vbcc
<i>IStack</i>	iastack.h	vbcc

Table 2 (Page 2 of 9). Class name, include and data files

Name	Header file	Data file
<i>IBase</i>	ibase.hpp	
<i>IBaseComboBox</i>	icombobs.hpp	
<i>IBaseListBox</i>	ilistbas.hpp	
<i>IBaseSpinButton</i>	ispinbas.hpp	
<i>IBitFlag</i>	ibitflag.hpp	
IBitmapControl	ibmpctl.hpp	
<i>IButton</i>	ibutton.hpp	
ICanvas	icanvas.hpp	
ICheckBox	icheckbx.hpp	
ICircularSlider	icslider.hpp	
<i>ICLibErrorInfo</i>	iexcept.hpp	
<i>IClipboard</i>	iclipbrd.hpp	
ICollectionViewComboBox	icombovw.hpp	
ICollectionViewListBox	ilistcvw.hpp	
<i>IColor</i>	icolor.hpp	
IComboBox	icombobx.hpp	
<i>ICompany</i>	icompany.hpp	vbsample
<i>IContainerColumn</i>	icnrcol.hpp	
<i>IContainerControl</i>	icnrcctl.hpp	
<i>IContainerObject</i>	icnrobj.hpp	
<i>IControl</i>	icontrol.hpp	
<i>ICurrentApplication</i>	iapp.hpp	
<i>ICurrentThread</i>	ithread.hpp	
ICustomButton	icustbut.hpp	
<i>ICustomer</i>	icust.hpp	vbsample
<i>IDate</i>	idate.hpp	
IDrawingCanvas	idrawcv.hpp	
<i>IDynamicLinkLibrary</i>	ireslib.hpp	
<i>IElemPointer</i>	iptr.h	vbcc
IEntryField	ientryfd.hpp	
<i>IErrorInfo</i>	iexcept.hpp	
<i>IFileDialog</i>	ifiledlg.hpp	
<i>IFlyOverHelpHandler</i>	iflyhhdr.hpp	
<i>IFlyText</i>	iflytext.hpp	
<i>IFont</i>	ifont.hpp	
<i>IFontDialog</i>	ifontdlg.hpp	
IFrameWindow	iframe.hpp	
IGraphicPushButton	igraphbt.hpp	
IGroupBox	igroupbx.hpp	
<i>IGUIErrorInfo</i>	iexcept.hpp	
<i>IHandle</i>	ihandle.hpp	
<i>IHandler</i>	ihandler.hpp	
<i>IHelpWindow</i>	ihelp.hpp	
IIconControl	iiconctl.hpp	
IInfoArea	iinfoa.hpp	

Table 2 (Page 3 of 9). Class name, include and data files

Name	Header file	Data file
IListBox	ilistbox.hpp	
IMenu	imenu.hpp	
<i>IMenuBar</i>	imenubar.hpp	
IMenuCascade	imnitem.hpp	
<i>IMenuHandler</i>	imenuhdr.hpp	
IMenuItem	imnitem.hpp	
IMenuSeparator	imnitem.hpp	
IMessageBox	imsgbox.hpp	
IMM24FramesPerSecondTime	immtime.hpp	vbmm
IMM25FramesPerSecondTime	immtime.hpp	vbmm
IMM30FramesPerSecondTime	immtime.hpp	vbmm
IMMAmpMixer	immamix.hpp	vbmm
IMMAudioCD	immcd.hpp	vbmm
IMMAudioCDContents	immcd.hpp	vbmm
<i>IMMConfigurableAudio</i>	immaud.hpp	vbmm
<i>IMMDevice</i>	immdev.hpp	vbmm
IMMDigitalVideo	immdigvd.hpp	vbmm
IMMErrorInfo	immexcpt.hpp	vbmm
<i>IMMFileMedia</i>	immfilem.hpp	vbmm
IMMHourMinSecFrameTime	immtime.hpp	vbmm
IMMHourMinSecTime	immtime.hpp	vbmm
IMMMasterAudio	immmaud.hpp	vbmm
IMMMillisecondTime	immtime.hpp	vbmm
IMMMinSecFrameTime	immtime.hpp	vbmm
<i>IMMPlayableDevice</i>	immplayd.hpp	vbmm
IMMPlayerPanel	immplypn.hpp	vbmm
<i>IMMRecordable</i>	immrecrd.hpp	vbmm
<i>IMMRemovableMedia</i>	immremed.hpp	vbmm
IMMSequencer	immsequ.hpp	vbmm
IMMSpeed	immspeed.hpp	vbmm
IMMTime	immtime.hpp	vbmm
IMMTrackMinSecFrameTime	immtime.hpp	vbmm
IMMWaveAudio	immwave.hpp	vbmm
IMngPointer	iptr.h	vbcc
IMultiCellCanvas	imcelcv.hpp	
IMultiLineEdit	imle.hpp	
INotebook	inotbk.hpp	
<i>INotifier</i>	inotify.hpp	
INumericSpinButton	ispinnum.hpp	
<i>IObserver</i>	iobsrvr.hpp	
IOrderedRecord	iordrrec.hpp	vbsample
ioSamples	io.h	vbsample
IOutlineBox	ioutlbox.hpp	
IPair	ipoint.hpp	
<i>IPartOrderedCollection</i>	ipartccl.h	vbcc

