IBM VisualAge COBOL

Visual Builder User's Guide

SC26-9053-02

IBM VisualAge COBOL

Visual Builder User’s Guide

SC26-9053-02

Note

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

Third Edition (April 1998)

This edition applies to Version 2.2 of IBM VisualAge COBOL 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 Corporation, W92/H3
P.O. Box 49023
San Jose, CA 95161-9023
U.S.A.

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.

Contents

Notices

Trademarks

About this book

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

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

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

What's new in this edition

There have been several significant additions and improvements made to this book. A few of these include:

Improved organization
There are have changes to the organization of the book. For example, information related to the Visual Builder interface has been grouped together in Part 2. Also, information listed in Chapter 15, "Hints and tips for using Visual Builder" on page 219 has been integrated into the book.

Inclusion of new information
Information previously found in the *Building Parts for Fun & Profit* book has been merged into this book.

Don't forget to look at the samples provided with VisualAge COBOL. There are Visual Builder samples that often provide more information than is covered in this book. You can access the samples from the **Guide to Samples** item in the VisualAge COBOL menu.

Who should use this book

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

How to use this book

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

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

How to read the syntax diagrams

Throughout this book, syntax for the compiler options is described using the structure defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following table shows the meaning of symbols at the beginning and end of syntax diagram lines.

Symbol	Indicates
>>–	the syntax diagram starts here
–>	the syntax diagram is continued on the next line
>–	the syntax diagram is continued from the previous line
–><	the syntax diagram ends here

  Diagrams of syntactical units other than complete statements start with the >– symbol and end with the –> symbol.

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

```
►►──STATEMENT──required item────────────────────────────────────────────►◄
```

- Optional items appear below the main path.

```
►►──STATEMENT──┬───────────────┬────────────────────────────────────────►◄
               └─optional item─┘
```

- When 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.

```
►►──STATEMENT──┬─required choice 1─┬────────────────────────────────────►◄
               └─required choice 2─┘
```

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

```
►►──STATEMENT──┬────────────────────┬─────────────────────────►◄
               ├─optional choice 1─┤
               └─optional choice 2─┘
```

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

```
               ┌─────────────────┐
►►──STATEMENT──▼─repeatable item─┴──────────────────────────►◄
```

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

  A comma or semicolon included in the repeat symbol indicates a separator that you must include between repeated parameters. These separators must be coded where shown.

- Keywords appear in mixed case letters (for example, VisualPart). They must be entered exactly as shown.
- Variables appear in all lowercase italic letters (for example, *item*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, they must be entered as part of the syntax.
- Use at least one blank or comma to separate parameters.

How this book is organized

This book is grouped into the following parts and appendixes:

- Part 1, “Introducing the Visual Builder” on page 1

  The first part introduces you to Visual Builder and its concepts.

- Part 2, “Touring Visual Builder” on page 29

  This part shows you how to work with the Visual Builder window and provides an overview of the Visual Builder editors.

- Part 3, “Developing Visual Builder applications” on page 77

  In this part, you learn how to add advanced features to a Visual Builder application, such as menus, online help, and additional windows. This part assumes you have gone through the Visual Builder tutorial in the *Getting Started* book.

- Part 4, “Extending Visual Builder applications” on page 217

  This part provides some additional information which may help you during your Visual Builder development.

- Appendix A, “Creating part information files” on page 223

  An appendix has been added that includes information previously found in *Building Parts for Fun & Profit*. This appendix describes the syntax of part information files.

- Glossary and Reader’s Comment Form

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

Highlighting conventions

This book uses the following highlighting conventions:

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

`Monospace` : COBOL coding examples; text that the user enters; messages within text. Examples follow:

> The following code from the `CFrameWindow` class illustrates ...
>
> The `street` method returns the current street address.

Italics : Emphasis of words; feature names; 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*.

Part 1. Introducing the Visual Builder

This part describes Visual Builder, its benefits, and its key concepts.

Introducing the Visual Builder

Chapter 1. Learning Visual Builder application development concepts

This chapter introduces you to the concepts, terms, and paradigms you need to understand in order to use the Visual Builder effectively. The first section gives a brief introduction to the object-oriented programming paradigm, and includes definitions of terms used to describe the features of Visual Builder. The second section defines a part, the fundamental building block in Visual Builder, and how you use parts to easily create complex applications. The third sections describes a part in detail, describing the features of a part, the different kinds of parts, and how parts interact with each other. The fourth, and final, section of this chapter describes the notification framework. Use the notification framework to define the order in which you want your parts to interact.

Object technology overview

This section provides a brief overview of some of the concepts and terminology of object-oriented programming. The information presented in this chapter is language-independent, but the application development methodology described is applicable to VisualAge COBOL, and forms the basis for the rest of the book. Subsequent chapters describe construction from parts specifically in relation to the VisualAge COBOL environment.

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 the computer is able to deal with the things 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 information” on page 243).

The application segmentation paradigm

The overall structure of an application developed using the guidelines presented in this book is shown in Figure 1 on page 4. This structure follows an application segmentation paradigm common in the cooperative processing environment, where an application is divided into three segments: user interface, business logic, and data access.

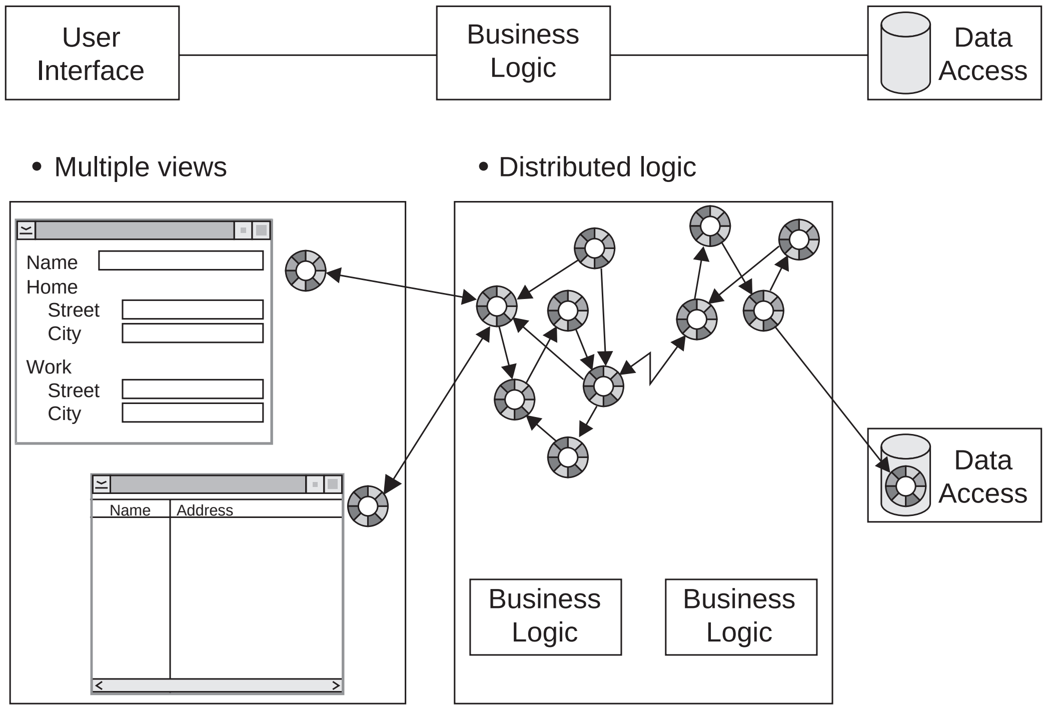


Figure 1. Overall application structure

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.

Segmenting an application in this way provides you with several benefits, even outside the 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 models.
- It facilitates cooperative processing.

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

Separation of models from views

To segment your application into manageable chunks, first separate your views from your models. By segmenting them, you can provide several views of the same model or models. Figure 2 shows two views (a detailed view and a tabular view) of a single model.

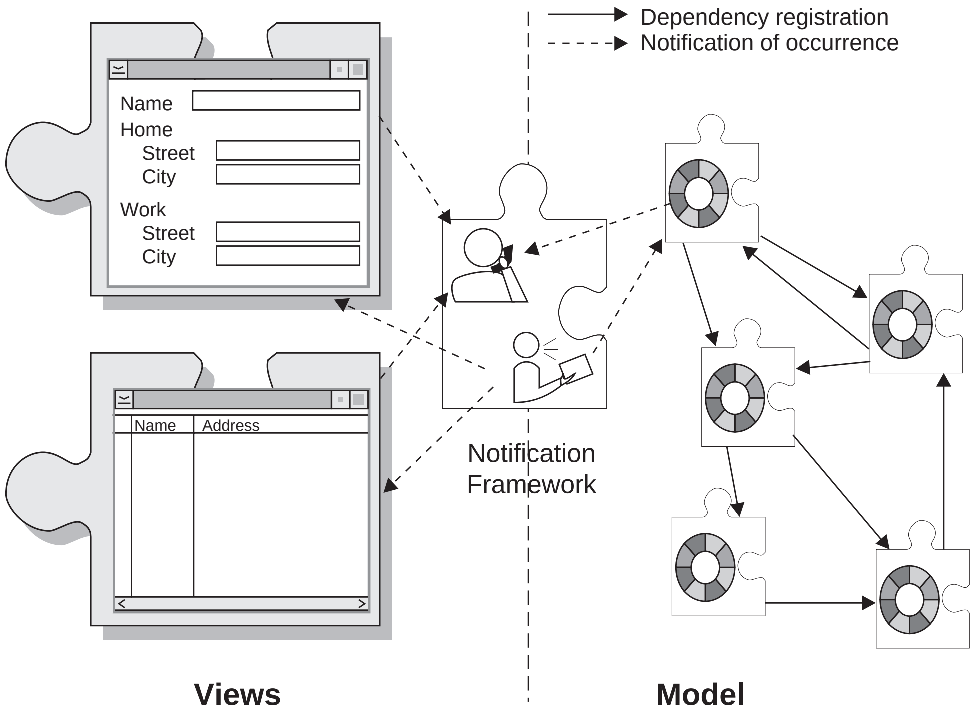


Figure 2. Connecting views to a model

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 models.

The dependency manager (in COBOL, 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 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 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 2 on page 5 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 figure is a person model. Because it maintains the data about a person, it has a dependency on the *nameChanged* event. Both views 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 view and then tab out of the field. The view 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 model receives the notification and updates its internal data. The tabularView view receives the notification and refreshes the display with the updated name.

Because the detailView view is also dependent on the *nameChanged* event, you might wonder why the notification was not forwarded back to it. Also, because the tabularView view 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 3 shows these divisions using the analogy of an iceberg, because most models contain many more hidden (nonvisual) objects than visible real-world objects.

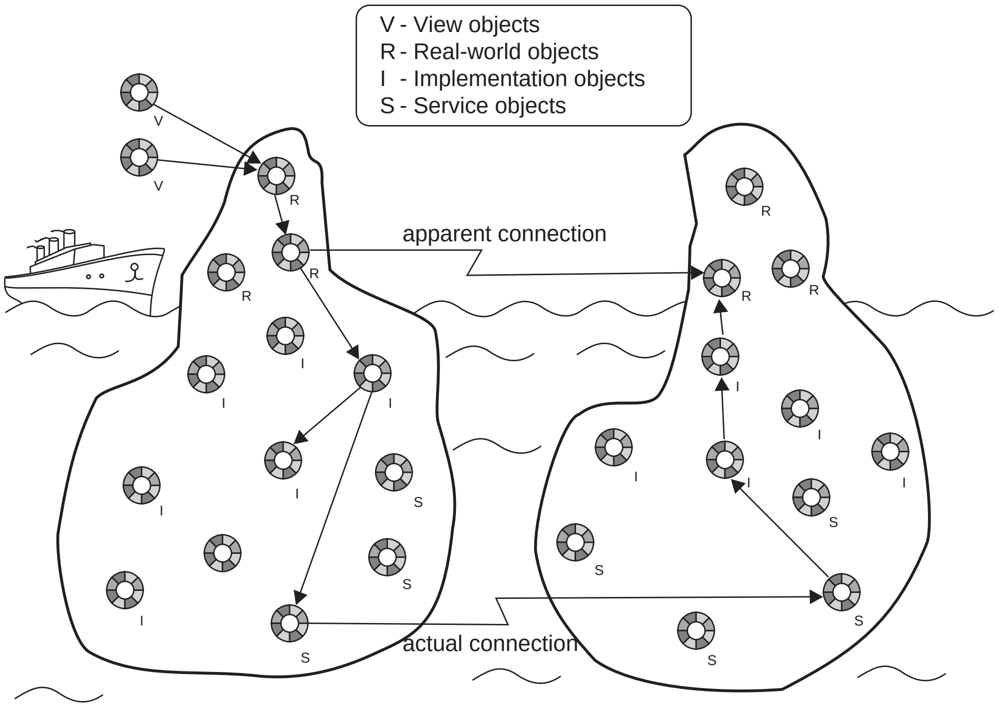


Figure 3. The Iceberg Model

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 development 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.

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.

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.

What is construction from parts?

Just about any construction project 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.

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 4 on page 9 shows a few examples.

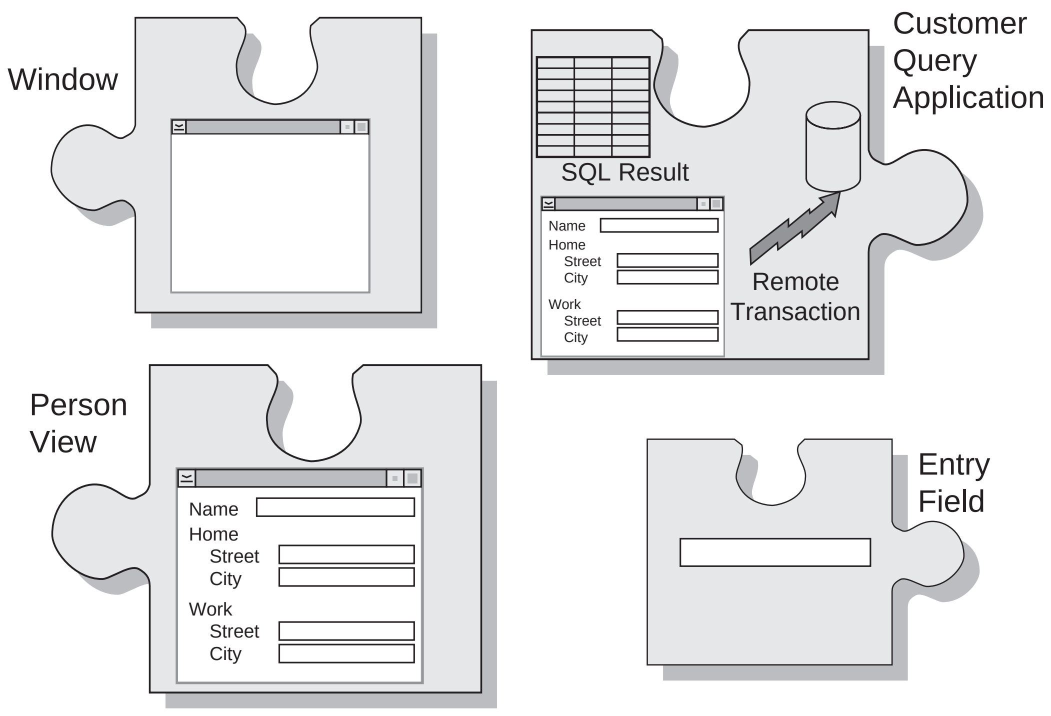


Figure 4. The range of primitive and composite parts

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 21.

Note that although construction from parts is an object-oriented methodology, the parts you use and reuse need not be written in object-oriented COBOL. There is great scope for exploiting existing code within the construction from parts methodology. For example, you can use construction from parts to build visual front-ends for your existing applications.

The origins of construction from parts

The 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 5 on page 10 compares the manufacturing process of constructing a computer system and the software process of constructing an application.

Introducing the Visual Builder

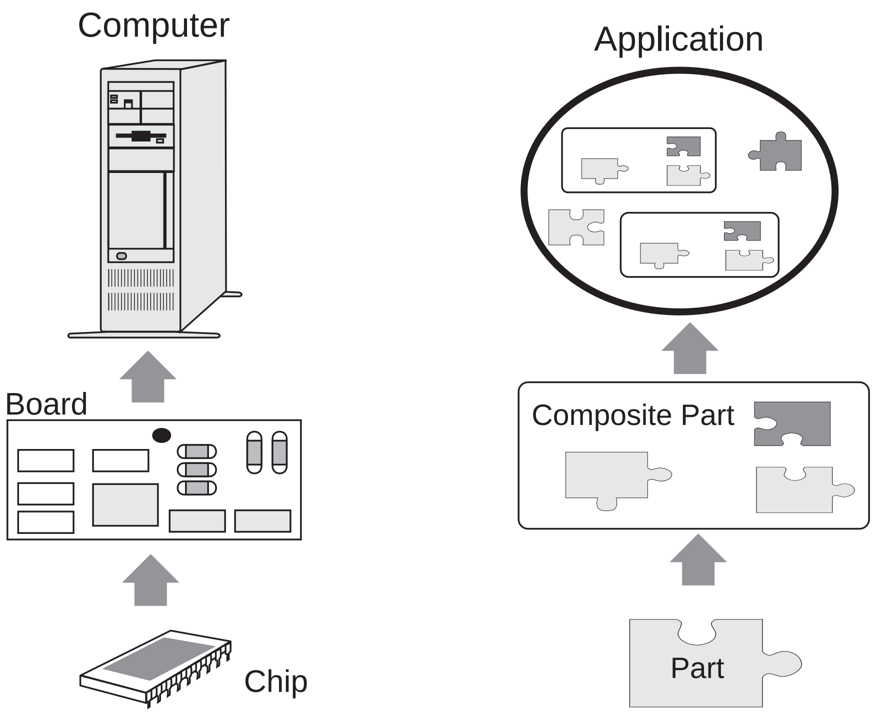


Figure 5. Construction from parts technology

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 you and your company can realize from using the 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 part?

A part is a software object implemented as a COBOL class[1] 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 6 on page 12 shows an example of a part interface.

1 Visual Builder supplied parts are not COBOL objects, but pointers managed by the CInterfaceManager class.

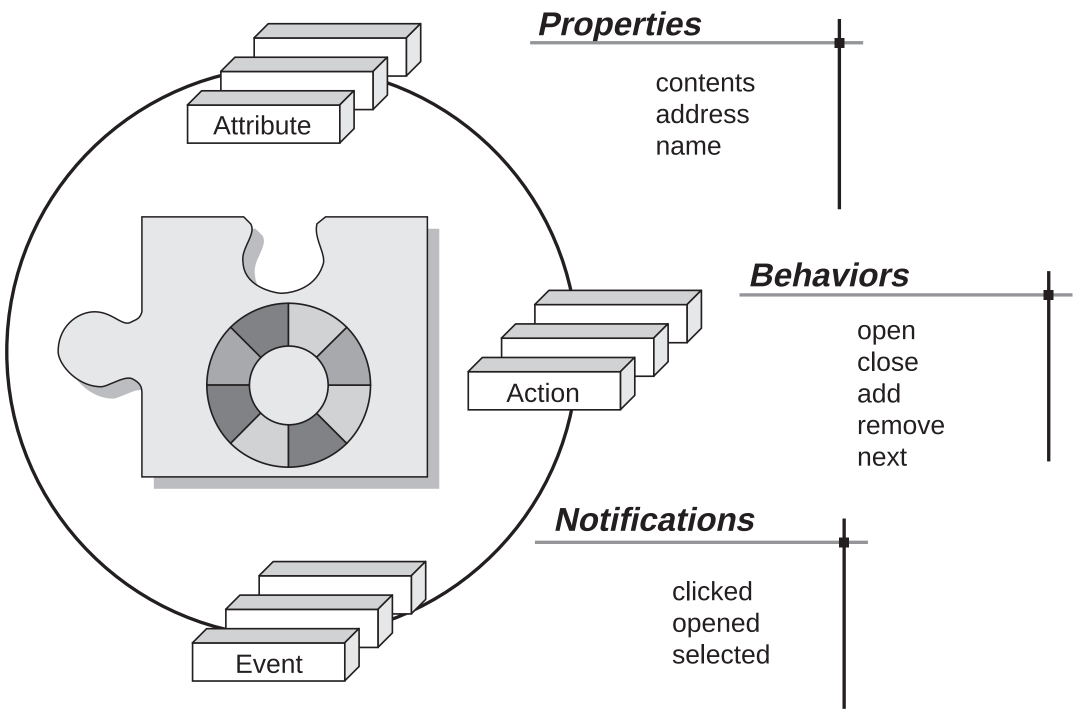


Figure 6. A part interface

- 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. You might also have thought a part is very similar to a class in object-oriented programming. In fact, parts and classes are closely related because COBOL 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 a COBOL class definition with some special characteristics, such as a method to handle events. 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.