Table 2 (Page 4 of 9). Class name, include and data files

Name	Header file	Data file
<i>IPoint</i>	ipoint.hpp	
<i>IPointArray</i>	iptarray.hpp	
<i>IPopUpMenu</i>	ipopmenu.hpp	
<i>IProfile</i>	iprofile.hpp	
IProgressIndicator	islider.hpp	
IPushButton	ipushbut.hpp	
IRadioButton	iradiobt.hpp	
<i>IRange</i>	ipoint.hpp	
<i>IRBag</i>	irbag.h	vbcc
<i>IRDeque</i>	irdeque.h	vbcc
<i>IRecord</i>	irecord.hpp	vbsample
<i>IRectangle</i>	irect.hpp	
<i>IRefCounted</i>	irefcnt.hpp	
<i>IREqualitySequence</i>	ireqseq.h	vbcc
<i>IResource</i>	ireslock.hpp	
<i>IResourceId</i>	ireslib.hpp	
<i>IResourceLibrary</i>	ireslib.hpp	
<i>IRHeap</i>	irheap.h	vbcc
<i>IRKeyBag</i>	irkeybag.h	vbcc
<i>IRKeySet</i>	irkeyset.h	vbcc
<i>IRKeySortedBag</i>	irksbag.h	vbcc
<i>IRKeySortedSet</i>	irksset.h	vbcc
<i>IRMap</i>	irmap.h	vbcc
<i>IRPriorityQueue</i>	irprioqu.h	vbcc
<i>IRQueue</i>	irqueue.h	vbcc
<i>IRRelation</i>	irrel.h	vbcc
<i>IRSequence</i>	irseq.h	
<i>IRSet</i>	irset.h	vbcc
<i>IRSortedBag</i>	irsrtbag.h	vbcc
<i>IRSortedMap</i>	irsrtmap.h	vbcc
<i>IRSortedRelation</i>	irsrtrel.h	vbcc
<i>IRSortedSet</i>	irsrtset.h	vbcc
<i>IRStack</i>	irstack.h	vbcc
IScrollBar	iscroll.hpp	
ISetCanvas	isetcv.hpp	
<i>ISettingButton</i>	isetbut.hpp	
<i>ISize</i>	ipoint.hpp	
ISlider	islider.hpp	
ISplitCanvas	isplitcv.hpp	
<i>IStandardNotifier</i>	istdntfy.hpp	
IStaticText	istattxt.hpp	
<i>IString</i>	istring.hpp	
<i>IStringGenerator</i>	istrngen.hpp	
ISubmenu	isubmenu.hpp	
<i>ISystemErrorInfo</i>	iexcept.hpp	

Table 2 (Page 5 of 9). Class name, include and data files

Name	Header file	Data file
<i>ITextControl</i>	itextctl.hpp	
ITextSpinButton	ispintxt.hpp	
<i>ITime</i>	itime.hpp	
<i>ITitle</i>	ititle.hpp	
IToolBar	itbar.hpp	
IToolBarButton	itbarbut.hpp	
<i>ITrace</i>	itrace.hpp	
<i>IVAvlKeySortedSet</i>	ivksset.h	vbcc
<i>IVBag</i>	ivbag.h	vbcc
<i>IVBagOnBase</i>	ivbag.h	vbcc
<i>IVBagOnBSTKeySortedSet</i>	ivbag.h	vbcc
<i>IVBagOnHashKeySet</i>	ivbag.h	vbcc
<i>IVBagOnSortedDilutedSequence</i>	ivbag.h	vbcc
<i>IVBagOnSortedLinkedSequence</i>	ivbag.h	vbcc
<i>IVBagOnSortedTabularSequence</i>	ivbag.h	vbcc
<i>IVBase</i>	ivbase.hpp	
<i>IVBBooleanPart</i>	ivbbool.hpp	vbsample
<i>IVBCheckMenuHandler</i>	ivbmenuh.hpp	
IVBContainerControl	ivbcnr.h	
<i>IVBDataTypePart</i>	ivbdtype.hpp	vbsample
<i>IVBDoublePart</i>	ivbdb1.hpp	vbsample
<i>IVBDragDropHandler</i>	ivbdragh.hpp	
<i>IVBFileDialog</i>	ivbfiled.hpp	
<i>IVBFlyText</i>	ivbfly.hpp	
<i>IVBFontDialog</i>	ivbfontd.hpp	
<i>IVBLogicalAndPart</i>	ivbland.hpp	vbsample
<i>IVBLogicalOrPart</i>	ivblor.hpp	vbsample
<i>IVBLongPart</i>	ivblong.hpp	vbsample
<i>IVBMinSizeViewPortHandler</i>	ivbvpmsz.hpp	
<i>IVBShortPart</i>	ivbshort.hpp	vbsample
<i>IVBSTKeySortedSet</i>	ivksset.h	vbcc
<i>IVBStringPart</i>	ivbstrng.hpp	vbsample
<i>IVBUnsignedLongPart</i>	ivbulong.hpp	vbsample
<i>IVBUnsignedShortPart</i>	ivbushrt.hpp	vbsample
<i>IVDeque</i>	ivdeque.h	vbcc
<i>IVDequeOnBase</i>	ivdeque.h	vbcc
<i>IVDequeOnDilutedSequence</i>	ivdeque.h	vbcc
<i>IVDilutedSequence</i>	ivseq.h	vbcc
<i>IVEqualitySequence</i>	iveqseq.h	vbcc
<i>IVEqualitySequenceOnBase</i>	iveqseq.h	vbcc
<i>IVEqualitySequenceOnDilutedSequence</i>	iveqseq.h	vbcc
<i>IVEqualitySequenceOnTabularSequence</i>	iveqseq.h	vbcc
<i>IVGAvlKeySortedSet</i>	ivksset.h	vbcc
<i>IVGBag</i>	ivbag.h	vbcc
<i>IVGBagOnBSTKeySortedSet</i>	ivbag.h	vbcc