A *part instance* is created from a part class and can be accessed using methods of the class. Each part instance has a unique set of values for the variables defined in the working-storage section of the class. These values distinguish one part instance from another. Object or subpart is used as shorthand for part instance. In COBOL, you might code an expression such as

```
  INVOKE BananaClass "somNew" RETURNING BananaInstance.
```

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 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.

“The part interface architecture” on page 14 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

In the Composition Editor of the Visual Builder, you can connect parts to define application flow. VisualAge COBOL supports the following part connections:

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

Refer to “Reviewing the key concepts” on page 25 for a complete list of the types of connections you can make.

Sources of parts

There are many possible sources for parts 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 building blocks of a VisualAge COBOL application. Entry fields and push buttons are examples of visual parts shipped with Visual Builder.

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 COBOL classes to enable them as parts, or by building parts just as you would any other COBOL class. The most common approach to constructing parts with VisualAge COBOL is to take a collection of existing COBOL programs and create a dynamic link library (DLL). Then create a part information file to create a part interface for the DLL. You can do this by using the `//VBComposerInfo: Programs` part information statement. See “Describing part interfaces in part information files” on page 223 for more information about part information statements.

The part interface architecture

The Construction from Parts architecture has been developed to give you an easier, more productive way to create high-quality applications in today's complex application development environment. Our ability to conceive this architecture has been enabled by the evolution of application structure, and by the emergence of application-building power tools.

The earliest batch 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.

Eventually these first-generation *batch* applications evolved into second-generation *transaction* applications, as shown in Figure 7. While this evolution did enable users to gain direct, on-line access to the applications through dumb terminals, it still left the applications in control of the users' interactions with the system.

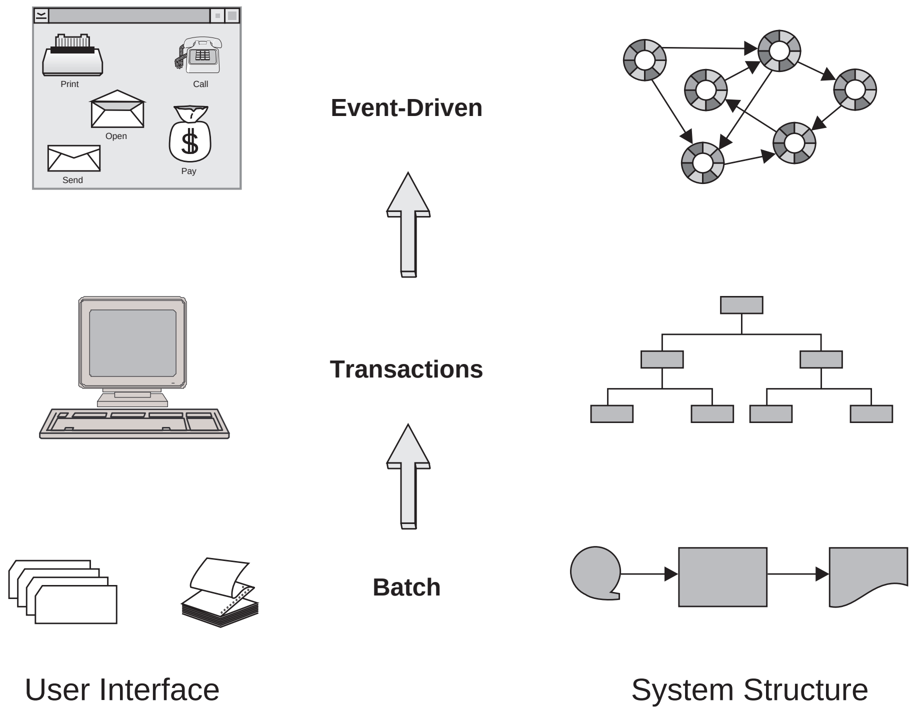


Figure 7. Evolution of application structure

Introducing the Visual Builder

As workstations became popular, another step in the evolution of application structure began. This evolutionary step came in response to the distributed nature of systems, the users' need 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 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.

Architecture characteristics

The Construction from Parts architecture formalizes this application structure, and facilitates the development of parts to be used in the new environment. 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 COBOL programming environment where applications are created using an editor, 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.

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 item defined as instance data in the Working-Storage section of the COBOL part class definition. The definition of the street in an address part is an example.
- An actual data item that is accessed via another part or via a system interface, such as the contents of an entry field.
- A computed data item that is a transformed version of an actual data item, such as the temperature in Fahrenheit when the actual data item is the temperature in Celsius.
- A computed data item 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 implemented in COBOL as a collection of definitions for methods to set or get the value of a property. We might for example have `getData` as the method to get the current value of the property; `setData` as the method to set the value of the property to some value; and `getEventId` as the method to get the notification ID for the property change event.

You can use the attribute interface by coding method invocation statements as follows:

```
INVOKE obj "getData" RETURNING aValue.
INVOKE obj "setData" USING aValue.
INVOKE obj "getEventId" RETURNING eventId.
```

where `aValue` is declared according to the requirements of the property, and `eventId` is a pointer used to determine the origin of an event.

The method that sets the value of the property can use the following expression to notify observer parts that the value of its property has changed:

```
INVOKE CNotificationEvent "somNew" returning event.
SET anObject TO SELF.
INVOKE event "initializeNotificationEvent" USING
   by value eventId by value anObject.
INVOKE SELF "notifyObservers" using by value event.
INVOKE event "somFree".
```

`notifyObservers` is the method that signals the event; `eventId` is the notification ID for the property change event; `anObject` is the notifier object. If you want to retrieve data from the notifier of an event, you can use the notifier object and a method in that object to obtain the data. (For more information about events, see “Notification of changes to parts” on page 19.)

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

While a property is often represented as instance data in 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 method 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 methods defined in Visual Builder's Part Interface Editor or in a part information file. For example, the following working storage definition and get and set methods support the attribute interface protocol for the temperature property of the `iThermometer` part:

```
 IDENTIFICATION DIVISION.
 CLASS-ID. iThermometer INHERITS CStandardNotifier, CObserver.

 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  iTemperature PIC 9(9) COMP-5.
 PROCEDURE DIVISION.

* getTemperature method
 IDENTIFICATION DIVISION.
 METHOD-ID. "getTemperature".
 DATA DIVISION.
 LINKAGE SECTION.
 01 Temperature PIC 9(9) COMP-5.
 PROCEDURE DIVISION RETURNING Temperature.
     MOVE iTemperature TO Temperature.
 END METHOD "getTemperature".

* setTemperature method
 IDENTIFICATION DIVISION.
 METHOD-ID. "setTemperature".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01 aThermometer USAGE OBJECT REFERENCE IThermometer.
 01 thermometerId USAGE POINTER.
 01 event USAGE OBJECT REFERENCE CNotificationEvent.
 LINKAGE SECTION.
 01 Temperature PIC 9(9) COMP-5.
 PROCEDURE DIVISION USING Temperature.
     MOVE Temperature TO iTemperature.
     INVOKE getTemperatureId RETURNING temperatureId.
     INVOKE CNotificationEvent "somNew" returning event.
     SET aThermometer TO SELF.
     INVOKE event "initializeNotificationEvent" USING
        by value temperatureId aThermometer.
     INVOKE SELF "notifyObservers" using by value event.
     INVOKE event "somFree".
 END METHOD "setTemperature".

* getTemperatureId method
 IDENTIFICATION DIVISION.
 METHOD-ID. getTemperatureId.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 initFlag PIC 9 VALUE 1.
 01 temperatureId USAGE POINTER.
 01 temperatureIdString PIC X(28)
      VALUE Z"iThermometer::temperatureId".
 LINKAGE SECTION.
 01  aTemperatureId USAGE POINTER.
 PROCEDURE DIVISION RETURNING aTemperatureId.
     IF initFlag = 1 THEN
       MOVE ZERO TO initFlag
       SET temperatureId TO ADDRESS OF temperatureIdString
     END-IF.
     SET aTemperatureId to temperatureId.
 END METHOD getTemperatureId.
```

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 methods to `getTempInCelsius` and `setTempInCelsius`. You could then implement two additional methods, such as the following, that return and set the temperature in Fahrenheit:

```
* getTempInFahrenheit method
 IDENTIFICATION DIVISION.
 METHOD-ID. "getTempInFahrenheit".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 LINKAGE SECTION.
 01 FTemperature PIC 9(9) COMP-5.
 PROCEDURE DIVISION RETURNING FTemperature.
     COMPUTE FTemperature = (iTemperature * (9/5)) + 32.
 END METHOD "getTempInFahrenheit".

* setTempInFahrenheit method
 IDENTIFICATION DIVISION.
 METHOD-ID. "setTempInFahrenheit".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01 Temperature PIC 9(9) COMP-5.
 LINKAGE SECTION.
 01 FTemperature PIC 9(9) COMP-5.
 PROCEDURE DIVISION USING FTemperature.
     COMPUTE Temperature = ((FTemperature - 32) * (5/9)).
     INVOKE SELF "setTempInCelsius" using Temperature.
 END METHOD "setTempInFahrenheit".
```

Notice that we did not introduce any additional instance data when we added these two new methods. 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 method (for example, `setTempInFahrenheit`) to change the value of a property. It also shows that a property is not always implemented as instance data.

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 implemented in COBOL as a method definition in the part class. You can use the action interface by coding method invocation statements as follows:

```
INVOKE "anActionMethod" USING parm1 ... RETURNING aValue.
```

where `anActionMethod` is the name of the method for the action to be performed; `parm1` is the first parameter (if any); and `aValue` is the return variable (if any).

A part implements the action interface by supplying a method that responds to the behavior defined in Visual Builder's Part Interface Editor or in a Part Information File. For example, the following method supports the action interface to set the default value of the temperature attribute in the `IThermometer` part:

```
* setDefaultTemp method
 IDENTIFICATION DIVISION.
 METHOD-ID. "setDefaultTemp".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01 defaultTemperature PIC 9(9) COMP-5 VALUE ZERO.
 PROCEDURE DIVISION .
     INVOKE SELF "setTempInCelsius" using defaultTemperature.
 END METHOD "setDefaultTemp".
```

This example shows that actions can cause values of attributes to change.

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 an event ID and a get method to retrieve it.

The event ID is the notification ID for the event and is implemented as a pointer to an address that uniquely identifies the event. When an instance of the class that will signal these events is created, the event ID must be initialized to the address of a data element in WORKING-STORAGE of the method used to get the event ID. This data element can be anything, but to simplify debugging it is conventionally a null-terminated string of the form `className::eventId`. The following sample shows a declaration for a notification ID (`eventId`), and a method (`getEventId`) to retrieve it:

```
01 eventId USAGE POINTER.

IDENTIFICATION DIVISION.
METHOD-ID. getEventId.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 eventIdString PIC X(19) VALUE Z"className::eventId".
LINKAGE SECTION.
01 anEventId USAGE POINTER.
PROCEDURE DIVISION RETURNING anEventId.
    SET anEventId TO ADDRESS OF eventIdString.
    GOBACK.
END METHOD getEventId.
```

To implement notification within a part, you can invoke `notifyObservers` in one or more of its methods. For example, the following `setTempInCelsius` method notifies other parts when the temperature has changed and when the temperature is above the boiling point of water:

```
      * setTempInCelsius method
       IDENTIFICATION DIVISION.
       METHOD-ID. "setTempInCelsius".
       DATA DIVISION.
       LINKAGE SECTION.
       01  newTEMP PIC 9(9) COMP-5.
       PROCEDURE DIVISION USING newTemp.
           IF iTemperature NOT EQUAL newTemp THEN
               MOVE newTemp TO iTemperature
               INVOKE CNotificationEvent "somNew" RETURNING event
               SET anObj TO SELF
               INVOKE event "initializeNotificationEvent"
                   USING BY VALUE temperatureId anObj
               INVOKE SELF "notifyObservers" USING BY VALUE event
               IF iTemperature > 100 THEN
                   INVOKE event "initializeNotificationEvent"
                       USING BY VALUE boilingId anObj
                   INVOKE SELF "notifyObservers" USING BY VALUE event
               END-IF
               INVOKE event "somFree"
           ENDIF.
       END METHOD "setTempInCelsius".
```

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.

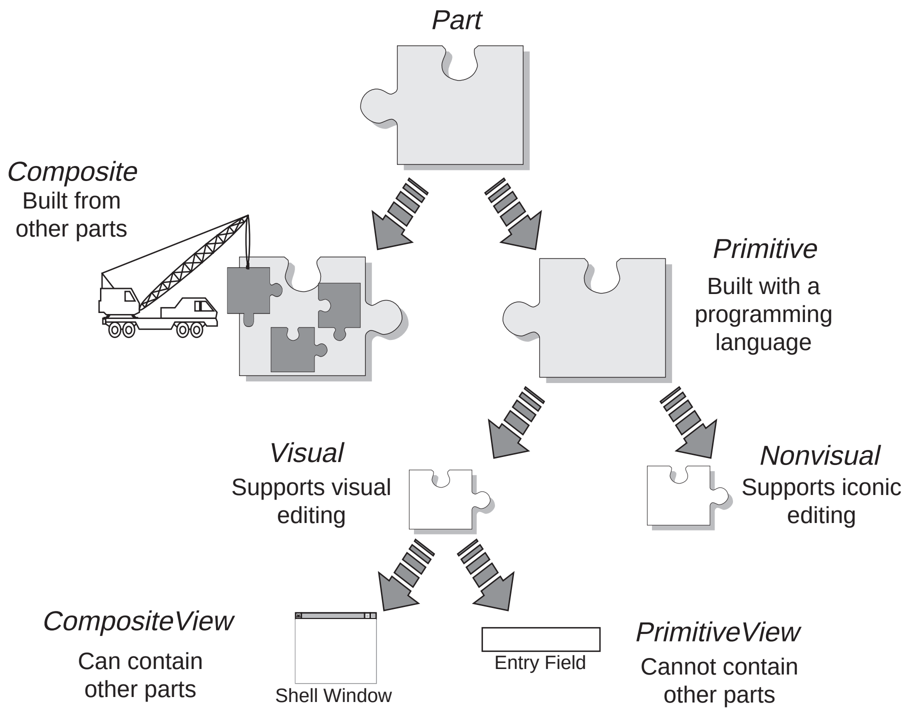


Figure 8. Kinds of Parts

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.

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 views. 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. You can use these parts in the Composition Editor in their visual runtime forms to create composite parts (*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 nonvisual parts are business logic parts, data table parts, communication access protocol parts, and database query parts.

The notification framework

You use the notification framework to implement event and attribute notification for visual and nonvisual parts. With previous versions of VisualAge COBOL, events were only predefined for IBM-supplied parts, and event routines followed specific naming conventions. Now you can define your own parts, and process events via the notification framework.

The notification framework contains the following entities:

- Notifier objects that support the notifier protocol defined by the CNotifier class
- Observer objects that support the observer protocol defined by the CObserver class
- Notification IDs, which are defined for parts that have been enabled for event notification
- Notification event objects defined by the CNotificationEvent class

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 invoking the `handleNotificationsFor` method on the observer object:

```
INVOKE anObserver "handleNotificationsFor"
    USING BY VALUE aNotifier.
```

The `CObserver` class also supports removing an observer from a notifier by invoking the `stopHandlingNotificationsFor` method on the observer object:

```
INVOKE anObserver "stopHandlingNotificationsFor"
    USING BY VALUE 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, objects invoke the `notifyObservers` method defined by the `CNotifier` class:

```
INVOKE aNotifier "notifyObservers" USING BY VALUE anEvent.
```

The `CNotifier` 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 `CObserver` class defines the protocol that accepts event signals from the notifier object. An observer can override the `processNotification` method of the `CObserver` class to implement the required event processing as follows:

```
* processNotification method
 IDENTIFICATION DIVISION.
 METHOD-ID. "processNotification" OVERRIDE.
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 LINKAGE SECTION.
 01 anEvent USAGE OBJECT REFERENCE CNotificationEvent.
 01 observer USAGE OBJECT REFERENCE.
 PROCEDURE DIVISION USING BY VALUE anEvent RETURNING observer.
     INVOKE anEvent "getNotificationId" RETURNING eventId.
     IF eventId = ...
     END-IF
     SET observer TO SELF
     GOBACK.
 END METHOD "processNotification".
```

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 a CNotificationEvent object. When the address stored in the eventId of the event is the same as the notificationId, code to process the notification event can be executed.

Notifier objects establish the notification events they support by providing a series of methods that return the eventIds. These methods must return a pointer to a fixed address in the Run Unit. Observers then monitor events for eventIds that are equal to this address.

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. When the observer needs additional data from the notifier about an event, it can use methods in the notifier to obtain that data.

To support the use of existing classes in a part-building tool, it is highly desirable to be able to derive new classes from the existing classes and add all required notification behavior in the derived class. You do this by multiply inheriting from the parent class and the `CNotifier` 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 parent class.

Notification protocol

Classes that inherit from the CNotifier 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.)

You can invoke methods of the `CNotifier` class to enable and disable notification, as follows:

```
INVOKE SELF "enableNotification".

INVOKE SELF "disableNotification".
```

- Within the notifier object, invoking the following method every time an event of interest occurs:

  ```
  INVOKE SELF "notifyObservers" USING anEvent.
  ```

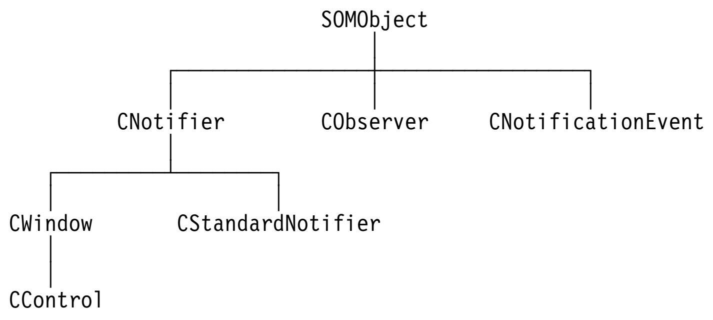
  While the classes providing notification must call this method, in many cases it makes sense that the responsibility be delegated to another class.

The CStandardNotifier 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 CWindow. These classes inherit from a notifier class that supports registration of and notification to observer objects. The notification under the CWindow classes occurs primarily using the existing handlers.

Note: Notification does not work across multiple threads.

IBM notification class hierarchy

The following diagram shows a partial hierarchy for classes in the notification framework:



Within this partial hierarchy, note the following:

- The `CNotifier` class defines the notifier protocol.
- The `CObserver` class defines the observer protocol.
- The `CNotificationEvent` class implements the notification event object.
- The `CStandardNotifier` and `CWindow` classes are concrete implementations of the notifier protocol.
- Nonvisual parts would normally be derived from `CStandardNotifier`.
- Visual parts would normally be derived from `CWindow`.

Introducing the Visual Builder

Chapter 2. What is Visual Builder?

Visual Builder is a visual programming tool that can help you create applications using the COBOL programming language. With Visual Builder, you can create applications with graphical user interfaces (GUI) faster and easier than you ever could using a text editor. Visual Builder provides a powerful visual editor, the Composition Editor, which enables you to create complete applications, often without writing code.

What are the benefits of using Visual Builder?

With Visual Builder, you can quickly create applications with advanced graphical user interfaces (GUIs). Visual Builder is part of VisualAge COBOL, a state-of-the-art COBOL application development product that includes a project management tool (WorkFrame), a live-parsing editor (LPEX), and a sophisticated debugger (Debugger).

With Visual Builder, you can adopt OO technology immediately. Other benefits of using the Visual Builder include the following:

- When you work with Visual Builder's visual programming tools, you are creating OO applications.
- You can also access other logic written in COBOL using Visual Builder's support for COBOL programs.
- You can shorten your application development cycle time considerably by creating reusable parts.
- By encapsulating the data of a part, you can change the way a part works without affecting its external interface. Dividing your application into parts also helps you deploy your business logic where it is needed. Critical calculations can be moved into shared libraries.
- By its nature, Visual Builder caters to all skill levels, so that programmers using Visual Builder can build not only simple applications but also complex ones.

Reviewing the key concepts

Chapter 1, "Learning Visual Builder application development concepts" on page 3 describes the important concepts behind application development using Visual Builder. In this section, we summarize some of the key concepts. Understanding these concepts will help you take advantage of the Visual Builder's features. The three key concepts are parts, connections, and source code generation. With the Visual Builder, you *connect* the *parts* and then *generate* the source code.

Parts

In Visual Builder, a *part* is a COBOL class that can send and receive events. A part has a well-defined *part interface*. The part interface defines how the part can interact with other parts.

Three kinds of *features* make up a part interface: attributes, actions, and events. It is with these features that you make connections. The following list provides a brief description of each kind of feature:

attributes	The logical data, usually stored in data items. This data can represent any property of a part, such as the balance of an account, the size of a shipment, or the text of a push button.
actions	Services or operations that a part can perform. Actions, such as placing an order or displaying a window, can be triggered by connections from other part's features.
events	Signals that a part can send to notify itself or other parts that a change has occurred. When events are connected to attributes or actions of other parts, the connections monitor these events and trigger the target features when the events occur. For example, when a push button's *press* event is connected to an action feature of another part, the other part's action is invoked when the push button is clicked.

Visual Builder provides a set of base parts that have attributes, actions, and events. These parts are included in the VAccess.vcb file, which Visual Builder loads each time it is started. Visual Builder does not allow you to modify these parts, but you can create *composite* parts. Composite parts are parts you create using the base parts provided by with the Visual Builder. For example, you can create a composite part based on a canvas that contains an entry field part. The composite part behavior will be similar to the behavior of an entry field.

Connections

To define how the parts interact with each other, you make connections. Connections are classified according to part interface features they connect. For instance, if you connect an event to an attribute, it is called an **event-to-attribute** connection. The following definitions apply to most cases in which these connections are used. Exceptions and special cases are noted in Chapter 9, "Learning to use connections" on page 153.

Attribute-to-attribute	Connections that link two data values together. When one changes, the other also changes.
Event-to-attribute	Connections that change the value of an attribute when a certain event occurs.
Event-to-action	Connections that start an action when a certain event occurs.
Attribute-to-action	Connections that start an action whenever an attribute's event identifier is signaled.
Parameter	Connections that provide a parameter value for an action. The parameter value can be provided by connecting a parameter to an attribute or an action with a return item.

Source code generation

Visual Builder can generate COBOL code for the graphical user interface (GUI) that you design in the Composition Editor, as well as for all of the connections that you make between parts. It can also generate COBOL code for any new parts that you create. You can then use the code that Visual Builder generates when you build your application. This capability allows you to concentrate on your application logic instead of spending time writing code for the GUI and its connections.

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

- Easier code modifications.

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

- Fewer errors.

  Because Visual Builder can generate the majority of the code for your application, there is less opportunity for human errors, such as typographical and syntax errors, to creep into your code. That means you spend less time debugging your code for minor errors.

- Support for pre-existing COBOL code.

  You can create part information files (.VCE) containing information about other COBOL classes and programs. You can then import that information into Visual Builder so that you can use those classes and programs as parts. Refer to Appendix A, “Creating part information files” on page 223 for more information on creating part information files (.VCE).

Introducing the Visual Builder

Part 2. Touring Visual Builder

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

Touring Visual Builder

Chapter 3. Getting acquainted with the Visual Builder window

This chapter describes the Visual Builder window and tasks you can perform from it. This chapter contains the following:

- Getting to know the Visual Builder window
- Working with part files
- Importing other types of files
- Customizing the information area
- Seeing the base files
- Seeing where part files are located
- Seeing the type list
- Using File Allocation Table (FAT) file names
- Setting the working directory
- Refreshing the display

Getting to know the Visual Builder Window

The Visual Builder window is shown in Figure 9.

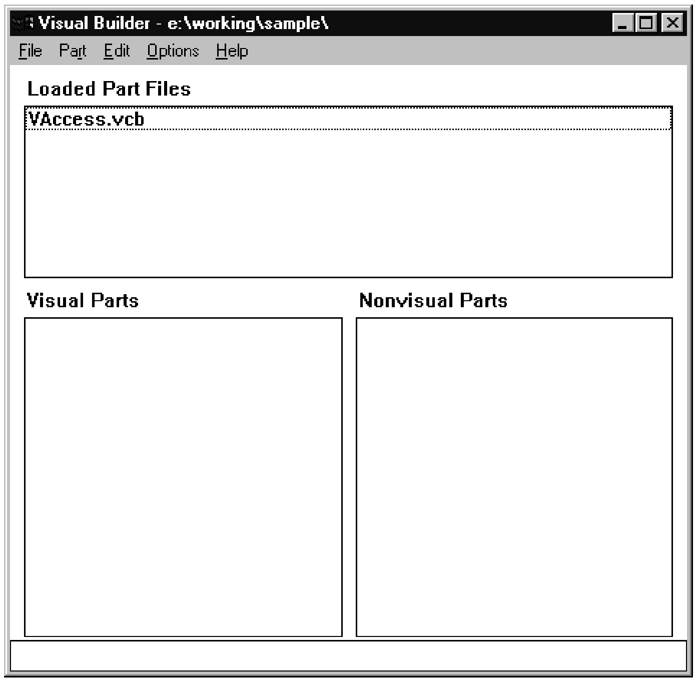

Visual Builder - e:\working\sample\
File Part Edit Options Help
Loaded Part Files
VAccess.vcb
Visual Parts
Nonvisual Parts

Figure 9. Visual Builder window

This window contains the following areas:

- The **Loaded Part Files** list box

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

This list box shows all of the part files currently loaded. Visual Builder provides one part file: VAccess.vcb. This part file contains the base parts that Visual Builder provides. This file is always loaded.

Note: VAccess.vcb is a read-only file. Store the parts you create in your own part files.

Part files must be loaded into Visual Builder for you to edit or refer to the parts they contain. Once part files are loaded, you can work on the part files and on the parts they contain using the Visual Builder's menu bar.

For more information about part files and the tasks you can perform on them, see "Working with part files" on page 33.

- The **Visual Parts** list box

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

- The **Nonvisual Parts** list box

  This list box displays the names of the nonvisual parts, the class interface parts, program parts, and data parts the selected part file contains. *Nonvisual parts* are parts your application uses to perform its functions, but the person using your application never sees them. Refer to "Constructing the part" on page 85 for more information on nonvisual parts.

- The **Loaded Type Information** list box

  You can view the **Loaded Type Information** list box by selecting **Options→Show type list** from the Visual Builder menu bar. This list box contains all types defined in the part file(s) you have selected. Types used by a part must be loaded when you edit that part or generate code. VAccess.vcb comes loaded with basic types. You can define your own types in a part information file and import the file into Visual Builder. Refer to Appendix A, "Creating part information files" on page 223 for more information on creating part information files.

You can access the same choices available in the menu bar by moving the mouse pointer over a list box and pressing mouse button 2 to display a pop-up menu. The pop-up menu contains only menu choices relevant to the contents the mouse points to in the list box .

The pop-up menu Visual Builder displays for the **Loaded Part Files** list box contains only menu choices pertaining to part files.

The pop-up menu for the **Visual Parts** list box only affects parts selected in the **Visual Parts** list box, even if parts are also selected in the **Nonvisual Parts** list box, and vice versa.

To simultaneously open both a visual and a nonvisual part, you must select **Part→Open** from the menu bar. If you select **Open** from the pop-up menu for the **Visual Parts** list box, you only open a visual part. The same is true if you select **Open** from the pop-up menu for the **Nonvisual Parts** list box.

The choices on the **Part** menu apply to all selected items in the **Loaded Part Files**, **Visual Parts**, **Nonvisual Parts**, and **Loaded Type Information** list boxes in the Visual Builder window.

Working with part files

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

Loading part files

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

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

   Visual Builder displays the window shown in Figure 10.

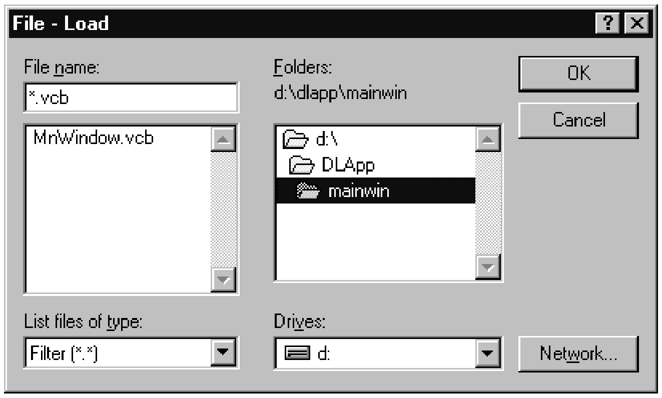


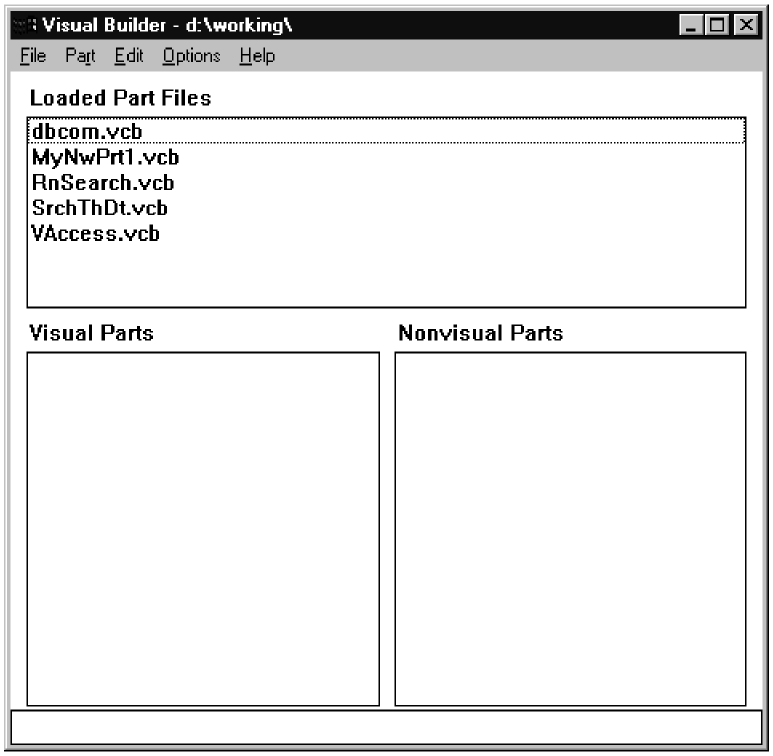
Figure 10. File–Load window

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

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

   The file name or names are displayed in the **Loaded Part Files** list box in the Visual Builder window. Figure 11 on page 34 shows the Visual Builder window with multiple part files loaded.

Touring Visual Builder



dbcom.vcb
MyNwPrt1.vcb
RnSearch.vcb
SrchThDt.vcb
VAccess.vcb

Figure 11. Visual Builder window with multiple part files loaded

Unloading part files

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

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

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

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

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

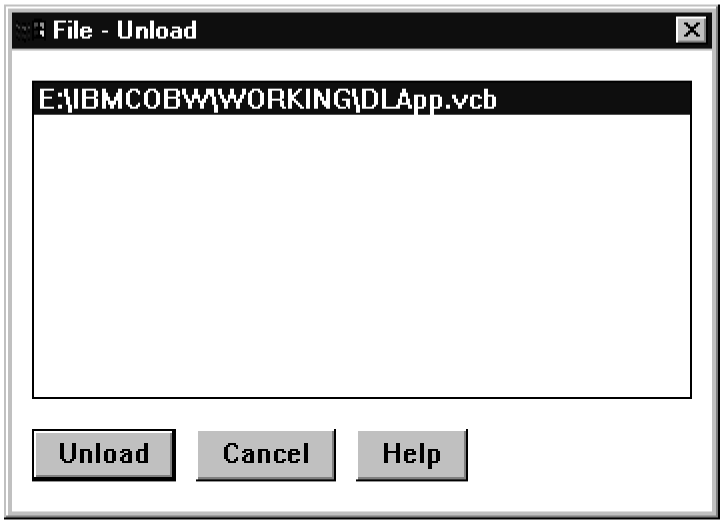


Figure 12. File — Unload window

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

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

   The window disappears and the file names are removed from the **Loaded Part Files** list box.

For information about loading files, see “Loading part files” on page 33.

Selecting all part files

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

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

Deselecting all part files

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

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

Importing other types of files

There are other types of files you can import into Visual Builder to use in your application development. This section describes the following types of files you can import:

- Part information files
- Version 1 components
- Copy files

Part information files (.VCE)

Use part information files to create an interface between pre-existing COBOL code or classes, and your visual parts. For more information on creating part information files and the proper syntax, refer to Appendix A, “Creating part information files” on page 223.

Once you create the file, refer to the steps in “Importing part information” on page 108 to import the part information file and create a nonvisual part. Then use this nonvisual part as you would any other nonvisual part.

Version 1 components

If you have applications you created using the GUI Designer tool in VisualAge for COBOL Version 1.0, you can import the .ODX file to create the appropriate .VCB file for that component. Once imported, you can use the newly created .VCB file as a visual or nonvisual part in Visual Builder.

To import a Version 1.0 component:

1. Start the Visual Builder.
2. Select the **File→Import V1 Component** menu item. The **Import Version 1 Program** window appears, as shown in Figure 13.

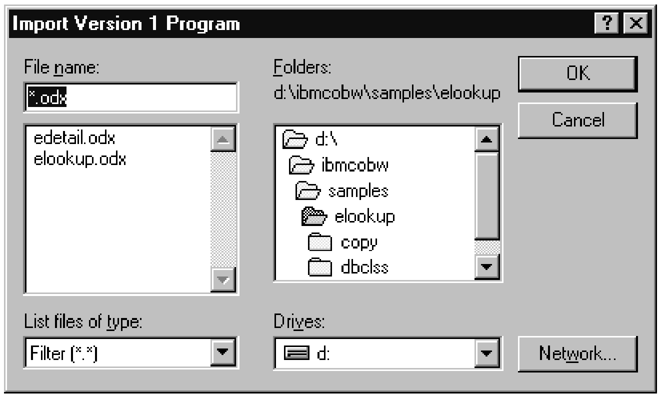


Figure 13. Import Version 1 Program window

3. Select the appropriate directory and file name for the .ODX file you want to import. Then select the **OK** push button.
4. The **New Part Name** window appears. Use this window to enter a new part name other than the one selected by Visual Builder. Select the **OK** push button to import the file.
5. If the import is successful, the appropriate visual or nonvisual parts are created and loaded into the Visual Builder. If the import is unsuccessful, an **Import from ...** window appears listing the errors. Make appropriate changes and repeat the process until you can successfully import the .ODX file.

Once the component is successfully imported, you can use the visual and nonvisual parts created like any visual and nonvisual parts you construct in the Visual Builder.

Copy files

Import copy files to use the data items specified in the copy files in your Visual Builder application. For more information on importing copy files, refer to “Importing copy files” on page 101.

Customizing the information area

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

Show parent class
Displays the name of the parent class in the information area. For example, if you select CContainerControl when this option is selected, Visual Builder displays the following in the information area to show CTextControl is CContainerControl’s parent class:

```
parent: CTextControl
```

For more information on the heirarchy of classes, refer to “The part interface architecture” on page 14.

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