Table 2 (Page 6 of 9). Class name, include and data files

Name	Header file	Data file
<i>IVGBagOnHashKeySet</i>	ivbag.h	vbcc
<i>IVGBagOnSortedDilutedSequence</i>	ivbag.h	vbcc
<i>IVGBagOnSortedLinkedSequence</i>	ivbag.h	vbcc
<i>IVGBagOnSortedTabularSequence</i>	ivbag.h	vbcc
<i>IVGBSTKeySortedSet</i>	ivksset.h	vbcc
<i>IVGDeque</i>	ivdeque.h	vbcc
<i>IVGDequeOnDilutedSequence</i>	ivdeque.h	vbcc
<i>IVGDilutedSequence</i>	ivseq.h	vbcc
<i>IVGEqualitySequence</i>	iveqseq.h	vbcc
<i>IVGEqualitySequenceOnDilutedSequence</i>	iveqseq.h	vbcc
<i>IVGEqualitySequenceOnTabularSequence</i>	iveqseq.h	vbcc
<i>IVGHashKeyBag</i>	ivkeybag.h	vbcc
<i>IVGHashKeySet</i>	ivkeyset.h	vbcc
<i>IVGHeap</i>	ivheap.h	vbcc
<i>IVGHeapOnDilutedSequence</i>	ivheap.h	vbcc
<i>IVGKeyBag</i>	ivkeybag.h	vbcc
<i>IVGKeySet</i>	ivkeyset.h	vbcc
<i>IVGKeySetOnBSTKeySortedSet</i>	ivkeyset.h	vbcc
<i>IVGKeySetOnSortedDilutedSequence</i>	ivkeyset.h	vbcc
<i>IVGKeySetOnSortedLinkedSequence</i>	ivkeyset.h	vbcc
<i>IVGKeySetOnSortedTabularSequence</i>	ivkeyset.h	vbcc
<i>IVGKeySortedBag</i>	ivksbag.h	vbcc
<i>IVGKeySortedBagOnSortedDilutedSequence</i>	ivksbag.h	vbcc
<i>IVGKeySortedBagOnSortedTabularSequence</i>	ivksbag.h	vbcc
<i>IVGKeySortedSet</i>	ivksset.h	vbcc
<i>IVGKeySortedSetOnSortedDilutedSequence</i>	ivksset.h	vbcc
<i>IVGKeySortedSetOnSortedLinkedSequence</i>	ivksset.h	vbcc
<i>IVGKeySortedSetOnSortedTabularSequence</i>	ivksset.h	vbcc
<i>IVGLinkedSequence</i>	ivseq.h	vbcc
<i>IVGMap</i>	ivmap.h	vbcc
<i>IVGMapOnBSTKeySortedMap</i>	ivsrtmap.h	vbcc
<i>IVGMapOnBSTKeySortedSet</i>	ivmap.h	vbcc
<i>IVGMapOnHashKeySet</i>	ivmap.h	vbcc
<i>IVGMapOnSortedDilutedSequence</i>	ivmap.h	vbcc
<i>IVGMapOnSortedLinkedSequence</i>	ivmap.h	vbcc
<i>IVGMapOnSortedTabularSequence</i>	ivmap.h	vbcc
<i>IVGPriorityQueue</i>	ivprioqu.h	vbcc
<i>IVGQueue</i>	ivqueue.h	vbcc
<i>IVGQueueOnDilutedSequence</i>	ivqueue.h	vbcc
<i>IVGQueueOnTabularSequence</i>	ivqueue.h	vbcc
<i>IVGRelation</i>	ivrel.h	vbcc
<i>IVGSequence</i>	ivseq.h	
<i>IVGSet</i>	ivset.h	vbcc
<i>IVGSetOnBSTKeySortedSet</i>	ivset.h	vbcc
<i>IVGSetOnHashKeySet</i>	ivset.h	vbcc

Table 2 (Page 7 of 9). Class name, include and data files

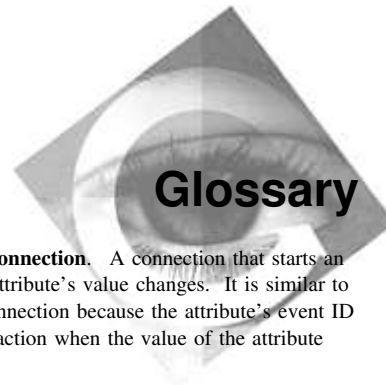
Name	Header file	Data file
<i>IVGSetOnSortedDilutedSequence</i>	ivset.h	vbcc
<i>IVGSetOnSortedLinkedSequence</i>	ivset.h	vbcc
<i>IVGSetOnSortedTabularSequence</i>	ivset.h	vbcc
<i>IVGSortedBag</i>	ivsrtbag.h	vbcc
<i>IVGSortedBagOnSortedDilutedSequence</i>	ivsrtbag.h	vbcc
<i>IVGSortedBagOnSortedLinkedSequence</i>	ivsrtbag.h	vbcc
<i>IVGSortedBagOnSortedTabularSequence</i>	ivsrtbag.h	vbcc
<i>IVGSortedMap</i>	ivsrtmap.h	vbcc
<i>IVGSortedMapOnSortedDilutedSequence</i>	ivsrtmap.h	vbcc
<i>IVGSortedMapOnSortedLinkedSequence</i>	ivsrtmap.h	vbcc
<i>IVGSortedMapOnSortedTabularSequence</i>	ivsrtmap.h	vbcc
<i>IVGSortedRelation</i>	ivsrtrel.h	vbcc
<i>IVGSortedRelationOnSortedDilutedSequence</i>	ivsrtrel.h	vbcc
<i>IVGSortedRelationOnSortedTabularSequence</i>	ivsrtrel.h	vbcc
<i>IVGSortedSet</i>	ivsrtset.h	vbcc
<i>IVGSortedSetOnBSTKeySortedSet</i>	ivsrtset.h	vbcc
<i>IVGSortedSetOnSortedDilutedSequence</i>	ivsrtset.h	vbcc
<i>IVGSortedSetOnSortedLinkedSequence</i>	ivsrtset.h	vbcc
<i>IVGSortedSetOnSortedTabularSequence</i>	ivsrtset.h	vbcc
<i>IVGStack</i>	ivstack.h	vbcc
<i>IVGStackOnDilutedSequence</i>	ivstack.h	vbcc
<i>IVGStackOnTabularSequence</i>	ivstack.h	vbcc
<i>IVGTabularSequence</i>	ivseq.h	vbcc
<i>IVHashKeyBag</i>	ivkeybag.h	vbcc
<i>IVHashKeySet</i>	ivkeyset.h	vbcc
<i>IVHeap</i>	ivheap.h	vbcc
<i>IVHeapOnBase</i>	ivheap.h	vbcc
<i>IVHeapOnDilutedSequence</i>	ivheap.h	vbcc
IVViewPort	ivport.hpp	
<i>IVKeyBag</i>	ivkeybag.h	vbcc
<i>IVKeyBagOnBase</i>	ivkeybag.h	vbcc
<i>IVKeySet</i>	ivkeyset.h	vbcc
<i>IVKeySetOnBase</i>	ivkeyset.h	vbcc
<i>IVKeySetOnBSTKeySortedSet</i>	ivkeyset.h	vbcc
<i>IVKeySetOnSortedDilutedSequence</i>	ivkeyset.h	vbcc
<i>IVKeySetOnSortedLinkedSequence</i>	ivkeyset.h	vbcc
<i>IVKeySetOnSortedTabularSequence</i>	ivkeyset.h	vbcc
<i>IVKeySortedBag</i>	ivksbag.h	vbcc
<i>IVKeySortedBagOnBase</i>	ivksbag.h	vbcc
<i>IVKeySortedBagOnSortedDilutedSequence</i>	ivksbag.h	vbcc
<i>IVKeySortedBagOnSortedTabularSequence</i>	ivksbag.h	vbcc
<i>IVKeySortedSet</i>	ivksset.h	vbcc
<i>IVKeySortedSetOnBase</i>	ivksset.h	vbcc
<i>IVKeySortedSetOnSortedLinkedSequence</i>	ivksset.h	vbcc
<i>IVKeySortedSetOnSortedTabularSequence</i>	ivksset.h	vbcc