```
COBOL container control
```

Show full file names
Displays the fully qualified name of the part file in which the part is stored.

Seeing the base files

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

Seeing where part files are located

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

Seeing the type list

The type list shows the data types for the parts contained in the part file that is selected. If no part file is selected or if the selected part file has no types defined, this list is empty. You can add to this list by importing a part information file (.VCE) and specifying files that contain type definitions in the VBPartDataFiles statement. Refer to Appendix A, “Creating part information files” on page 223 for additional information on part information files.

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

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

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

To display the type list, do the following:

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

   A list box titled **Loaded Type Information** is displayed at the bottom of the Visual Builder window, as shown in Figure 14 on page 39.

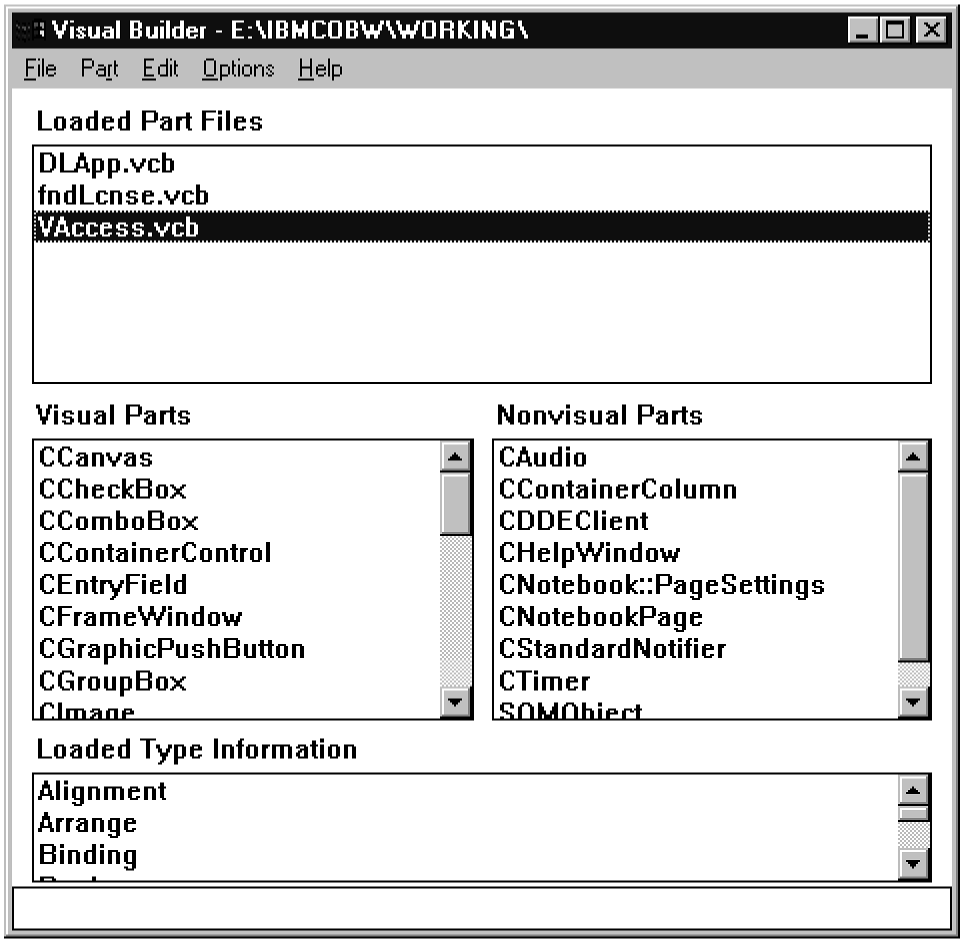


Figure 14. Visual Builder window with loaded type information displayed

Using File Allocation Table (FAT) file names

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

When you select this option, Visual Builder uses these limits to create file names and extensions. For instance, the System Interface Editor uses the file names provided in the **Code generation files** section to store generated code.

If you provide a part name longer than eight character, Visual Builder reduces the name to eight characters. For example, suppose you create a part and name it MyNewPart1. This name has nine characters. If you generate code for this part, the default file name

Visual Builder uses for the files it generates have only eight characters, as does the part file in which the part is saved.

Attention: Visual Builder does not check for existing file names when creating default file names. If you always use the default file name Visual Builder creates on a FAT system, Visual Builder may use a file name that already exists. This causes the existing file to be written over. For example, if you create another part named MyNewPart12, Visual Builder uses the same default file name it used for MyNewPart1.

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

Setting the working directory

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

Note: This option does not appear if you start Visual Builder from a COBOL Visual Builder project. The working directory is the directory you specified in the **Source file directory** field when you created the project. This directory is the working directory for COBOL Visual Builder projects.

When you select this option, Visual Builder displays the window shown in Figure 15:

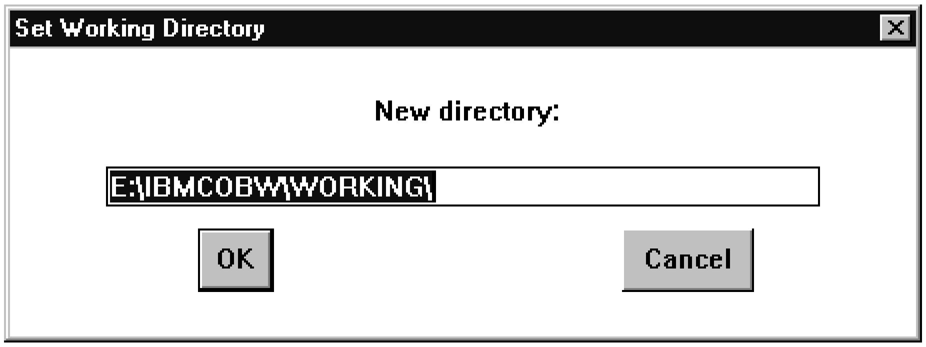


Figure 15. Set Working Directory window

To change the working directory, do the following:

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

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

2. Select the **OK** button.

If the path you enter in the **Set Working Directory** window is not valid, Visual Builder displays an error message and leaves the working directory unchanged.

Refreshing the display

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

Touring Visual Builder

Chapter 4. Getting to know the Visual Builder editors

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

The editor symbols

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

Composition Editor

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

To learn more about the Composition Editor, see "The Composition Editor."

System Interface Editor

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

To learn more about the System Interface Editor, see "The System Interface Editor" on page 51.

Part Interface Editor

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

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

The Composition Editor

Use the Composition Editor to lay out the visual parts that make up your views, choose the parts that perform the logic you need, and make connections between them. This section provides an overview of the Composition Editor's components. See Chapter 8, "Learning to use parts" on page 107 for information about visually composing an application.

The Composition Editor is shown in Figure 16 on page 44.

Touring Visual Builder

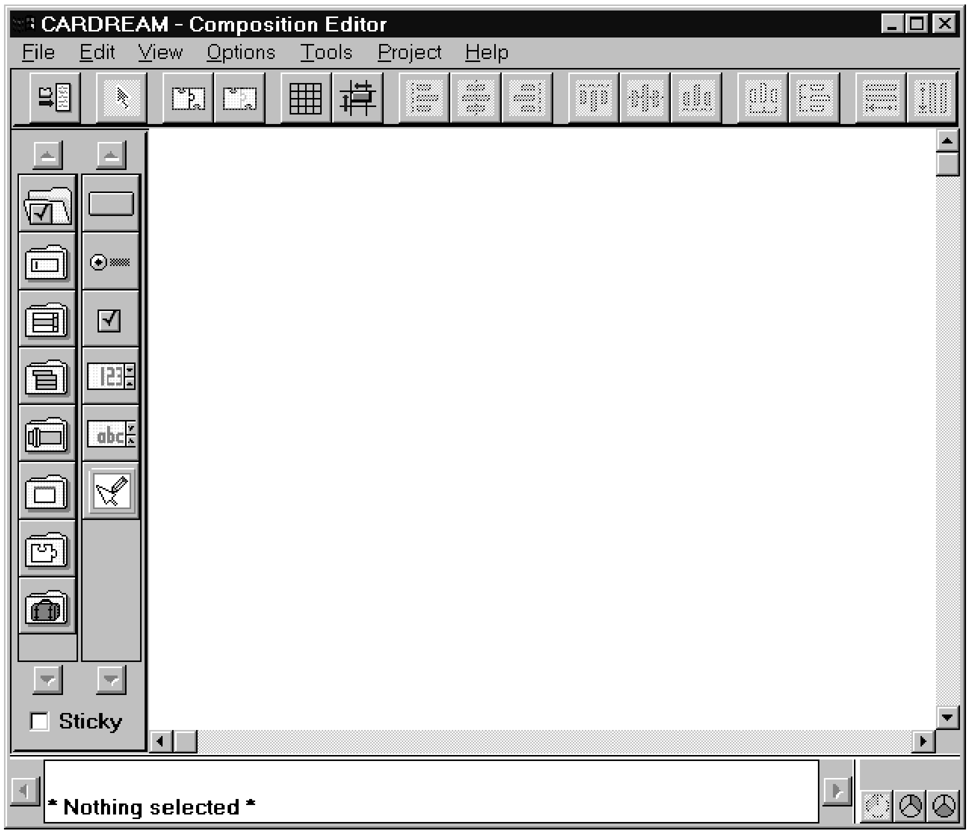

CARDREAM - Composition Editor
File Edit View Options Tools Project Help
Sticky
* Nothing selected *

Figure 16. The Composition Editor

The tool bar

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

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

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

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

The tool bar contains the following tools:

Generate Part Source
Generates COBOL source code for the part that you are currently editing. This tool performs the same function as the **File→Save and generate→Part source** menu choice.

Selection tool
Changes the mouse pointer from the crosshairs, which are used when the mouse pointer is loaded with a part, to the arrow which is used to select parts and perform functions on them. This tool is available only if the mouse pointer is loaded.

Connection tools

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

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

Grid tools

Toggle Grid	Toggles the display of the part alignment grid on and off for the selected parts. You can use separate alignment grids for parts in the Composers category and for the free-form surface.
Snap To Grid	Causes the selected parts to be repositioned to the nearest grid coordinate. The grid does not need to be visible for Snap To Grid to work.

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

Alignment tools

Align Left	Aligns the selected parts to the left edge of the last part selected.
Align Center	Aligns the selected parts along the vertical axis of the last part selected.
Align Right	Aligns the selected parts to the right edge of the last part selected.
Align Top	Aligns the selected parts to the top edge of the last part selected.
Align Middle	Aligns the selected parts along the horizontal axis of the last part selected.
Align Bottom	Aligns the selected parts to the bottom edge of the last part selected.

Distribution tools

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

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

For information about the horizontal and vertical distribution of visual parts within a bounding box, see “Spacing parts within a bounding box” on page 129.

Sizing tools

	Match Width	Sizes the width of the selected parts to match that of the last part selected.
	Match Height	Sizes the height of the selected parts to match that of the last part selected.

The parts palette

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

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

Notes:

- The information area at the bottom of the Composition Editor indicates which category and part are currently selected on the palette or which part or connection is currently selected on the free-form surface.
- You can add categories and parts to the parts palette, as well as delete categories and parts from it. Refer to “Adding categories and parts to the parts palette” on page 143 for more information.

The Visual Builder parts palette contains the following categories and parts.

	Buttons	Contains the following button parts:

- CPushButton
- CRadioButton
- CCheckBox
- CNumericSpinButton
- CTextSpinButton
- CGraphicPushButton

Data entry Contains the following data entry parts:

CStaticText

CNumericStaticText

CEntryField

CNumericEntryField

CMultiLineEdit

CGroupBox

COutlineBox

CImage

Lists Contains the following list parts:

CListBox

CComboBox

CContainerControl

CContainerColumn

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

CMenu

CMenuItem

CMenuCascade

CMenuSeparator

Sliders

Consists of the following slider part:

CSlider

Composers

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

CNotebook

CCanvas

CFrameWindow

Models

Consists of the following parts:

Factory

[ ] Variable

Other

Contains the following miscellaneous parts:

CHelpWindow

CMessageBox

CTimer

CAudio

CMediaPanel

CDDEClient

The free-form surface

The large open area in the Composition Editor (see Figure 16 on page 44) is called the *free-form surface*. This is the working area for visual programming, where you compose the various visual parts of your application and where you make connections to the logic of your application.

You add visual parts, such as static text and push buttons, to either a frame window part or to another part from the Composers category. Only parts from the Composers, Models and Frame Extensions categories can be added to the free-form surface. You add nonvisual parts (such as class interface parts) to your application by placing them on the free-form surface, not on a frame window part or on any other part from the Composers category. Figure 17 summarizes the which categories contains parts allowed on the free-form surface.

Figure 17. Placement of parts on the free-form surface

Category	**Allowed on free-form surface?**
Buttons	No
Data Entry	No
Lists	No
Frame Extensions	Yes
Sliders	No
Composers	Yes
Models	Yes
Other	Yes

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

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

The parts are deleted from the free-form surface.

For more information about using the free-form surface, see “Working with parts on the free-form surface” on page 118.

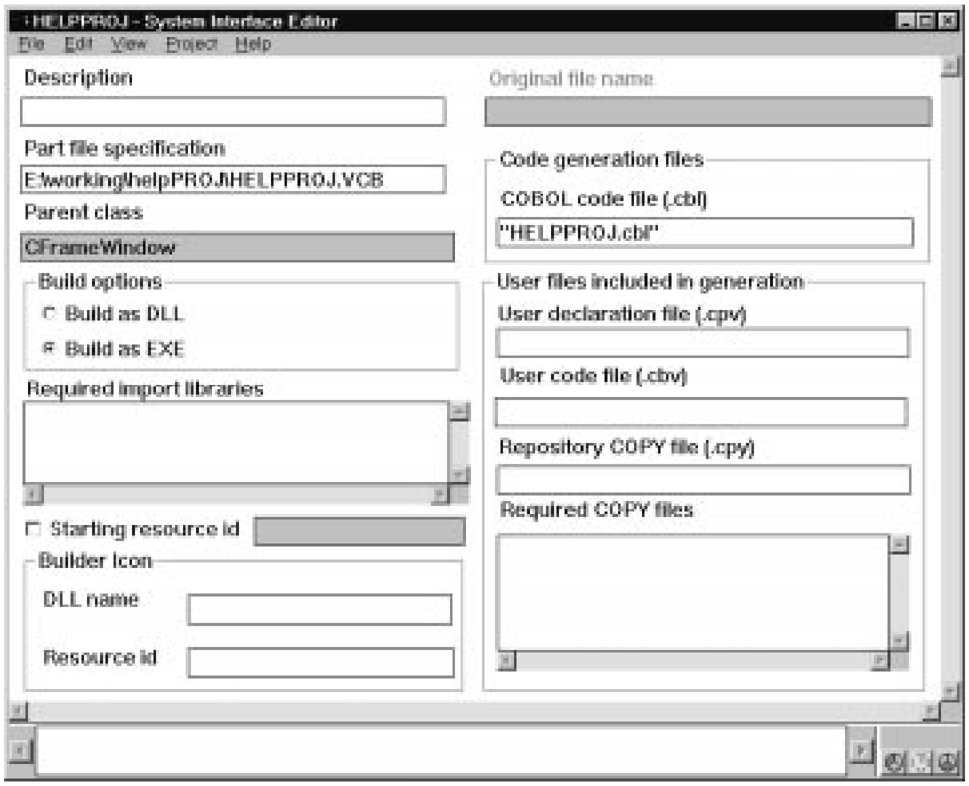
The System Interface Editor

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

- Enter a description of the part
- Specify a different part file in which to store the part
- See the name of the part’s parent class
- Specify the target to create
- Specify the import libraries
- Specify the starting resource ID for the part
- Assign an icon resource to the part
- See the original file name
- Specify the name of the part source file
- Specify additional code files to generate and include in the build

Use your favorite text editor to create new classes and methods, write application logic, and modify existing methods. Then use the System Interface Editor to create the interface for the part you are currently editing.

The System Interface Editor is shown in Figure 18 on page 52.


HELPPROJ - System Interface Editor
File Edit View Project Help

Description

Original file name

Part file specification: E:\working\helpPROJ\HELPPROJ.VCB

Parent class: CFrameWindow

Build options
- Build as DLL
- Build as EXE [✓]

Required import libraries

Starting resource id

Builder Icon
- DLL name
- Resource id

Code generation files
- COBOL code file (.cbl): "HELPPROJ.cbl"

User files included in generation
- User declaration file (.cpv)
- User code file (.cbv)
- Repository COPY file (.cpy)
- Required COPY files

Figure 18. The System Interface Editor

If you cannot see all of the fields shown in Figure 18, use the scroll bar on the right side of the System Interface Editor to see the remaining fields.

Note: When you are editing a *data part*, some of the fields are unavailable.

Entering a description of a part

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

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

Specifying a different part file

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

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

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

Note: If the part file you are copying to exists, the part file must be loaded into Visual Builder before you save the change.

Seeing the parent class of a part

The **Parent class** field in the System Interface Editor shows the name of the parent class for your part. This is the class name you specified as the parent class when you created the part.

You cannot modify the parent class name.

Specifying the target to create

Use the **Build Options** group to specify whether you want the part to be built as a dynamic link library (.DLL) or an executable (.EXE).

If you choose to build a DLL, an import library file is created. Any other part using this part as a subpart links to the DLL of this part using the import library file. Any subparts in this part are linked into this DLL unless they are already linked into another imported DLL.

If you choose to build an executable, all subparts in this part not included in another imported DLL are linked into this executable.

Specifying the import libraries to include

Use the **Required import libraries** field to specify the import libraries required during the build process of your part.

Specifying a starting resource ID

Beginning with the main part in the main program (.APP), Visual Builder generates resource IDs in sequence starting with 5000. You can change this starting number. The check box next to the field enables the starting resource ID entry field. The first time you select the check box, Visual Builder enables the field and inserts the default starting resource ID (5000).

Note: Resource IDs are not generated for non-visual parts.

Specifying a unique icon for your part

Fill in the fields in the **Builder Icon** group in the System Interface Editor before you add your part to the parts palette so you can use an icon other than the default icon provided by Visual Builder to represent your part. The default icon is .

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

DLL name
The resource dynamic link library (.DLL) containing the icon you want to use. Enter the file name only, not the extension.

Resource id
The resource ID number of the icon in the resource dynamic link library (.DLL) whose name you entered in the **DLL name** field.

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

Note: If you do not specify a dynamic link library (.DLL), Visual Builder uses the default icon. If you specify a dynamic link library (.DLL) but Visual Builder cannot find it, Visual Builder uses the question mark icon, .

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

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

Seeing the original file name of imported parts

Use the **Original file name** field to see the file name of the source of an imported part. For instance, if you create a nonvisual part by importing a copy file, the **Original file name** field contains the path and file name of the copy file.

You can not modify this field.

Specifying the names of your part source files

The **Code generation files** group box in the System Interface Editor contains the **COBOL code file (.CBL)** field.

The file name displayed in this field is the file into which your COBOL source code is written. This occurs when you generate code from the Visual Builder window or from any of the editors by selecting **File→Save and generate→Part source**.

The field initially contains the COBOL source code (.CBL) file name based on the name of the part you are editing. To change the file name in this field, enter a new name in the field and select **File→Save** so Visual Builder uses the new file name and writes this file into the working directory. If the file already exists, Visual Builder replaces its contents with the code currently being generated.

Note: If you import an existing COBOL source code file as a nonvisual part and specify the name of the existing COBOL source code file in the **COBOL code file (.CBL)** field, do not generate part source for that nonvisual part. If you do, Visual Builder generates its own COBOL source code file, which overwrites your file.

Specifying additional code files

The **User files included in generation** group in the System Interface Editor allows you to specify files you want to generate when you generate part source. These files are including during the build process. The .CPV and .CBV file extensions in the **User declaration file** and **User code file** fields are used because these files are not meant to be compiled by themselves.

The group contains the following fields:

User declaration file (.CPV)
The copy (.CPY) file you want to include in the working-storage section of the class Visual Builder generates. You must enter a file name in this field before you can generate feature code. Attribute declarations are appended to this file.

User code file (.CBV)
The file of methods you want to include in the part source Visual Builder generates. You must enter a file name in this field before you can generate feature code. Feature methods are appended to this file.

Repository COPY file
The copy (.CPY) file you want to include in the repository section of the class. In this file include any classes not defined in the part but referenced in your feature code.

Required COPY files
The names of other files you want Visual Builder to include in the working storage section of the class when you generate part source for your application.

Visual Builder generates the `COPY` statements in the part source code.

The Part Interface Editor

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

The parts that you create must also have a defined part interface so the part can be used by other parts. Use the Part Interface Editor to define the interface.

Other uses for the Part Interface Editor include the following:

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

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

- The **Attribute** page
- The **Event** page
- The **Action** page
- The **Promote** page
- The **Preferred** page

The **Attribute**, **Event**, and **Action** pages all contain the same set of push buttons along the bottom of the page:

Add with defaults

If you select the **Add with defaults** button, Visual Builder assumes you want to use the default values it provides for the new attribute, action, or event. To view the default values, select the **Defaults** button. After you view the default values, the select the **Clear** button to clear those values.

Suppose you create a part called Employee. You design the part to contain an attribute called *Age*. Visual Builder uses Integer as the default type for an attribute. In the **Attribute** page, enter *Age* in the **Attribute name** field and select the **Add with defaults** button. Visual Builder adds the *Age* attribute with the following settings:

Attribute type

```
Integer
```

Get method

```
getAge RETURNING Age
```

Set method

```
setAge USING Age
```

Event identification

```
AgeId
```

Add
Select the **Add** button to add the attribute, event, or action with the current settings. Selecting the **Defaults** button, then the **Add** button produces the same results as selecting the **Add with defaults** button.

Update
Select the **Update** button to update the attribute, action, or event with the new settings you provide.

Delete
Select the **Delete** button to delete the currently selected attribute, event, or action.

Defaults
If you select the **Defaults** button, Visual Builder supplies default values for the new attribute, action, or event. Select the **Clear** button to clear those values. Select the **Add** button to use those values.

Clear
Select the **Clear** button to clear out the settings for the currently selected attribute, action, or event. This does not delete the currently selected attribute, action, or event.

The Attribute page

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

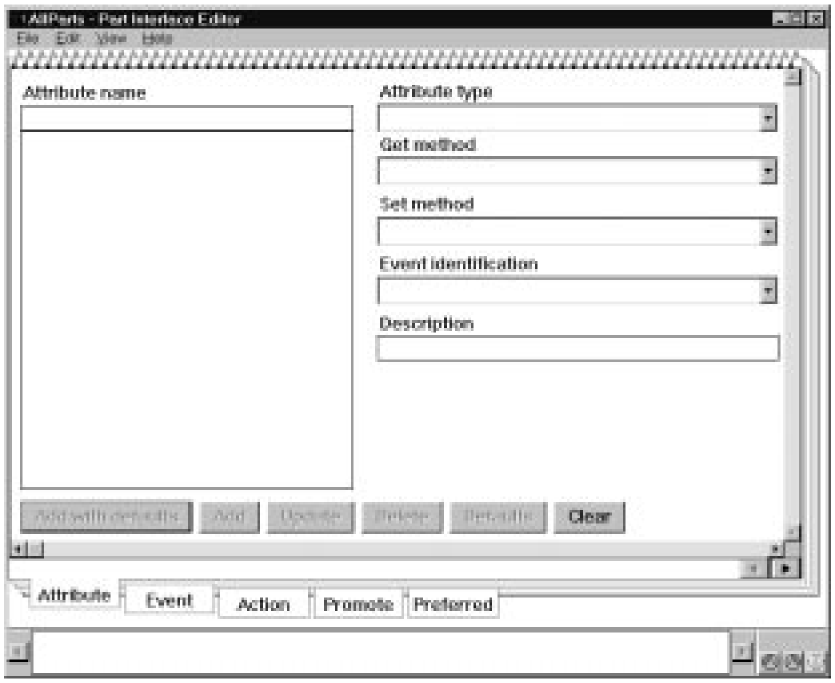


Figure 19. The Attribute page of the Part Interface Editor

You can define many attributes for a part, each with unique characteristics, such as **attribute name** and **attribute type**. An attribute's value is always acquired by using the attribute's get method. You can store data for the attribute in an instance variable of the part class, calculate it, or acquire it from some other location. The default feature code generated defines an instance variable used by the get and set methods to store the data.

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

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

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

no-set attribute
Has no set method. It can be used to initialize another attribute when it is the source of an attribute-to-attribute connection, but it cannot be set to the value of another attribute.

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

Adding an attribute

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

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

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

  Attribute type
  Visual Builder uses types to generate declarations for data items in USING and RETURNING clauses of the get and set methods for an attribute. The first time you create an attribute for your part, the default data type that Visual Builder uses is Integer. If you change Integer to another data type, such as VarLengthString, the new data type becomes the default data type for any new attributes you create until you change it to something else or close the Part Interface Editor. When you reopen the Part Interface Editor, the default data type

for any new attributes you create reverts to Integer. It stays that way unless you change it again when creating another attribute or when modifying an existing attribute.

You can define your own types by importing part information files (.VCE). For example, if you define a monetary type in your part information file with the following declaration:

```
01 monetary PIC $(8)9.99.
```

you can import the part information file and use the attribute of the resulting nonvisual part and connect it to the *asNumeric* attribute of the CStaticText part. When the application runs, the current value of the attribute is displayed using the picture clause. Refer to Appendix A, "Creating part information files" on page 223 for more information on creating part information files.

Typically, the attributes you define correspond to instances variables of the part class. The get and set methods you define for an attribute give other programs access to the attribute's value.

Get method
The method used by the part and other parts to query or *get* the value of the attribute.

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

Event identification
The name of the instance variable in a connection class used to monitor events from this part. You can create one of your own or you can use the default event identifier Visual Builder supplies when you select either the **Defaults** or the **Add with defaults** button. Visual Builder automatically generates a get method used to obtain the event identifier. The name of the get method is the name of the event identifier prefaced with `get`.

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

– Attribute-to-attribute
– Attribute-to-action

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

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

Description
A description of the attribute. This entry field is blank unless you enter a description.

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

Changing an attribute

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

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

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

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

Deleting an attribute

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

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

Setting defaults for an attribute

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

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

Clearing the attribute page fields

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

Visual Builder clears all of the fields on the **Attribute** page. This does not delete the attribute from the part interface.

An attribute example

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

- An attribute type of VarLengthString
- An getAge get method
- A setAge set method
- An ageId event identifier

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

The feature source code Visual Builder generates for the *age* attribute is similar to the following code segment:

Note: You must enter the names of the .CPV and .CBV files in the **User .CPV file** and **User .CBV file** fields, respectively, in the System Interface Editor before generating the feature source code.

- The declarations of the data member in the user header (.CPV) file.

```
* Feature source code generation begins here...
 01 iAge.
    03 iAge-Length      PIC 9(9) COMP-5.
    03 iAge-String.
       05 iAge-Chars  PIC X
            OCCURS 1 TO 255 TIMES
            DEPENDING ON iAge-Length.

* Feature source code generation ends here.
```

- The get and set methods for the *age* attribute, as defined in the feature source code (.CBV) file.

```
* Feature source code generation begins here...
* METHOD
 IDENTIFICATION DIVISION.
 METHOD-ID. "getAge".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 LINKAGE SECTION.
 01 Age.
    03 Age-Length      PIC 9(9) COMP-5.
    03 Age-String.
       05 Age-Chars  PIC X
            OCCURS 1 TO 255 TIMES
            DEPENDING ON Age-Length.
 01 rc PIC S9(9) USAGE COMP-5.
 PROCEDURE DIVISION
     RETURNING Age.
     MOVE iAge-Length TO Age-length.
     MOVE iAge-String TO Age-string.
     GOBACK.