Table 2 (Page 8 of 9). Class name, include and data files

Name	Header file	Data file
<i>IVLinkedSequence</i>	ivseq.h	vbcc
<i>IVMap</i>	ivmap.h	vbcc
<i>IVMapOnBase</i>	ivmap.h	vbcc
<i>IVMapOnBSTKeySortedSet</i>	ivmap.h	vbcc
<i>IVMapOnHashKeySet</i>	ivmap.h	vbcc
<i>IVMapOnSortedDilutedSequence</i>	ivmap.h	vbcc
<i>IVMapOnSortedLinkedSequence</i>	ivmap.h	vbcc
<i>IVMapOnSortedTabularSequence</i>	ivmap.h	vbcc
<i>IVPriorityQueue</i>	ivprioqu.h	vbcc
<i>IVPriorityQueueOnBase</i>	ivprioqu.h	vbcc
<i>IVQueue</i>	ivqueue.h	vbcc
<i>IVQueueOnBase</i>	ivqueue.h	vbcc
<i>IVQueueOnDilutedSequence</i>	ivqueue.h	vbcc
<i>IVQueueOnTabularSequence</i>	ivqueue.h	vbcc
<i>IVRelation</i>	ivrel.h	vbcc
<i>IVRelationOnBase</i>	ivrel.h	vbcc
<i>IVSequence</i>	ivseq.h	
<i>IVSequenceOnBase</i>	ivseq.h	
<i>IVSet</i>	ivset.h	vbcc
<i>IVSetOnBase</i>	ivset.h	vbcc
<i>IVSetOnBSTKeySortedSet</i>	ivset.h	vbcc
<i>IVSetOnHashKeySet</i>	ivset.h	vbcc
<i>IVSetOnSortedDilutedSequence</i>	ivset.h	vbcc
<i>IVSetOnSortedLinkedSequence</i>	ivset.h	vbcc
<i>IVSetOnSortedTabularSequence</i>	ivset.h	vbcc
<i>IVSortedBag</i>	ivsrtbag.h	vbcc
<i>IVSortedBagOnBase</i>	ivsrtbag.h	vbcc
<i>IVSortedBagOnSortedDilutedSequence</i>	ivsrtbag.h	vbcc
<i>IVSortedBagOnSortedLinkedSequence</i>	ivsrtbag.h	vbcc
<i>IVSortedBagOnSortedTabularSequence</i>	ivsrtbag.h	vbcc
<i>IVSortedMap</i>	ivsrtmap.h	vbcc
<i>IVSortedMapOnBase</i>	ivsrtmap.h	vbcc
<i>IVSortedMapOnSortedDilutedSequence</i>	ivsrtmap.h	vbcc
<i>IVSortedMapOnSortedLinkedSequence</i>	ivsrtmap.h	vbcc
<i>IVSortedMapOnSortedTabularSequence</i>	ivsrtmap.h	vbcc
<i>IVSortedRelation</i>	ivsrtrel.h	vbcc
<i>IVSortedRelationOnBase</i>	ivsrtrel.h	vbcc
<i>IVSortedRelationOnSortedDilutedSequence</i>	ivsrtrel.h	vbcc
<i>IVSortedRelationOnSortedTabularSequence</i>	ivsrtrel.h	vbcc
<i>IVSortedSet</i>	ivsrtset.h	vbcc
<i>IVSortedSetOnBase</i>	ivsrtset.h	vbcc
<i>IVSortedSetOnBSTKeySortedSet</i>	ivsrtset.h	vbcc
<i>IVSortedSetOnSortedLinkedSequence</i>	ivsrtset.h	vbcc
<i>IVSortedSetOnSortedTabularSequence</i>	ivsrtset.h	vbcc
<i>IVStack</i>	ivstack.h	vbcc

Table 2 (Page 9 of 9). Class name, include and data files

Name	Header file	Data file
<i>IVStackOnBase</i>	ivstack.h	vbcc
IVStackOnTabularSequence	ivstack.h	vbcc
IVTabularSequence	ivseq.h	vbcc
<i>IWindow</i>	iwindow.hpp	
mathSamples	math.h	vbsample
staticWindowSamples	iwindow.hpp	vbsample
stdioSamples	stdio.h	vbsample
stdlibSamples	stdlib.h	vbsample



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.



Bibliography

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.

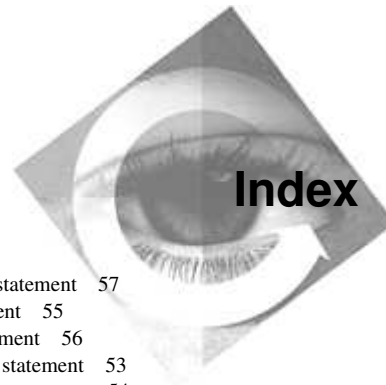
C and C++ Related Publications

- *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.



Index

A

- abstract nonvisual part 31
- action
 - definition 5
 - description 22
 - example 35
 - implementing 35
 - naming 30
- application segmentation 9
- architecture characteristics 19
- assignment operator, implementing 37
- attribute
 - defining get member function 33
 - defining set member function 34
 - definition 5
 - description 19
 - implementing 33
 - naming 30
 - notification ID 34
- attribute-to attribute connection definition 8
- attributeEvent-to-action connection definition 7

B

- benefits of using parts 5
- books
 - object-oriented programming and design xiv
 - user interface programming and design xiv
 - Visual Builder xiv
- business logic segment description 10

C

- c file description 83
- C++ construction from parts
 - architecture characteristics 19
 - definition 3
 - origin of architecture 17
 - origin of technology 4
- class
 - naming conventions 84
 - positioning a part 31
 - positioning an abstract part 31
 - relationship to part 6

class (continued)

- VB continuation statement 57
- VBAction statement 55
- VBAttribute statement 56
- VBBeginPartInfo statement 53
- VBComposerInfo statement 54
- VBConstraints statement 54
- VBConstructor statement 55
- VBEndPartInfo statement 57
- VBIncludes statement 53
- VBParent statement 53
- VBPartDataFile statement 54
- VBPreferredFeatures statement 57
- code listing
 - IAddress 107
 - IButton 102
 - INotificationEvent 99
 - INotifier 90
 - IObserver 96
 - IStandardNotifier 93
- coding conventions for notification IDs 86
- composite part example 3
- connecting parts 7
- connection definitions 7
- constructor, implementing 36
- copy constructor, implementing 36
- cpp file description 83

D

- data access segment description 10
- data member naming conventions 84
- def file description 83
- defining get member function 33
- defining set member function 34
- dependency manager 12
- design guidelines 29
- destructor, implementing 36
- dispatchNotificationEvent function
 - code example 77
 - overview 72
- distributing parts to others 67
- dll file description 83

E

enumeration

- naming conventions 85
- VB continuation statement 63
- VBBeginEnumInfo statement 62
- VBEndEnumInfo statement 64
- VBEnumerators statement 63
- VBIncludes statement 62
- VBPartDataFile statement 63

event

- definition 5
- description 22
- implementing 32
- naming 30

event-to-action connection definition 7

example

- action 22, 35
- attribute 19
- class information syntax 57
- composite part 3
- dispatchNotificationEvent 77
- enumeration information 64
- event notification 22
- function group information 61
- get member function 33
- handleNotificationsFor 77
- IAddress code 107
- IButton header code 102
- INotificationEvent header code 99
- INotifier header code 90
- IObserver header code 96
- IStandardNotifier header code 93
- nonvisual part 3
- notification code 77
- notification flow 79
- notification ID 34
- notifier class 79
- notifyObservers 79
- observer class 77
- part information 51
- primitive part 3
- set member function 34
- visual part 3

F

file extension descriptions 83

function argument type conventions 86

function group

- VB continuation statement 61
- VBAction statement 60
- VBBeginPartInfo statement 59
- VBComposerInfo statement 60
- VBConstraints statement 60
- VBEndPartInfo statement 61
- VBIncludes statement 59
- VBPartDataFile statement 59
- VBPreferredFeatures statement 61

function naming conventions 84

function return type conventions 85

G

get member function, defining 33

H

h file description 83

handleNotificationsFor function

- code example 77
- overview 71

highlighting conventions xii

hpp file description 83

I

IAddress part

- header code listing 107
- source code listing 110
- test code listing 117

IBM file name descriptions 84

IBM Open Class Library coding conventions 83

IButton header code listing 102

iconic editing 24

implementation checklists for part 38

implementation object 14

implementing action 35

implementing attribute 33

implementing event notification 32

inl file description 83

INotificationEvent class

- header file listing 99
- overview 72
- position in class hierarchy 74

INotifier class

- header file listing 90
- overview 72
- position in class hierarchy 74

- IObserver class
 - header file listing 96
 - position in class hierarchy 74
- IStandardNotifier header code listing 93

L

- lib file description 83

M

- mak file description 83
- model-view separation 11

N

- nonvisual part
 - abstract 31
 - iconic editing 24
 - implementing action 35
 - implementing assignment operator 37
 - implementing attribute 33
 - implementing constructor 36
 - implementing copy constructor 36
 - implementing notification 32
- nonvisual part example 3
- notification
 - class hierarchy 74
 - code examples 77
 - overview 71
 - protocol description 73
 - sample flow 79
- notification framework 12
- notification ID
 - coding conventions 86
 - overview 72
- notification ID for attribute 34
- notifier protocol
 - overview 71
- notifyObservers function
 - code example 79
 - overview 72

O

- object technology overview 9
- object-oriented programming and design books xiv
- observer protocol
 - overview 71