 END METHOD "getAge".

* METHOD
 IDENTIFICATION DIVISION.
 METHOD-ID. "setAge".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01 aCustomer USAGE OBJECT REFERENCE Customer.
 01 event USAGE OBJECT REFERENCE CNotificationEvent.
 01 ageId USAGE POINTER.
 LINKAGE SECTION.
 01 Age.
    03 Age-Length      PIC 9(9) COMP-5.
    03 Age-String.
       05 Age-Chars  PIC X
            OCCURS 1 TO 255 TIMES
            DEPENDING ON Age-Length.
 01 rc PIC S9(9) USAGE COMP-5.
 PROCEDURE DIVISION USING Age.
     MOVE Age-Length TO iAge-length.
     MOVE Age-String TO iAge-string.
     INVOKE CNotificationEvent "somNew" returning event.
     SET aCustomer TO SELF.
     INVOKE SELF "getAgeId"
        RETURNING ageId.
     INVOKE event "initializeNotificationEvent" USING
        by value ageId by value aCustomer.
     INVOKE SELF "notifyObservers" using by value event.
     INVOKE event "somFree".
     GOBACK.

 END METHOD "setAge".
```

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

The Event page

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

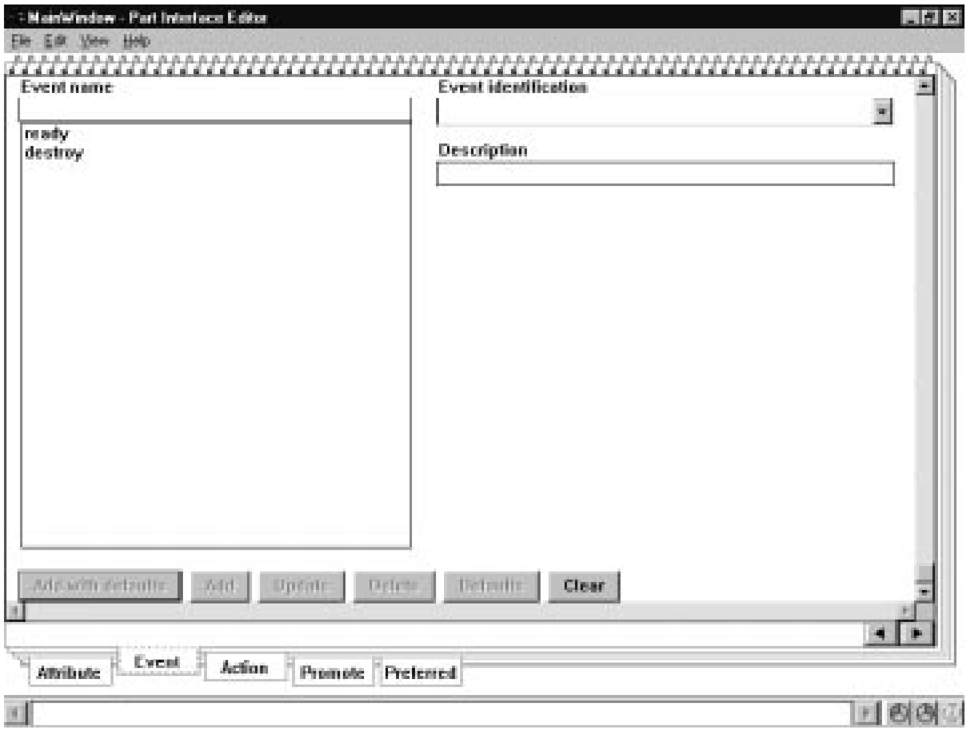


Figure 20. The Event Page of the Part Interface Editor

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

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

The *ready* event

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

By connecting the *ready* event to a subpart of your part, you can cause an action or method to be invoked when your application is executed. The *ready* event after both of the following occur:

- All subparts have been constructed.
- All connections have been made and initialized.

For example, suppose you have a part named MyPart and you want an audio file played every time MyPart is instantiated. By connecting the MyPart's *ready* event to the audio part's *setAudioMode* action and setting the connection parameter to MediaMode-Play, the audio file plays as soon as the part is instantiated.

The *ready* event is not a preferred feature by default, but you can add it to your part's preferred features list. For information on how to do this, see "The Preferred page" on page 72.

Note: The behavior of the *ready* event is different from the *create* event in version 1 of VisualAge for COBOL. After importing a version 1 application, you might need to adjust the connections to the *ready* event. Only windows opened immediately are completely initialized.

Adding an event

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

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

  Visual Builder adds the event to the part interface.

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

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

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

  Event identification
  The name of the instance variable used in connections from this event. When prefaced with *get*, this defines the method used to obtain the event identifier.

  Description
  A description of the event. This entry field is blank unless you enter a description.

Changing an event

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

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

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

Deleting an event

To delete an event, do the following:

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

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

Setting defaults for an event

To set defaults for an event, do the following:

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

Clearing the Event page fields

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

Visual Builder clears all of the fields on the **Event** page. This does not delete an event.

An event example

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

- invalidDataEntered event name
- invalidDataEnteredId event identifier

You create this event because you want to show an error message for the Customer part whenever invalid data is entered for any of the Customer part's attributes, such as the customer's age being outside a valid range. Then, in your feature source code for the *age* attribute's set method, you can call the `notifyObservers` method to display a message asking for a valid age if the check fails.

The default feature code Visual Builder generates for the *invalidDataEntered* event is similar to the following code segment:

- The get method in the COBOL source code (.CBL) file.

```
*------------------------------------------------------------------------
* getinvalidDataEnteredID method
*------------------------------------------------------------------------
 IDENTIFICATION DIVISION.
 METHOD-ID. getinvalidDataEnteredId.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 initFlag PIC 9 VALUE 1.
 01 invalidDataEnteredId USAGE POINTER.
 01 invalidDataEnteredIdString PIC X(35).
 LINKAGE SECTION.
 01 aInvalidDataEnteredId USAGE POINTER.
 PROCEDURE DIVISION RETURNING aInvalidDataEnteredId.
     IF initFlag = 1 THEN
        MOVE 0 TO initFlag
        MOVE Z"nvp402::invalidDataEnteredId"
           TO invalidDataEnteredIdString
        SET invalidDataEnteredId
           TO ADDRESS OF invalidDataEnteredIdString
     END-IF
     SET aInvalidDataEnteredId TO invalidDataEnteredId.
     GOBACK.
 END METHOD getinvalidDataEnteredId.
```

The Action page

Use the **Action** page of the Part Interface Editor, shown in the following figure, to define the *actions* your part uses to perform specific tasks.

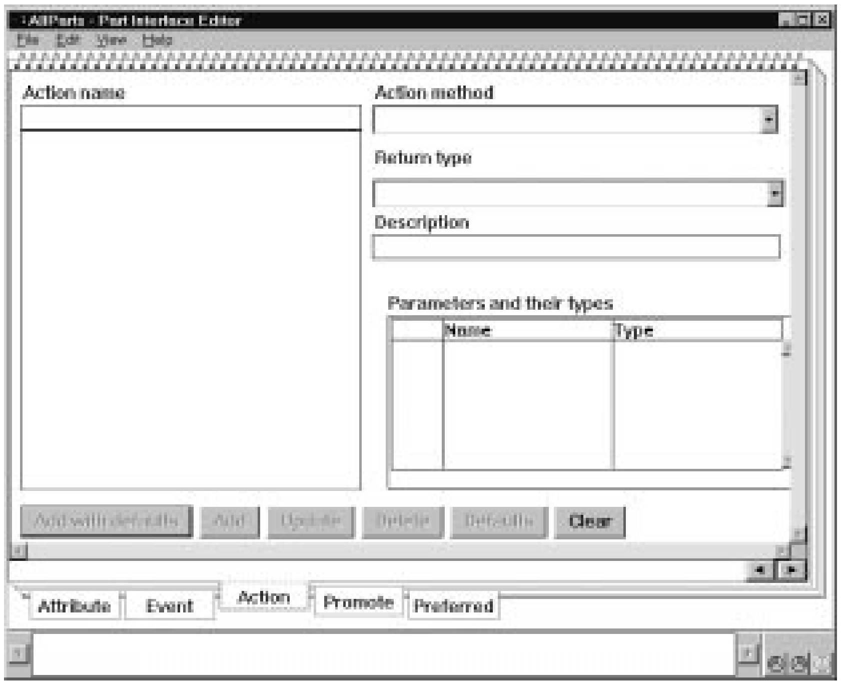


Figure 21. The Action page of the Part Interface Editor

Often, you will want to perform some task when a specific event occurs. For example, you might want to update a *balance* attribute each time a push button's *press* event is triggered. You might create an action named *updateBalance* to perform this task and connect it to the push button's *press* event.

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

Adding an action

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

- If you want to add the action without parameters and without a return parameter, enter a name in the **Action name** field and select the **Add with defaults** button. Visual Builder adds the action to the part interface. The action name is used as the action method name in the **Action method** field.
- Here are the descriptions of additional fields on the page:

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

  Action method
  The name of the method, defined within your reusable part, that performs the action, together with its USING and RETURNING parameters (if any).

Return type
The return type of the RETURNING parameter. The default is Integer. If there is a RETURNING parameter, the *actionResult* feature is available. You can connect the *actionResult* feature to an attribute to store its value, or you can connect the *actionResult* feature as a parameter to another method.

The default feature code generated for an action does nothing. You can, and in most cases should, edit the default feature code to specify what you want returned. The following list shows what is generated:

```
IDENTIFICATION DIVISION.
METHOD-ID. "actionName"
DATA DIVISION.
LOCAL-STORAGE SECTION.
LINKAGE SECTION.
PROCEDURE DIVISION.
  GOBACK.
END METHOD "actionName"
```

Description
A description of the action.

Parameter names and types
Parameters of the selected action. The names in the **Parameter** table are the same as the parameter names that you specify in the **Action method** field.

The parameter names in the table are linked to the action name and can appear in a pop-up menu. The purpose of the table is to allow you to specify their types.

Adding a parameter and its type

To add a parameter and its type you must first add it to the **Action method** field. When you update the page, the parameter table is updated and then you update the types.

Changing parameter types

To change the parameter types in the **Parameters and their types** table, do the following:

1. Click on the parameter type with mouse button 1.
2. Type the parameter type you want to use.

   Note: To exit the edit mode, press Shift+Enter.
3. Select the **Update** button.

Changing an action

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

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

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

Deleting an action

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

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

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

Setting defaults for an action

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

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

Clearing the action page fields

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

Visual Builder clears all of the fields on the **Action** page. This does not delete an action.

The Promote page

Use the **Promote** page of the Part Interface Editor to specify features you want to connect to another part when this part is embedded as a subpart within another part. The features (attributes, events, and actions) you specify appear in the window displayed when you select **More** from this part's pop-up menu. For a complete description of promoting a part's features, see “Promoting a part's features” on page 139.

Note: The **Promote** page only supports visual parts. You can not promote features in a nonvisual part.

Figure 22 on page 70 shows the **Promote** page:

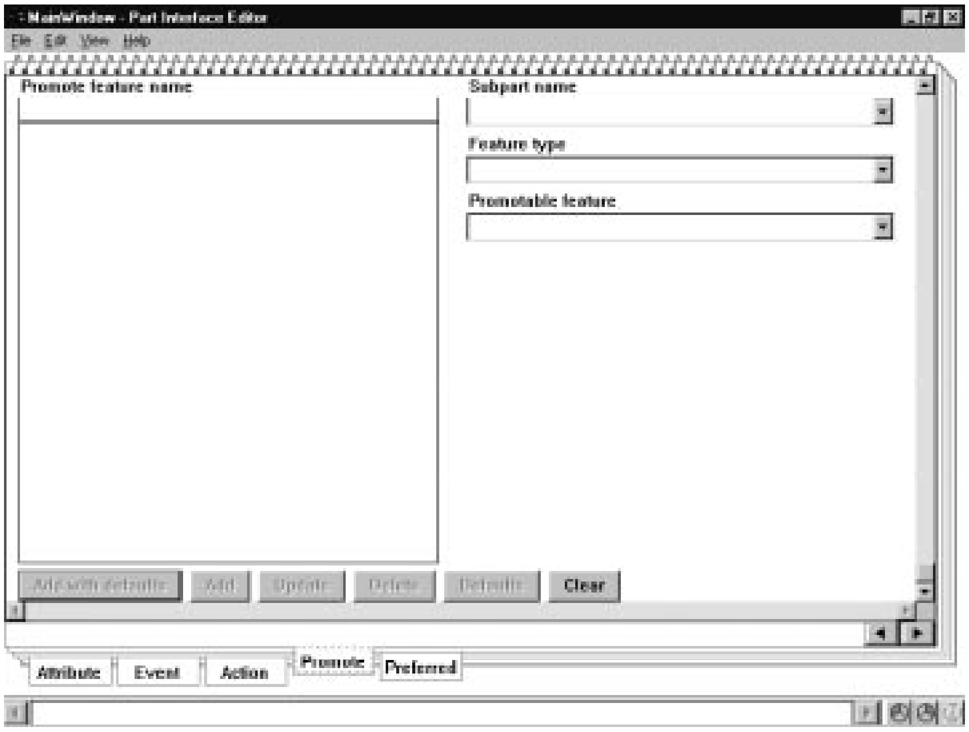


Figure 22. The Promote Page of the Part Interface Editor

Promoting a feature

To promote a feature of a part nested in your part, in the Part Interface Editor do the following:

1. Type the feature name in the **Promote feature name** field.
2. Select a subpart name from the list box beneath the **Subpart name** field or type the subpart name in the field. A subpart is a part nested in the part you are editing. Visual Builder displays the name of the subpart you select in the **Subpart name** field.
3. Select a feature type from the list box beneath the **Feature type** field or type the feature type in the field. Visual Builder displays the type you select (attribute, event, or action) in the **Feature type** field.
4. Select the feature you want to promote from the list box beneath the **Promotable feature** field or type the name in the field. Visual Builder displays the feature you select in the **Promotable feature** field.
5. Do one of the following:
   - If you want to promote the feature using the default name Visual Builder provides, select the **Add with defaults** button. Visual Builder promotes the

feature and displays the feature name in both the **Promote feature name** field and in the list box below this field.

- If you want to see the default feature name Visual Builder provides before you promote the feature, select the **Defaults** button. Visual Builder displays the default name for the feature in the **Promote feature name** field.
- If you want to promote a feature after seeing its default name or typing another name you prefer, select the **Add** button. Visual Builder promotes the feature using the name in the **Promote feature name** field and displays the name in the list box below this field.

Note: In many cases, the default feature name is too long. In such cases, use your own feature name before pressing the **Add** button.

Changing a promoted feature

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

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

The only noticeable change is the subpart name if you selected a different one. The subpart name shown in parentheses behind the promoted feature name changes if you selected a different subpart. For example, suppose you promoted the *press* feature of PushButton1 and then realized you should have promoted the *press* feature of PushButton2. Instead of deleting the promoted feature, you can update it by changing only its subpart.

Deleting a promoted feature

To delete a promoted feature, do the following:

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

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

Clearing the promote page fields

To clear the fields on the **Promote** page, select the **Clear** button. Visual Builder removes the information from all of the fields on the page. This does not delete any features from the part interface.

The Preferred page

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

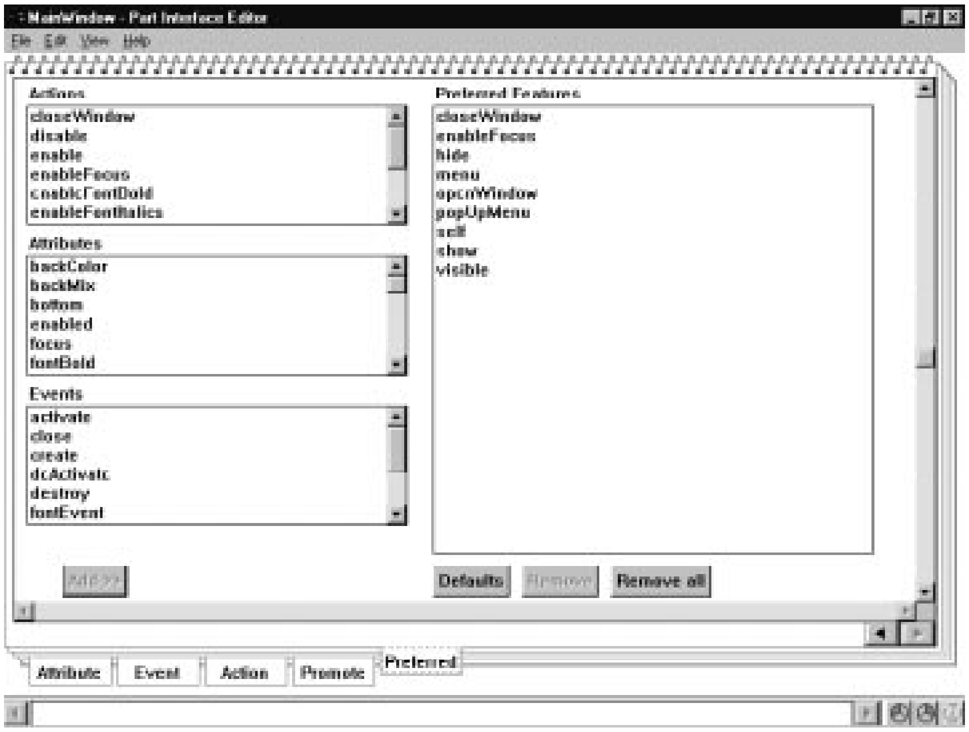


Figure 23. The Preferred Page of the Part Interface Editor

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

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

Adding a preferred feature

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

1. Select a feature name from one of the list boxes on the left side of the page.
2. Do either of the following:
   - Select the **Add** button at the bottom of the page.
   - With the mouse pointer still over the list box in which you selected the feature name, do the following:
     a. Click mouse button 2. A pop-up menu with the **Add** choice appears.
     b. Select **Add** to add the feature.

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

Removing a preferred feature

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

To remove a preferred feature from the pop-up menu for a part, do the following:

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

   A message box is displayed to make sure you want to remove the name of this preferred feature.
3. Select **Yes** to remove the feature name from the list. The feature name you selected is removed from the list.

Removing all preferred features

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

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

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

2. Select **Remove all** to remove all of the selected features.

A message box is displayed to confirm you want to remove all of the preferred features. Select **Yes** to remove all of the feature names from the list. All of the feature names are removed from the list.

Showing inherited preferred features only

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

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

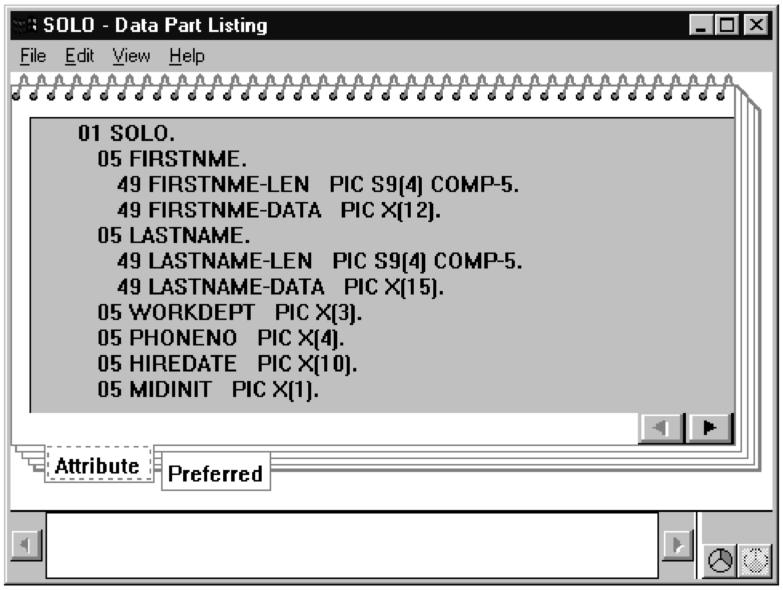
A note on data parts

When you import a copy file using a part information file, a type of nonvisual part called a *data part* is created. Like other nonvisual parts, you can use the System Interface Editor to specify files and resource for the data part and you can use a modified version of the Part Interface Editor called the Data Part Listing to view the attributes of the data part and choose preferred attributes. The Data Part Listing contains the following pages:

- The Attribute page
- The Preferred page

The Attribute page

The **Attribute** page of the Data Part Listing allows you to view the declarations in a data part.