- origins of C++ construction from parts architecture 17

P

- part
 - behavior description 22
 - benefits of using 5
 - Composers, creating own 37
 - connecting 7
 - definition 3
 - description 5
 - design guidelines 29
 - implementation checklists 38
 - implementing action 35
 - implementing attribute 33
 - implementing notification 32
 - implementing virtual destructor 36
 - kinds supported 23
 - naming 30
 - notification description 22
 - positioning in class hierarchy 31
 - primitive visual, creating own 37
 - property description 19
 - relationship to class 6
 - sources 8
 - VB continuation statement 51
 - VBAction statement 49
 - VBAAttribute statement, general use 48
 - VBAAttribute statement, use with user primitive parts 47
 - VBBeginPartInfo statement 43
 - VBComposerInfo statement 44
 - VBConstraints statement 46
 - VBConstructor statement 47
 - VBEndPartInfo statement 51
 - VBEvent statement 48
 - VBIncludes statement 43
 - VBLibFile statement 44
 - VBParent statement 43
 - VBPartDataFile statement 44
 - VBPreferredFeatures statement 50
- part definition 3
- part information file
 - example 51
 - syntax for class 52
 - syntax for enumeration 62
 - syntax for function group 58
 - syntax for part 42
 - syntax for type definition 64
- parts, sharing with others 67

- portability publications 139
- positioning a part in the class hierarchy 31
- primitive part example 3
- primitive visual parts, creating own 37
- public interface
 - definition 5
- publications, portability 139
- publications, related xiii, 139

R

- rc file description 83
- reading syntax diagrams xii
- real-world object 14
- related publications xiii
- related publications, VisualAge for C++ 139
- relationship between parts and classes 6
- return type conventions for functions 85
- rsp file description 83

S

- separation of model from view 11
- service object 15
- set member function, defining 34
- sharing parts with others 67
- sources of parts 8
- special notices ix
- stopHandlingNotificationsFor function
 - overview 71
- syntax diagrams, how to read xii
- syntax, part information file 42

T

- trademarks ix
- type conventions for function arguments 86
- type definition
 - VB continuation statement 66
 - VBBeginTypedefInfo statement 65
 - VBEndTypedefInfo statement 66
 - VBIncludes statement 65
 - VBPartDataFile statement 65

U

- user interface programming and design books xiv
- user interface segment description 10

V

- VB continuation statement
 - class 57
 - enumeration 63
 - function group 61
 - part 51
 - type definition 66
- VBAction statement
 - class 55
 - function group 60
 - part 49
- VBAAttribute statement
 - class 56
 - part, general use 48
 - part, use with user primitive parts 47
- VBBeginPartInfo statement
 - class 53
 - function group 59
 - part 43
 - VBBeginEnumInfo statement for enumeration 62
 - VBBeginTypedefInfo statement for type definition 65
- VBComposerInfo statement
 - class 54
 - function group 60
 - part 44
- VBConstraints statement
 - class 54
 - function group 60
 - part 46
- VBConstructor statement
 - class 55
 - part 47
- vbe file syntax 42
- VBEndPartInfo statement
 - class 57
 - function group 61
 - part 51
 - VBEndEnumInfo statement for enumeration 64
 - VBEndTypedefInfo statement for type definition 66
- VBEnumerators statement 63
- VBEvent statement
 - part 48
- VBIncludes statement
 - class 53
 - enumeration 62
 - function group 59
 - part 43
 - type definition 65

- VLibFile statement
 - part 44
- VBParent statement
 - class 53
 - part 43
- VBPartDataFile statement
 - class 54
 - enumeration 63
 - function group 59
 - part 44
 - type definition 65
- VBPreferredFeatures statement
 - class 57
 - function group 61
 - part 50
- view object 14
- virtual destructor, implementing 36
- Visual Builder books xiv
- visual editing 24
- visual part
 - primitive, creating own 37
 - visual editing 24
- visual part example 3
- VisualAge for C++ publications 139

Communicating Your Comments to IBM

IBM VisualAge for C++ for Windows
Building VisualAge for C++ Parts for Fun and Profit
Version 3.5

Publication No. S33H-5036-00

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
Building VisualAge for C++ Parts for Fun and Profit
Version 3.5

Publication No. S33H-5036-00

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-5036-00

IBM®

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-5036-00

Cut or Fold
Along Line

IBM®

Part Number: 33H5036

Printed in U.S.A.

S33H-5036-00



33H5036