```
01 SOLO.
  05 FIRSTNME.
    49 FIRSTNME-LEN   PIC S9(4) COMP-5.
    49 FIRSTNME-DATA   PIC X(12).
  05 LASTNAME.
    49 LASTNAME-LEN   PIC S9(4) COMP-5.
    49 LASTNAME-DATA   PIC X(15).
  05 WORKDEPT   PIC X(3).
  05 PHONENO   PIC X(4).
  05 HIREDATE   PIC X(10).
  05 MIDINIT   PIC X(1).
```

Figure 24. The Attribute page of the Data Part Listing

Each name displayed is an attribute that can be used to connect this data part to other parts.

The Preferred page

Use the **Preferred** page in the same way you used the **Preferred** page in the Part Interface Editor for any other part. The difference is the **Preferred** page in the Data Part Listing only contains attributes listed in the **Attributes** list box. The **Actions** and **Events** list boxes do not apply.

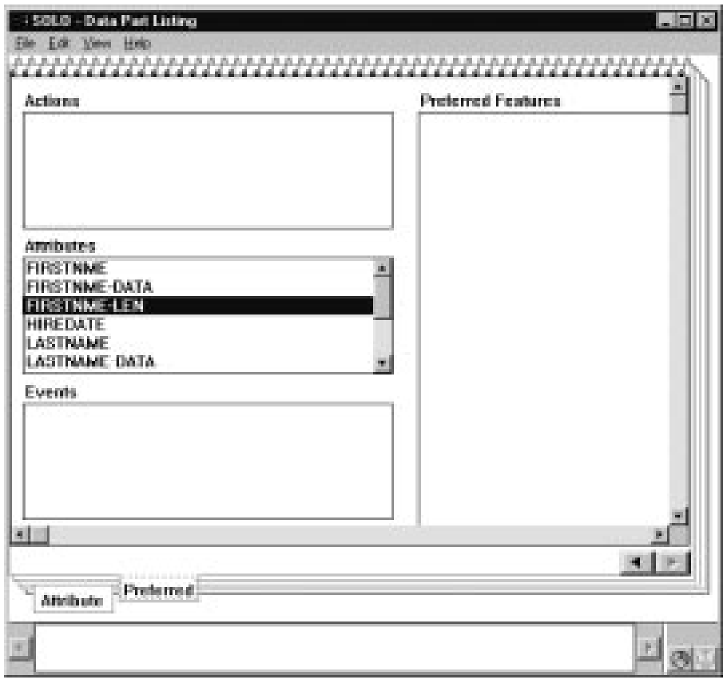


Figure 25. The Preferred page of the Data Part Listing

Touring Visual Builder

Part 3. Developing Visual Builder applications

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

Developing Applications

Chapter 5. Starting Visual Builder

This chapter describes the different ways you can start Visual Builder.

- From a COBOL Visual Builder project
- From VisualAge COBOL
- From a command prompt

Starting Visual Builder from a COBOL Visual Builder project

This section assumes you have already created a COBOL Visual Builder project. Refer to the Information Notebook for instructions on creating projects.

You can start Visual Builder from a COBOL Visual Builder project in the following ways:

In Windows

- To start Visual Builder and also load and open an existing part file (.VCB), open a COBOL Visual Builder project and do either of the following:
  - Double-click on the name of the part file.
  - Select the part file, then select **Selected→Visual** from the project's menu bar.
  - Click on the name of the part file with mouse button 2 and select **Visual** from the pop-up menu.
- To start Visual Builder without loading and opening an existing part file, do either of the following:
  - Click on the white space in the project folder with mouse button 2 and select **Visual** from the pop-up menu.
  - Select **Project** on the menu bar and then select **Visual** in the pull-down menu.

  Visual Builder displays the Visual Builder window, as shown in Figure 26 on page 81. If you double-clicked on a part file to open Visual Builder, the file is loaded and opened.

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

In OS/2

The following steps assume your project inherits the settings of a COBOL Visual Builder project or it was created using the Project Smarts Visual template.

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

   Visual Builder displays the Visual Builder window, as shown in Figure 26 on page 81. If you double-clicked on a part file to open Visual Builder, that file is loaded and opened.

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

Starting Visual Builder from VisualAge COBOL

To start Visual Builder in Windows NT, do the following:

1. Select **Programs** from the **Start** button on the Taskbar. Select **VisualAge COBOL for Windows**. The **VisualAge COBOL for Windows** submenu appears.
2. Select the **Visual Builder** menu item. Visual Builder displays the Visual Builder window, as shown in Figure 26 on page 81. You need to specify a working directory and load part files. Refer to “Getting to know the Visual Builder Window” on page 31 to learn more about specify working directories and loading part files.

   You will also need to run certain tasks like building, debugging, and executing from the command line. Refer to the *Programming Guide* for more information on these operations.

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

1. Select **VisualAge COBOL for Windows** from the **OS/2 Warp** button on the OS/2 Taskbar. The **VisualAge COBOL** submenu appears.

2. Select the **Tools** folder.
3. Select the **COBOL Visual Builder** menu item. Visual Builder displays the Visual Builder window, as shown in Figure 26. You need to specify a working directory and load part files. Refer to “Getting to know the Visual Builder Window” on page 31 to learn more about specify working directories and loading part files.

   You will also need to run certain tasks like building, debugging, and executing from the command line. Refer to the *Programming Guide* for more information on these operations.

Starting Visual Builder from a command prompt

To start Visual Builder from an MS/DOS command window in Windows or from an OS/2 window, do the following:

1. Open an MS/DOS command window in Windows or an OS/2 window.
2. Type the following command:

   ```
   iwzbvb
   ```

3. Press the Enter key. Visual Builder displays the Visual Builder window, as shown in Figure 26.

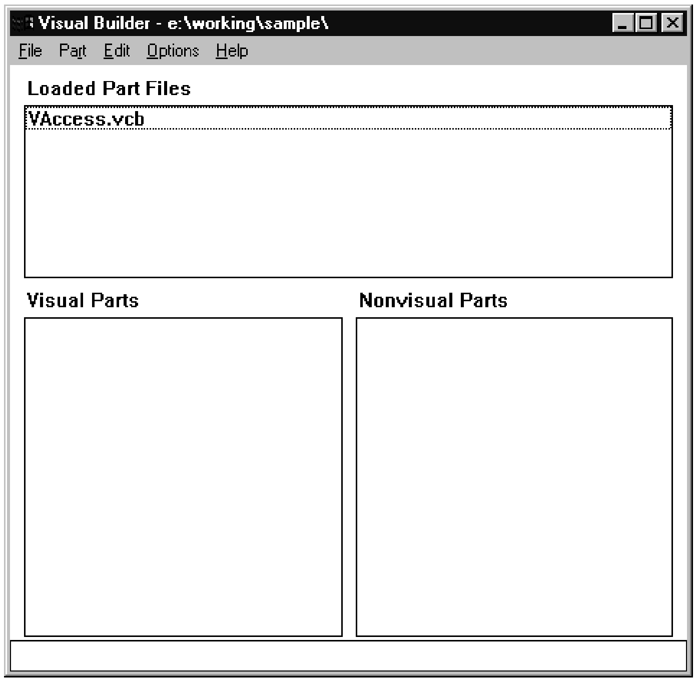


Figure 26. Visual Builder window

Visual Builder assumes the directory you are in when you execute the `iwzbvb` command is the working directory. Part files, part source, and build files generated by

Visual Builder are stored in the working directory. Refer to “Getting to know the Visual Builder Window” on page 31 for more information on working directories and how to change them.

Chapter 6. Creating parts – an overview

This chapter gives an overall description of the steps involved in creating a single part. These steps are similar to the steps involved in any application development cycle: designing, editing, compiling, and debugging. As with any application development project, good design in the beginning can help you avoid problems later on in the cycle. The steps involved in creating a part are:

1. Designing the part
2. Constructing the part
3. Generating the part source and build files
4. Building the part
5. Using or running the part
6. Debugging the part

Applications requiring several different parts from varying sources requiring additional steps, such as setting up subprojects. These steps are beyond the scope of this book. For more information on organizing complex applications, refer to the Information Notebook.

Designing the part

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 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.
- Minimize the number of connections that need to be made when using the Composition Editor.

Before creating a new primitive part, answer the following questions:

- Is the part visual or nonvisual?
- 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?

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.

You can create visual and nonvisual parts using Visual Builder. Visual parts are parts the person using your applications sees. Examples of visual parts are windows, buttons, and entry fields. Later chapters describe in detail how to create composite visual parts with menus, containers, and other primitive visual parts. Nonvisual parts are parts your application uses to perform its functions, but the person using your application does not see them. Nonvisual parts can be categorized into four groups:

- *Nonvisual parts* are nonvisual parts you create with the Visual Builder. You create the interface (features) of this part using the Part Interface Editor and you use feature code to define the feature methods.
- *Class interface parts* are nonvisual parts without event notification ability. Thus, these parts cannot send events to other parts. To create class interface parts, create a part information file (.VCE) file that contains the class interface of COBOL classes you have written. Then import the part information file into Visual Builder. Refer to Appendix A, “Creating part information files” on page 223 for information on creating part information files. Refer to “Importing other types of files” on page 36 for more information on importing part information files.
- *Program parts* are collections of COBOL programs compiled and linked into a single Dynamic Link Library (.DLL). You can define the interface to program parts in a part information file (.VCE) file, then import the part information file into Visual Builder. Refer to “Importing other types of files” on page 36 for more information on importing part information files.
- *Data parts* are parts created by COBOL copy files you import. Once imported, the data items become attributes you can use in connections. Refer to “Importing copy files” on page 101 for more information on importing copy files.

Related Topics

- “Kinds of parts supported in Visual Builder” on page 21
- “Segmentation within the model” on page 7

Naming parts

Because the names of COBOL classes come from a flat name-space, you must ensure that your class names are unlikely to duplicate 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 class names in the global name space begin with the letter “C” for COBOL. In addition, avoid using "i", "a", and "an" as prefixes to your part names. Refer to Chapter 15, “Hints and tips for using Visual Builder” on page 219 for more information on naming restrictions.

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 openWindow, hide, and setFocus.
- Name attributes with phrases that indicate the physical property they represent. Examples of feature names for attributes are height, label, 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 press, close, and menuSelect.

Note: Names are restricted to a length of 23 characters in Visual Builder, compared to the 30 characters normally allowed by COBOL. This allows Visual Builder to suffix or prefix the names under certain circumstances without causing an error.

The main place that users see your action, attribute, and event names is on the Connect 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. In addition, avoid using "i", "a", and "an" as prefixes to your feature names. Refer to Chapter 15, “Hints and tips for using Visual Builder” on page 219 for more information on naming restrictions.

Constructing the part

The following checklist contains 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.

Make the following changes to each class to support parts:

1. To enable notification, make sure your class inherits from the appropriate notifier.
2. Add an initialization (somInit) method, and if required, a somUninit method. Visual Builder expects the initialization method for a part to be called `initialize`.
3. For each event, add a get method that returns the notification ID.
4. For each attribute, add a get method that returns the notification ID.

5. For each attribute, define a get method with no parameters so users can obtain the value of the attribute.
6. If the attribute can be changed, define a set method with a single parameter containing the new value.
7. Define any action methods. Consider reset or default actions for attributes.
8. Code to initialize each event notification ID must be added to the get method for the event ID so that the initialization is performed during the first invocation of the method. The ID should be set to the address of a string in WORKING-STORAGE of the method. The string should be initialized to contain the class name and the event name.

You can also use existing classes without converting them to parts. To do this you need to define a class interface part to access the existing classes.

Implementing attributes

Each property that your part exposes through its attribute interface has one or two corresponding methods to support the attribute interface protocol for accessing the property. The method that retrieves the value of a property is called the *get method*. The method that sets the value of a property is called the *set method*. In addition, you need to define an event ID and an event ID method for notification of changes to the attribute. An example of the definition of a street attribute follows:

```
***********************************************
* street attribute
***********************************************
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 iStreet PIC X(256).

 PROCEDURE DIVISION.

* getStreet method
 IDENTIFICATION DIVISION.
 METHOD-ID. "getStreet".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 LINKAGE SECTION.
 01 Street PIC X(256).
 PROCEDURE DIVISION RETURNING Street.
     MOVE iStreet TO Street.
     GOBACK.
 END METHOD "getStreet".

* setStreet method
 IDENTIFICATION DIVISION.
 METHOD-ID. "setStreet".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01 aStreet USAGE OBJECT REFERENCE Street.
 01 StreetId USAGE POINTER.
 01 event USAGE OBJECT REFERENCE CNotificationEvent.
 LINKAGE SECTION.
 01 Street PIC X(256).
 PROCEDURE DIVISION USING Street.
     MOVE Street TO iStreet.
     INVOKE CNotificationEvent "somNew" RETURNING event.
     SET aStreet TO SELF.
     INVOKE SELF "getStreetId" RETURNING StreetId.
     INVOKE event "initializeNotificationEvent" USING
        BY VALUE StreetId BY VALUE aStreet.
     INVOKE SELF "notifyObservers" USING BY VALUE event.
```

Get and set methods 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 method.

Always use the get and set methods 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.

Get methods

Get methods return the value of a part's property. They are always accessed via an `INVOKE ... RETURNING` statement that does not specify any `USING` parameters.

The simplest get method returns the data member that holds the value of a property. The following `getStreet` method is an example of a simple get method:

```
* getStreet method
 IDENTIFICATION DIVISION.
 METHOD-ID. "getStreet".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 LINKAGE SECTION.
 01 Street PIC X(256).
 PROCEDURE DIVISION RETURNING Street.
     MOVE iStreet TO Street
     GOBACK.
 END METHOD "getStreet".
```

Set methods

Set methods modify the value of a part's property and notify dependent objects that the value has changed. Set methods are always accessed using an `INVOKE` statement with one parameter—the value to be set into the property.

The following `setStreet` method is an example of a simple set method::

```
* setStreet method
 IDENTIFICATION DIVISION.
 METHOD-ID. "setStreet".
 DATA DIVISION.
 LOCAL-STORAGE SECTION.
 01 aStreet USAGE OBJECT REFERENCE Street.
 01 event USAGE OBJECT REFERENCE CNotificationEvent.
 LINKAGE SECTION.
 01 Street PIC X(256).
 PROCEDURE DIVISION USING Street.
     MOVE Street TO iStreet
     INVOKE CNotificationEvent "somNew" RETURNING event
     SET aStreet TO SELF
     INVOKE event "initializeNotificationEvent"
        USING BY VALUE StreetId aStreet
     INVOKE SELF "notifyObservers" USING BY VALUE event
     INVOKE event "somFree"
     GOBACK.
 END METHOD "setStreet".
```

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

Set methods 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 events (notification IDs)

You define notification IDs in COBOL as pointers initialized to an address that uniquely identifies the event. It must be constant for the life of the class, and it must also be constant for the all the object instances of the class. By convention, and as an aid to debugging, the event ID pointer should be initialized as a null-terminated string of the form `className::eventId`. The string addressed by the event ID pointer must be a dat element in WORKING-STORAGE of the method used to get the event ID. The following example shows a get method for a `streetID` event that returns such a pointer:

```
IDENTIFICATION DIVISION.
METHOD-ID. getStreetId.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 initFlag PIC 9 VALUE 1.
01 streetId USAGE POINTER.
01 streetIdString PIC X(28)
    VALUE Z"IAddress::streetId".
LINKAGE SECTION.
01 outStreetId USAGE POINTER.
PROCEDURE DIVISION RETURNING outStreetId.
    IF initFlag = 1 THEN
        MOVE ZERO TO initFLag
        SET streetId TO ADDRESS OF streetIdString.
    END-IF.
    SET outStreetId TO StreetId.
END METHOD getStreetId.
```

Implementing actions

Each behavior that your part exposes in its action interface has a corresponding method to support the action protocol for that behavior. For example the street attribute could have an `initializeStreet` method as follows:

```
IDENTIFICATION DIVISION.
METHOD-ID. "initializeStreet".
DATA DIVISION.
LOCAL-STORAGE SECTION.
01  aStreet PIC X(256).
PROCEDURE DIVISION.
    MOVE SPACES TO aStreet
    INVOKE SELF "setStreet" USING aStreet
    GOBACK.
END METHOD "initializeStreet".
```

Just like other COBOL methods, actions can receive parameters and can also have a return parameter. You can specify the return parameter and any other parameters as part of the action interface.

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

For example, the `initializeStreet` method described above uses the `setStreet` method to set the street attribute instead of setting it directly.

Consider providing an action to reset each attribute to a default value when implementing a part. For Boolean attributes, provide a set method (for example, a disable action) that causes the attribute to be set to `false` and a set method (for example, an enable action) with a default parameter equal to `true`. You can use the set method with a default value to perform the reset action for the Boolean attribute.

Guidelines for implementing nonvisual parts

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

Place your nonvisual part as shown in Figure 27 under the `CStandardNotifier` class hierarchy. Inserted in this location, your part inherits certain default behavior from `CStandardNotifier` and the `CNotifier` protocol.

Figure 27. Class Hierarchy for Nonvisual Parts

Class	**Responsibility**
SOMObject	Base class
CNotifier	Notification protocol
CStandardNotifier	Implementation of notification protocol
New nonvisual part	

An example of a `CLASS-ID` statement for an `address` class definition follows:

```
CLASS-ID. address INHERITS CStandardNotifier.
```

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

Creating a COBOL class for a part is not much different from creating any other COBOL class. You just need to keep in mind the few additional guidelines presented in this chapter for those methods that support your part's interface.

Generating source and build files

Once you construct a part, generate the source and build files for the part. This sections describes the following steps:

- Generating COBOL source code for a part
- Generating feature code
- Generating build files
- Preparing generated files for compilation

Generating COBOL source code a part

You can generate source code for the part being edited from any of the Visual Builder editors.

1. From the editor's menu bar, select **File**.
2. Select **Save and generate**; then select **Part source**.

If you are using the Composition Editor, you can select , the Generate Part Code tool, from the tool bar. There is no difference between selecting this icon and using the menu item described previously.

One of the most common causes of code generation errors is changing the names of features connected to other features. For example, suppose feature A is connected to feature B. If you change the name of feature A and then regenerate the source code for your part, Visual Builder displays an error. This can also occur if you change the name of a promoted feature. To correct the error, double-click on the connection and replace the incorrect feature name with the correct one.

Some capabilities found in OS/2 parts are not available in Windows. If you port your OS/2 part to Windows, there may be code generation problems.

Because Visual Builder cannot discern why the settings or features are not currently valid, it writes a message stating items are no longer valid because the part interface has changed. The following messages may appear:

`The X part was not found.`
: There is a part missing.

`Source: X Target: Y Source event: E Target action: A Missing parameters: Z`
: There is a parameter missing.

`Source: X Target: Y`
: There is a possible data type conflict.

`The X menu is not connected.`
: There is a menu connection missing.

Source files created during part code generation

For each part processed, Visual Builder generates several source code files. As an example, the following files are created for a part called MyPart:

MyPart.cbl	A COBOL code file for part and connection classes.
MyPart.app	COBOL code for main executable.
MyPart.def	Export definitions for creating a dynamic link library (.DLL).
MyPart.odx	Runtime initialization file.
MyPart.rc	Main resource file.
MyPart.rch	Items for help table.
MyPart.rci	Help and accelerator tables; icons.
MyPart.rcs	Items for help subtable.
MyPart.rca	Items for accelerator table.
MyPart.cph	Factory resource computation.

If you selected **Default to FAT file names** under the **Options** pull-down menu of the Visual Builder window and your part name has more than eight characters, Visual Builder creates an eight-character name for the part. Refer to “Using File Allocation Table (FAT) file names” on page 39 for a detailed description.

Generating feature code

If you have defined attributes or actions for your part using the Part Interface Editor, you must provide feature code and you must set the **User declaration file** and the **User code file** fields in the System Interface Editor. The code in these files can be written and maintained by you or you can first generate them and then modify them. Unlike the generation of the part source files, Visual Builder appends code for selected items to these files.

Note: Regenerate feature code carefully. Regenerating feature code already generated causes duplicate definition errors during compilation.

The syntax of the get and set methods changed between version 2.0 and version 2.1 of the Visual Builder. The new syntax is similar to the following:

```
INVOKE InterfaceManager "setHeight" USING object Height
INVOKE InterfaceManager "getHeight" USING object RETURNING Height
```

If you used the get and set methods in your feature code, you must update the syntax of those methods in your feature code.

Generating build files

When you select **File→Save and generate→Build files**, a make file is generated. Using the MyPart example introduced earlier, the following file is created:

MyPart.mak	A make file.

If you selected **Default to FAT file names** as a preference under the **Options** pull-down menu of the Visual Builder window and your part name has more than eight characters, Visual Builder creates an eight-character name for the generated files. Refer to "Using File Allocation Table (FAT) file names" on page 39 for a detailed description.

Before you generate the build file, decide on a target: an executable (.EXE) or a dynamic link library (DLL). Set up the Visual Builder to generate a make file that builds the desired target. For instance, if your target is a dynamic link library (DLL), select **Build as DLL** in the System Interface Editor. If you target is an executable (.EXE) , select **Build as EXE** in the System Interface Editor. Refer to "Specifying the target to create" on page 53 for more information.

Preparing generated files for compilation

Before you compile your part, make sure you have the following files:

- Copy files for all parts
- Source code (.CBL) files for all parts
- Resource files (.RCI) for all visual parts
- A make file
- An .app file for the main part

Most of these files are created when you generate part source and build files in Visual Builder.

Final preparations for compilation and linking include the following:

WorkFrame

Specify options for the compiler and linker by opening the **GUI Compile** options notebook in Windows, or the **Compile** options notebook in OS/2. To open the options notebook, select **GUI Compile** (Windows) or **Compile** (OS/2) from the **Options** menu in WorkFrame.

non-WorkFrame

- Specify additional libraries (COBOL libraries and DLLs) via the environment variables DLLDEPENDS and DLLLIBS
- Specify options for the compiler and linker programs via the environment variables PARTCOMPFLAGS and PARTLINKFLAGS

Enhancing the Visual Builder make file

To add options, object files, libraries, etc., to the Visual Builder make file, set the following environment variables:

PARTCOMPFLAGS
Options used to compile the part class

APPCOMPFLAGS
Options used to compile the main application

PARTLINKFLAGS
Options used by ilink to link the part DLL

APPLINKFLAGS
Options used by ilink to link the main EXE

CVBDEBUG
Enable debugging when set; disable when not set

EXEDEPENDS
Dependent .obj and .lib for linking of main program

EXELIBS
.obj and .lib files to add to linking of main program

DLLDEPENDS
Dependent .obj and .lib for linking of part

DLLLIBS
.obj and .lib to add to linking of part

Specifying debug options for the compiler and linker programs

You can compile and link your part with or without debugging options. If you are using WorkFrame to build your application, select the **GUI Compile** menu item from the **Options** menu in WorkFrame and select the appropriate options to generate debug information.

If you are not using WorkFrame to build your application, specify the CVDEBUG environment variable to generate debug information. You can compile as follows:

```
NMAKE CVBDEBUG=1 MyPart.mak
```

where `MyPart.mak` is the make file for our MyPart example. The CVBDEBUG environment variable determines whether compilation flags are set to enable debugging. A value of 1 indicates debug information is generated. If you do not want to generate debug information, set the environment variable as follows:

```
SET CVDEBUG=
```

Building your part

Compiling and linking a Visual Builder part is called building. You can build your part from a COBOL Visual Builder project or from a command prompt. Before you build your part, you must generate the build files. Refer to “Generating build files” on page 93 for more information.

Building from a COBOL Visual Builder project

You can build your part either in the Visual Builder window or from your COBOL Visual Builder project. Select **Build** from the **Project** menu in you COBOL Visual Builder project or in the Visual Builder window, assuming you started the Visual Builder from COBOL Visual Builder project. You must generate part source and build files before building your part. Refer to “Generating source and build files” on page 91 for more information.

Building from the command line

To build your part from the command line, use the `nmake` command. For example, if you have a part file named MyPart.VCB and you generate the build file for this part, your make file is called MyPart.mak. Then, to run the `nmake` program, you issue the following command from an OS/2 window or MS/DOS command window:

```
nmake mypart.mak
```

After your part is successfully built, you have an executable in your directory which you can run or a dynamic link library you can use as a resource.

Using or running your part

Depending on the target you selected (an executable or a dynamic link library), you can either run your part or use it as a resource for another part or in your application. If your target is an executable, look for the executable (.EXE) file in the working directory.

Using the MyPart example, the executable created is `MyPart.EXE`. To run this part, select **Project→Run** from your COBOL Visual Builder project, or type `MyPart` on the command line in an OS/2 window or MS/DOS command window. If your target is a dynamic link library, look for the (.DLL) file in your COBOL Visual Builder project folder or the working directory. Using the MyPart example, the dynamic link library created is `MyPart.dll`.

Debugging your part

If your part is not performing as you expect, you can build your part with debugging options and use the debugger to step through your code. The following sections list some possible problems and suggestions on how to deal with them.

Developing Applications

Workstation beeps

If your workstation beeps while you are running Visual Builder, an exception has probably been thrown. Check for the following conditions:

- Missing resources (bitmaps or icons)
- Incorrect or incompatible part settings

No connections run in user code

You must explicitly instantiate connections and enable notification for them. Visual Builder does this in generated code by means of the `initializePart` method. This method initializes any static subparts and all connections from the part. If you instantiate a generated part using the `somNew` method in your own code, be sure to call the part's `initializePart` method, as follows:

```
* create screen
INVOKE pMine "initialize" USING ...
* initialize connections
INVOKE pMine "initializePart" USING ...
//
// Now I can do something.
//
INVOKE pMine "show"
```

Note: Make sure you have the correct number of parameters and they are intialized.

Offset calculation for windows

If compiled application windows appear partially off the screen, be aware Visual Builder calculates window position based on the x and y coordinates of the upper-left corner of the window. When you create a part containing the window, Visual Builder calculates a position offset from the upper-left corner of the scrollable free-form surface to the upper-left corner of the window. This does not usually present a problem because most parts are built from the upper-left corner of the free-form surface.

If you have added enough other parts to increase the size of the scrollable free-form surface, the offset might become large enough to push a compiled window subpart off the edge of the desktop display. To prevent this, place the window subpart immediately to the right of the primary window part. Place nonvisual parts at the extreme right of the free-form surface.

Cannot enter text into entry fields on Windows

The default font for Windows is taller than the default font for OS/2. This size difference can cause problems if you drop CEntryField parts on a CCanvas part in OS/2 and then try to use the composite part in Windows.

The default height of an entry field is calculated in OS/2 based on the size of the OS/2 font. Compiled in Windows, this entry field cannot accommodate the taller Windows font.

If you experience this problem, either reset all CEntryField parts in the Visual Builder on Windows to their default size from the part's contextual menu or increase the height of the entry field on OS/2.

Error handling

The CInterfaceManager has an *errorCode* method. After invoking a method in a class, you can invoke the method *getErrorCode* to return an integer. Use this integer to determine if the prior method in CInterfaceManager executed successfully by testing for 0. Use the *setErrorCode* method to reset the *errorCode* to zero.

Application terminates suddenly

If your application terminates suddenly, there are several ways to help you determine the cause of the termination.

Check for runtime error

If you are running your application from the command line, redirect the standard output and error to files, then read the file in a text editor. For instance, if you are running an application called *MyApp*, type the following at the command line:

```
MyApp > My.out 2> My.err
```

Any COBOL run time messages are redirected into the file My.err. System messages and results from your DISPLAY statements are redirected to My.out. You can then use any text editor to view My.err and read any run time messages.

If you are using WorkFrame, run the application monitored and both run time messages and standard output appear in the monitor.

Run application in debug mode

Follow these steps to enable the debugger for your application:

1. Determine which part appears to be the source of the problem.
2. Set the CVBDEBUG environment variable to 1.
3. Rebuild the part that appears to be the source of the problem. This enables the debugger.
4. Run the debugger. The debugger stops at the program in your main .app file.
5. Select run. If the code for this part fails, the debugger encounters an exception and the **Application Exception Action** window appears.
6. Select **Examine/retry** and open the stack monitor.
7. Click on one of your programs or methods to determine a source code line at which the failure occurs.
8. Set breakpoints in the source accordingly.
9. Select **restart** from the **Run** menu, then step through your code.

Use this process repeatedly, if necessary, until you are able to determine the source of the failure.

Note: Many events result in the execution of a *processNotification* method in a connection class. Avoid setting breakpoints at the method-id of these methods. Instead, set breakpoints after the event identifier is tested for the identifier relevant to the connection. If such a breakpoint is not reached during execution, check the event of the source of the connection and confirm it is the event you want. In the Composition Editor, click on the connection to view information about the generated class and code in the information area at the bottom of the window.

Type mismatch
A common cause of sudden application termination is the mismatch of interfaces between the invoking method and the method implementation. In many cases, compile your code using the TYPECHK option before invoking the debugger. See the *Programming Guide* for more information on compiling code with the TYPECHK option.

Tracing execution flow

Trace the execution of your feature code and supporting programs by inserting display statements. If you are running your application from the command line, type in the following:

```
MyApp > app.out
```

The output from the display statements executed are redirected into the file app.out. Use a text editor to view app.out.

If you are using WorkFrame, run the application monitored and the monitor window displays the display statements executed.

Chapter 7. Creating nonvisual parts

This chapter describes how to define your own nonvisual parts. For information about using the Composition Editor to create visual parts, see Chapter 8, “Learning to use parts” on page 107. For more information about using the Part Interface Editor to define part interfaces, see “The Part Interface Editor” on page 56.

Creating a nonvisual part

You create a nonvisual part by doing the following:

1. Design the part.
2. Define the part interface, either through the Part Interface Editor or by importing a part information file. (See “Defining the part interface” on page 103.)
3. Add code to your part. You can use COBOL code written outside of Visual Builder, or you can generate feature code in Visual Builder and modify it. (See “Adding code to your part” on page 104.)

Using existing COBOL code

If you have previously existing COBOL code you would like to use in your part, the following process can be more efficient:

1. Design the nonvisual part.
2. Define the part interface using part information files. Define actions to represent each program.
3. Define the interface to data types representing your data structures using part information files.
4. Import these part information files into Visual Builder.

Using COBOL data declarations

If you have existing COBOL data declarations you want to use in your part, it is more efficient to use the following process:

1. Put the declaration in a copy file.
2. Import this copy file into the Visual Builder to create a data part.
3. Use the data part as you would any other nonvisual part with attributes.

Using existing COBOL code with Visual Builder

This chapter describes how you can use your existing COBOL code in applications that you create with Visual Builder.

Defining the part interface using part information files

If COBOL code already exists for your Visual Builder application, you can more efficiently define the part interface using part information files. This involves the following steps:

1. Determine the part's features.
2. Create a part information file using your favorite editor. This file can include information for as many parts as you need.
3. In Visual Builder, import the part.

Creating a part information file

To create a part information file, add information about your part's interface to a file using your preferred editor. Refer to Appendix A, "Creating part information files" on page 223 for information on creating part information files. Some important syntax items to note are the following:

- The VBBeginPartInfo and VBEndPartInfo statements delimit the part information for the part.
- The VBParent statement specifies the parent class for the part.
- The VBCopy statement specifies a copy file to be added to the repository phrase when the code is generated.
- The VBPartDataFile statement specifies the part file (.VCB) holding the information for the part.
- The VBComposerInfo statement indicates the kind of part.
- The VBEvent, VBAction, and VBAttribute statements define features for this part.
- The VBGeneratorValues to indicate you want to build this part into a dynamic link library (DLL). As an alternative, you can specify **Build as DLL** in the System Interface Editor after you import the parts.

For information about part definition syntax, refer to Appendix A, "Creating part information files" on page 223.

Importing the part

Before importing the part, you must create a part information file. To import the part, follow these steps from the Visual Builder window:

1. From the menu bar, select **File**. Select **Import part information**.

   The **Enter Name for Part Information File** window appears.
2. Specify the path and name of the part information file containing the information you want to import. When the import is finished, the name of the part appears in the Visual Builder window.

If COBOL code for your part already exists, your part is finished. If you want to change the part interface later, do either of the following:

- Use the Part Interface Editor to edit feature specifications.
- Edit your part information file and re-import the part information. You must use this method if you are creating a program part.

If COBOL code for your part does not exist, see “Adding code to your part” on page 104.

Importing copy files

If you have pre-existing data structures defined in copy files and you want to use them in your application, you can import those copy files and use them as data parts. When you import a copy file, the 01 and 77 level data items it contains become nonvisual parts in the Visual Builder, and can be used like any other nonvisual parts.

There are several rules copy files must conform to in order to be imported correctly. These rules are as follows:

1. A copy file must conform to COBOL syntax rules before it is imported. A copy file imported with errors may generate code with errors.
2. A copy file can contain only data description entries (and comments) to be used in the data division.
3. OCCURS DEPENDING ON is not supported. Any 01 level record containing an OCCURS DEPENDING ON is not imported.
4. Nested COPY statements are supported. Each copy file must have complete declarations begining with either an 01 level data item or a 77 level data item. Copy files may contain multiple 01 level or 77 level data descriptions.
5. COPY REPLACING is not supported. A copy file containing Pseudo text may produce undesired results.
6. Level 66 items are ignored.
7. REDEFINEd data items are ignored.
8. Names must be less than 24 characters in length. If you attempt to import a 01 level data item with a name 24 characters or longer, or if it contains a lower level

data item with a name 24 characters or longer, the entire 01 level data item is not imported.

In addition, you must pay careful attention to any types. When you connect two features, they must have the same type. The Visual Builder does not check type mismatches. This could lead to problems in the generated code.

The data item names used in connections are used in the generated code. It may be useful to use your favorite editor to browse the generated code to see how your data items are being used.

Copy files conforming to these rules can be imported as follows:

1. In the Visual Builder window, select **File→Import COPY file**. The **Import COPY file** window appears, as shown in the following figure:

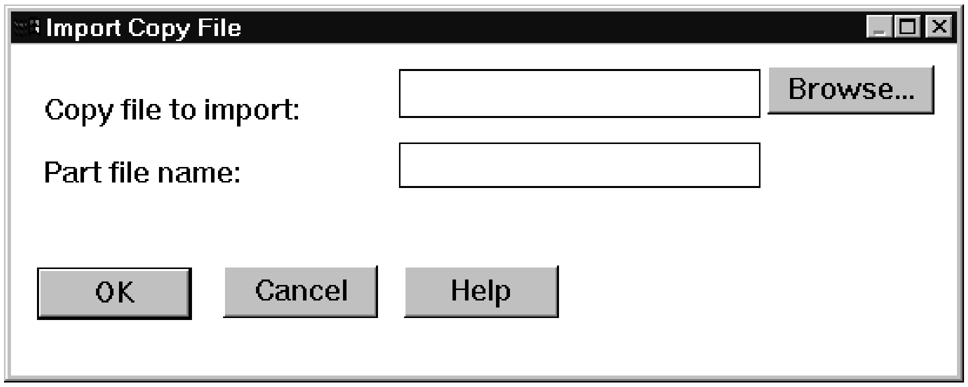


Figure 28. Import copy file window

2. Enter the name of the copy file in the **Copy file** entry field or select the copy file using the **File** button.
3. Once you have located the proper copy file, select the **OK** button. The imported copy file defines a part file (.VCB), which can be used like any other part file.

Once a file is imported, each 01 or 77 level defines a different part. These parts have attributes defined by their data item names. Connecting to and from these attributes amounts to a COBOL MOVE statement which moves data to and from the corresponding data item in the instance of the part. When connecting these attributes, you must be aware of COBOL MOVE rules. Type checking is not done; data may be converted. Use the settings notebook to set the initial value of any attribute.

In the generated code, the data parts contain an expanded, structured, uncommented form of each relevant data description. In the main data part, the declaration is an instance of the entire 01 group or elementary item. In the connections to specific attributes, this is the selected higher level group.

Notes:

1. Copy files can also be used to define types via a part information file (.VCE). In this case, the copy file can **not** have an 01 level data item. Types are used to define parameters of the get or set methods in various parts. The 01 level group item of such a variable is supplied by the code generator followed by the COPY statement.
2. The specific copy file of a component created while importing components created using GUI Designer of VisualAge for COBOL version 1.0 components have a special, reserved 01 level data item. The related variables are treated as a special case.

Defining the part interface

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

1. Define the attributes of your part.

   A part's attributes typically correlate to the class' data members and can additionally include *derived attributes*, or attributes based on the value of other attributes.
2. Define the methods that get or set the value of those attributes.
3. Define any actions you want the part to be able to do.

   A part's actions correlate to the class' methods.
4. Specify the event identifier used to signal a change in the value of each attribute.

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

- Use the Part Interface Editor to create the part interface from the Visual Builder window and then enter each feature of the part interface individually using the pages of the Part Interface Editor.
- Use a part information file to encode part information in a file and then create the part and its interface by importing the information into Visual Builder. This method might be more efficient if COBOL code already exists for your part.

  Note: You must use this method for program parts.

  Refer to Appendix A, "Creating part information files" on page 223 to learn how to create a part information file.

Adding code to your part

Once you specify the part interface for your part, you add the code to make the part work. To add code, complete the following tasks:

1. Generate feature code. If you already have COBOL code for your part and have imported the part information, this step is not necessary.
2. Modify the feature code.
3. Add code created outside Visual Builder, if it already exists.

Generating feature code

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

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

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

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

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

Adding code created outside Visual Builder

To include previously existing methods, change the file extension to .CBV. If the methods require new instance variables, declare these in a .CPV file. If the methods reference additional classes, declare these classes in the copy (.CPY) file. Add these file names to the System Interface Editor for the appropriate part.

Developing code outside Visual Builder

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

1. Write the code.
2. Compile it into a dynamic link library (.DLL).

3. Create an import library and rename it so it has a .imp extension.
4. Define the part interface using a part information file, including a `VBGeneratorValues:genMake('dll')` statement. Use the import library name as the part name.
5. Import the part information file. Refer to Appendix A, “Creating part information files” on page 223 for more information on creating part information files.

When you build the visual part that needs to reference methods or programs in the dynamic link library, add this nonvisual part. After you generate source code and build files and then make your executable, the references to methods or programs in the dynamic link library are resolved.

Chapter 8. Learning to use parts

This chapter provides general information on using parts.

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

With connections, you can construct the interactions between the parts of the application. For information about connecting parts to each other, see Chapter 9, “Learning to use connections” on page 153.

In Visual Builder you use the following kinds of parts:

- Visual parts, such as CPushButton, CListBox, and CEntryField, to construct the graphical user interface (GUI) of the application.
- Nonvisual parts to represent the data or objects manipulated by the application.

Working with parts in the Visual Builder Window

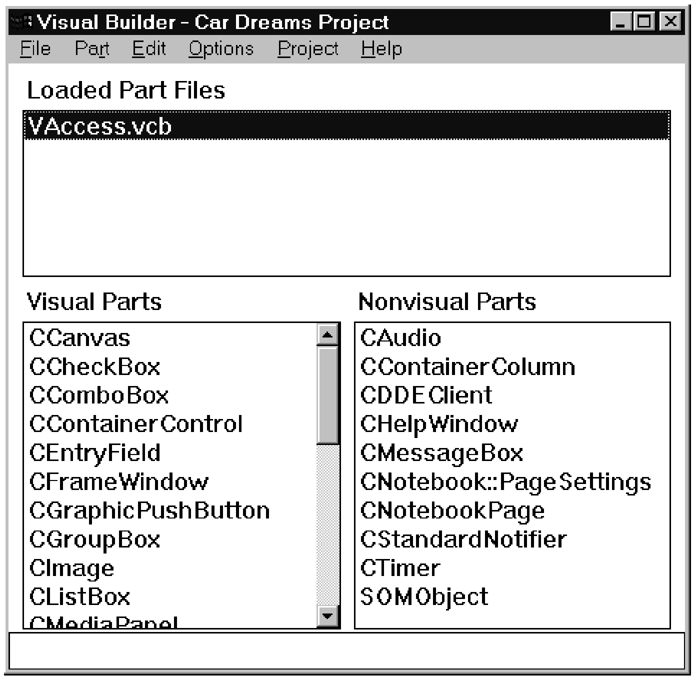
The topics in this section describe how to perform various actions on parts from the Visual Builder window.

Displaying part names

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

Once part names are displayed, you can perform actions on them, such as opening or deleting them. If you need information about loading part files, see “Loading part files” on page 33.

Figure 29 on page 108 shows the Visual Builder window with the names of the parts in the VAccess.vcb file displayed:


Visual Builder - Car Dreams Project
File Part Edit Options Project Help
Loaded Part Files
Visual Parts
Nonvisual Parts

VAccess.vcb

Visual Parts	Nonvisual Parts
CCanvas	CAudio
CCheckBox	CContainerColumn
CComboBox	CDDEClient
CContainerControl	CHelpWindow
CEntryField	CMessageBox
CFrameWindow	CNotebook::PageSettings
CGraphicPushButton	CNotebookPage
CGroupBox	CStandardNotifier
CImage	CTimer
CListBox	SOMObject
CMediaPanel	

Figure 29. Visual Builder window with VAccess.vcb parts displayed

Selecting all parts

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

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

Deselecting all parts

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

Importing part information

Using any text editor, you can create files called *part information files*, which are used to import existing COBOL classes and type definitions into Visual Builder as nonvisual parts. Part information files are normally recognizable by their .VCE extension. Refer to Appendix A, “Creating part information files” on page 223 to learn how to create part information files.

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

To import part information, do the following:

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

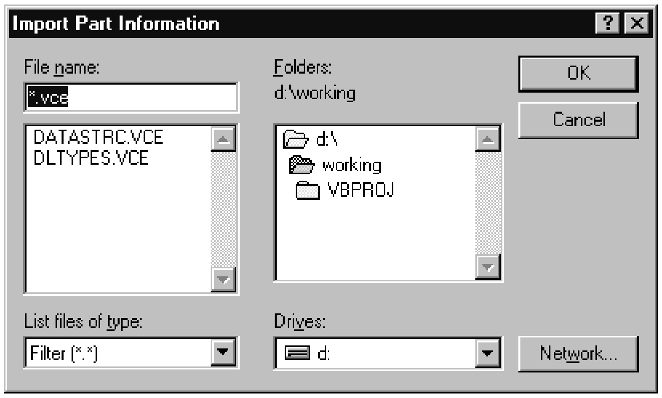


Figure 30. Import Part Information window

2. Select the part information file you want to import.
3. Select the **OK** button. The part information in the part information file is imported. Any nonvisual parts, class interface parts, and program parts it contains are displayed in the **Nonvisual Parts** list box. Any types or enumerations it contains are displayed in the **Loaded Type Information** list box. In addition, one or more part files might be created.

Exporting part information

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

To export part information, do the following:

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

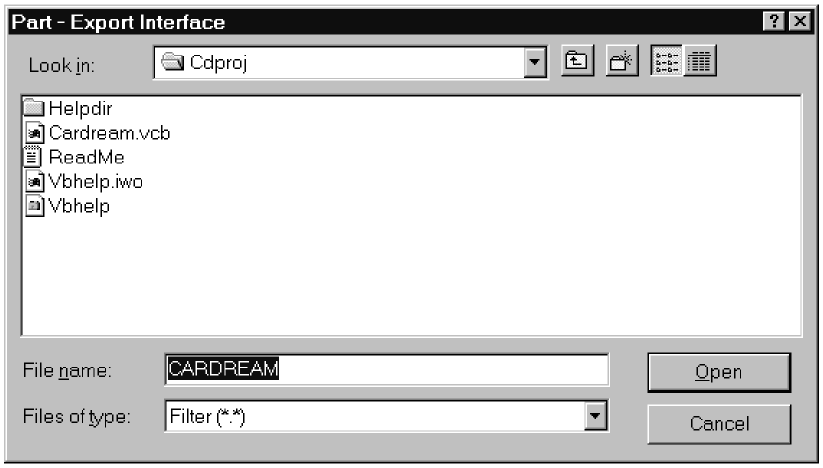


Figure 31. Part — Export Interface window

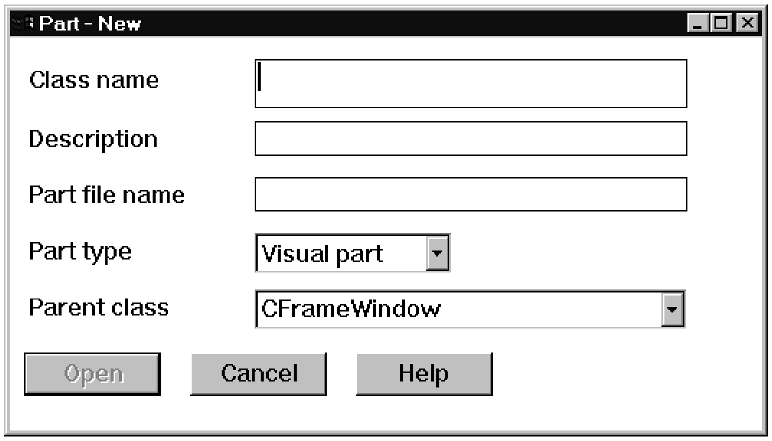
3. Type the name of the part information file (.VCE) in which you want the part information stored in the **Open file name** field. If you do not enter a file name, Visual Builder uses <partname>.VCE as the default file name. If you have multiple items selected, the default name is exported.vce.
4. Select the **OK** button. The part information maintained by Visual Builder is exported to the file name you specified in the **Open file name** field.

Creating a new part

This section provides only the basic steps for creating a new part. For a description of how to define a part once it has been created, see Chapter 7, “Creating nonvisual parts” on page 99.

To create a new part, do the following:

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



Class name

Description

Part file name

Part type: Visual part

Parent class: CFrameWindow

Figure 32. Part–New window

2. Enter the name you want to give to your part in the **Class name** field.
3. Enter a description of your part in the **Description** field. Visual Builder uses the description you enter here in the following ways:
   - If you add the part you create to the parts palette, Visual Builder displays the part's description in the information area at the bottom of the Composition Editor when the part is selected.
   - If you export the information about the part to a part information file, the description is included with the other information about the part.
4. Enter the name of the part file in which you want Visual Builder to store the part in the **File name** field. If the file does not already exist, Visual Builder creates it for you. If you leave this field blank, Visual Builder creates a part file as follows:
   - If you are using the File Allocation Table (FAT) file system and have selected **Options→Default to FAT file names**, Visual Builder creates a part file whose name has no more than eight characters. Without this selection, Visual Builder attempts to create a part file whose name is the same as the name of your part, which causes an error if your part name has more than eight characters. Refer to "Using File Allocation Table (FAT) file names" on page 39 for detailed information.
   - If you do not select **Options→Default to FAT file names**, the name of the part file is the same as the name of your part.
5. Select the type of part that you want to create in the **Part type** field. You can select one of the following:
   - Visual part
   - Nonvisual part
6. Either keep the default parent class name provided by Visual Builder in the **Parent class** field or change it.

   Note the following:

- A nonvisual part must have the CStandardNotifier class in its inheritance so it can exhibit the behavior required for all parts—a part interface (attributes, events, and actions). It must inherit this behavior from CStandardNotifier.
- A visual part must have the CFrameWindow class or CCanvas in its inheritance so it can inherit the visual behavior common to all windows or canvases, as well as part interface behavior, which CFrameWindow and CCanvas inherit from CVisualPart. The default parent class for a visual part is CFrameWindow.
- A class interface part has SOMObject for its parent class.

7. Select **Open**. One of the following occurs:
   - If you are creating a visual part, the Composition Editor is displayed.
   - If you are creating a nonvisual part, the Part Interface Editor is displayed.
8. Use the displayed editor to create your part.

Opening parts

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

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

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

You should not make any changes when there are unloaded parts or generate any code.

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

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

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

If you want to add a bitmap to the folder, see "Specifying a unique icon for your part" on page 54.

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

Opening one part

To open one part, do the following:

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

   Note: If the list boxes in the Visual Builder window are empty or if you cannot find the part, the part file that contains the part you want to open is not selected or not loaded. See "Loading part files" on page 33 if you need help loading part files.

   The Visual Builder window with parts loaded from the file VAccess.vcb is shown in Figure 29 on page 108.

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

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

Opening multiple parts

To open multiple parts, do the following:

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

   Note: If the list boxes in the Visual Builder window are empty, see "Loading part files" on page 33 if you need help loading part files.

   The Visual Builder window with parts loaded from the part file is shown in Figure 29 on page 108.

2. Select the first part you want to open.
3. Do one of the following, depending on how the other parts appear in the list:
   - If the other parts are adjacent in the list to the part previously selected, hold down the Shift key and click on the last part in the group you want to select. All parts between the first and last parts selected are now highlighted.

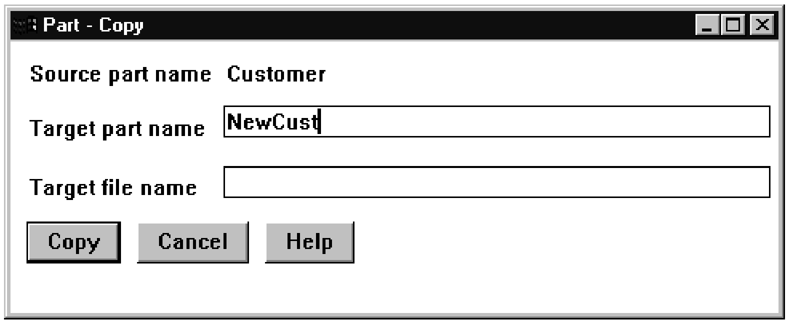
- If the parts are not adjacent in the list, hold down the Ctrl key while selecting each part.

4. Select **Part** on the menu bar.
5. Select **Open** in the pull-down menu. Visual Builder displays a separate window for each part you selected. The window displayed is the Composition Editor for visual parts or the Part Interface Editor for nonvisual parts.

Copying parts from one part file to another

To copy a part, do the following:

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


Part - Copy

Source part name Customer

Target part name NewCust

Target file name

Figure 33. Part — Copy window

The **Source part name** field shows the name of the part you selected to copy.

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

Moving parts to a different part file

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

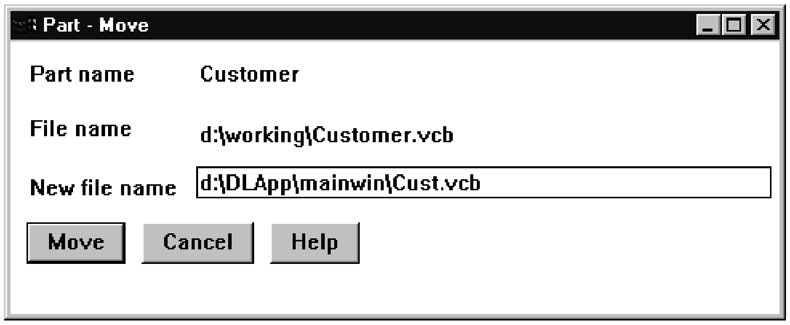
- If this part file does not exist, Visual Builder creates and loads it for you.
- If this part file already exists and is loaded, the part or parts are moved into it.
- If this part file already exists but is not loaded, Visual Builder displays a message to warn you the unloaded part file will be overwritten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

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

1. Select the part or parts that you want to move. If you do not select at least one part, the **Move** function is not available.
2. Select **Part→Move**.
3. Use the following instructions for moving one part or multiple parts:

   Moving one part

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



Part - Move

Part name Customer

File name d:\working\Customer.vcb

New file name d:\DLApp\mainwin\Cust.vcb

Figure 34. Part — Move window for moving one part

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

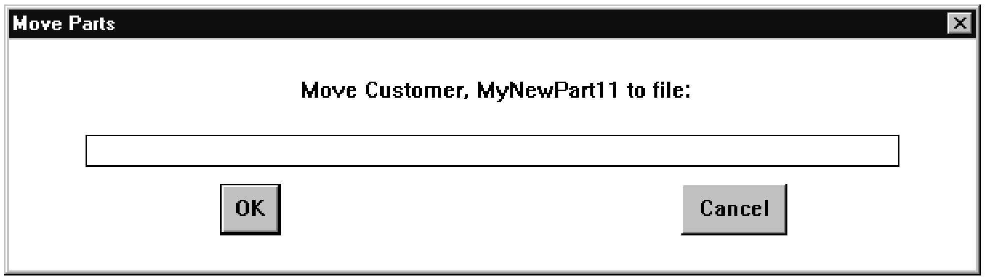
   Do the following:

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

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

   Moving multiple parts

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



Move Parts

Move Customer, MyNewPart11 to file:

Figure 35. Move Parts window for moving more than one part

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

a. In the entry field, enter the name of the part file to which you want to move the parts. If the part file is not in your current directory, specify the complete path for the part file.

b. Select the **OK** button. The parts are moved to the part file specified in the entry field.

An alternative method of moving a part is to change the name of the part file specified in the System Interface Editor. For more information, see "Specifying a different part file" on page 53.

Deleting parts from a part file

To delete a part, do the following:

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

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

2. Select **Part→Delete**.

   The following window is displayed:

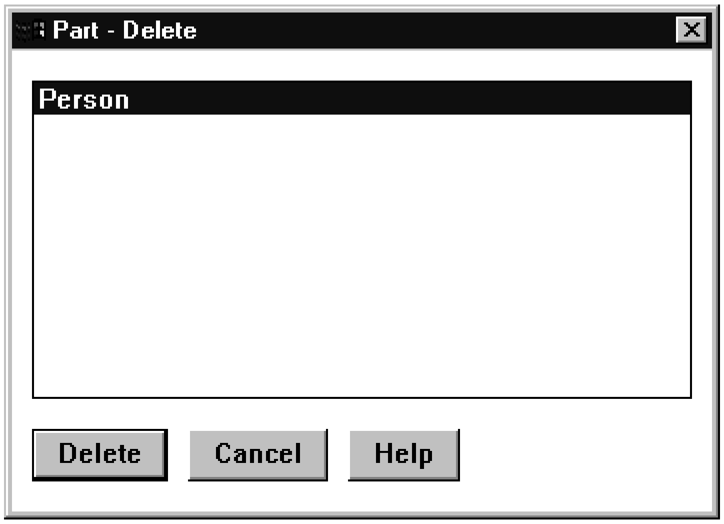


Figure 36. Part — Delete window

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

3. Select the **Delete** button.

The selected parts are deleted.

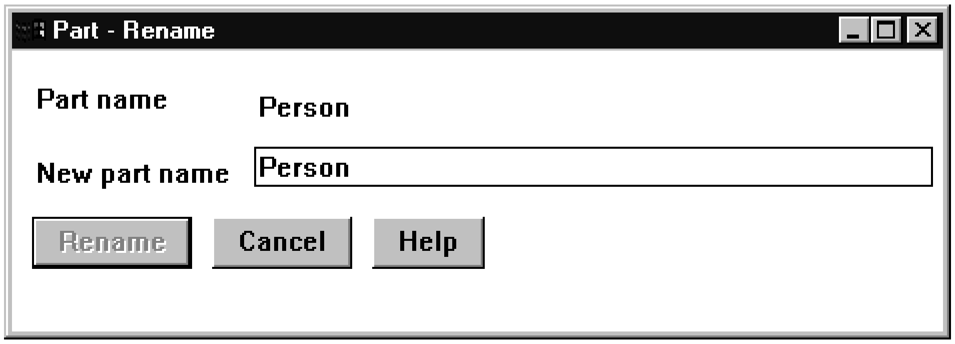
Renaming parts in part files

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

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

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

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



Part name: Person

New part name: Person

Figure 37. Part — Rename window

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

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

Working with parts on the free-form surface

The free-form surface is the large empty area in the Composition Editor where you place nonvisual parts and visual parts from the Composers, Model and Other categories.

Placing parts on the free-form surface

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

Placing a part that appears on the parts palette

1. From the left column of the parts palette, select the appropriate category. Then, from the right column, select the part you want to add. When the mouse pointer is moved over a place where the part can be placed, it changes to a cross-hair, indicating it is loaded with the part.
2. Move the mouse pointer to where you want to add the part.
3. Click mouse button 1. If you hold down mouse button 1 instead of clicking it, an outline of the part is displayed under the pointer to help you position the part. After the part is in position, release mouse button 1.

To unload the mouse pointer at any time, do either of the following:

- Select , the Selection tool, on the tool bar.
- Select **Tools→Selection tool** on the menu bar.

To add several copies of the same part, select **Sticky** on the parts palette. When **Sticky** is selected, the mouse pointer remains loaded with the part you last selected. When **Sticky** is not selected, the mouse pointer becomes unloaded after you add a part.

Placing a part that is not on the parts palette

You can place on the free-form surface any part whose part file (.VCB) is loaded by doing the following:

1. Select **Add part** from the **Options** pull-down menu. The Add Part window appears, which resembles the window shown in Figure 38 on page 119:

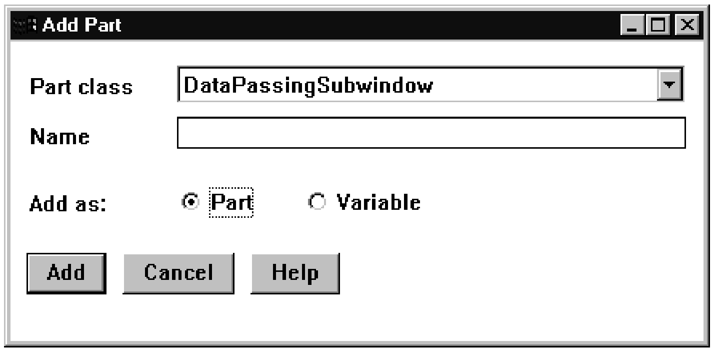


Figure 38. The Add Part window

2. Click on the down arrow of the **Part class** drop down list. Select from the list of available part classes. This list displays the part classes currently loaded in the Visual Builder. If you do not see a part class you are looking for, close the Add Part window and make sure the part file (.VCB) is loaded into the Visual Builder.
3. Type a name for the part in the **Name** field. This name will appear in the information area at the bottom of the Composition Editor when you select the part after it is placed; it is also used as the name of the part instance when you generate your part code.

   The **Name** field is optional. If you leave it blank, the part's class name is used.
4. Select either the **Part** radio button or the **Variable** radio button. Select **Part** to add a single instance of part. See "Creating the static visual parts" on page 214 for more information on static parts. Select **Variable** if you want this part to be dynamically created and destroyed as the application runs. Dynamic subwindows are an example of adding a part as a variable. See "Adding visual parts as dynamic instances" on page 214 for more information on dynamic parts.
5. Select the **Add** button to add the part. The Add Part window disappears and the mouse pointer turns into the same crosshairs used for placing a part on the free-form surface.
6. Move the crosshairs to the place where you want to add the part and click mouse button 1.

Guidelines for placing parts on the free-form surface

Following are guidelines for placing parts on the free-form surface:

- Avoid overlaying primitive parts

  You can overlay visual parts. Generally speaking, however, it is not good interface design for one primitive part to overlay another primitive part, such as one push button either completely or partially covering another push button. Be aware that completely overlaying a primitive part can cause focus problems, meaning your users might be able to see, but not select, the part.

Partially overlaying a primitive part can cause problems, too, because your users might not be able to see where the overlaying occurs. When they try to select the part partially overlaid, they might be lucky and select the right spot, or they might select the part overlaying the part they are trying to select. If you overlay primitive parts, be sure to do it in a way in which your user can understand why the primitive parts are overlaid and how to select them.

You cannot overlay, or cover up, nonvisual parts.

You can overlay composite visual parts, but as mentioned above, the primitive parts a composite part contains should not be overlaid.

- Place other parts on top of parts in the Composers category

  Parts included in the Composers category have a special behavior; these parts can contain other visual parts that are placed on top of them. The parts the Composers part contains automatically become subparts of the Composers part. For example, if you place an entry field, a list box, and two push buttons in a canvas, the canvas contains the other parts and they in turn become the canvas' subparts.

  Note: A subpart of a frame window takes the size of the frame window.

  The following table lists each of the Visual Builder categories and specifies how you can use the parts in each category.

Figure 39. Categories and how you can use their parts

Category	**Use Parts to Contain Other Parts?**	**Use Parts as Subparts?**
Buttons	No	Yes
Data entry	No	Yes
Lists	No	Yes
Frame Extensions	No	No
Sliders	No	Yes
Composers	Yes	Yes
Other	No	No

- **Use supplementary composite parts as subparts**

  Suppose you create a visual composite part that consists of a canvas on which you have placed other visual parts, such as static text and entry fields, with each entry field connected to a variable, as shown in Figure 40 on page 121.

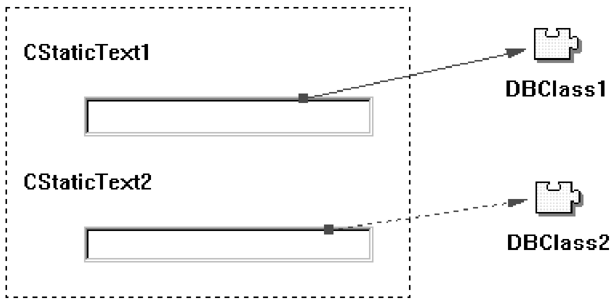


Figure 40. Visual parts connected to nonvisual parts

Assume this part is not your main composite part but is instead a supplementary composite part you want to use in your application's user interface. Then you place this part in your main composite part, such as in a frame window, as shown in Figure 41. You place the supplementary composite part and work with it as one part, not as a canvas and separate entry fields and static text. The frame window contains the entire supplementary composite part, which becomes a subpart of the frame window.

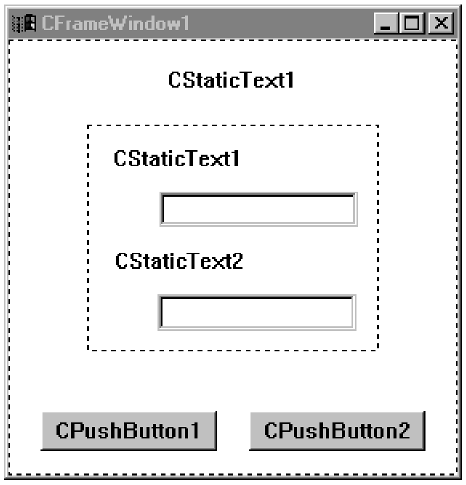


Figure 41. Composite part placed in frame window as subpart

One of the first things you have probably noticed is the connections for a supplementary composite part are not displayed when that part is added to another part. The connections and subparts are still there; you just cannot see them because you cannot edit them directly from the Composition Editor window containing the main composite part. Also, you cannot select the individual static text parts, entry field parts, or their connections in the supplementary part you nested in your main part.

To change the connections or the default text on the static text parts, or to do anything else to alter this part, you must edit the part indirectly, as described in “Editing parts placed on the free-form surface” on page 138.

Selecting and deselecting parts

Before you can perform an action on a part you have placed on the free-form surface, such as sizing it, you must first select the part. The name of the part currently selected is displayed in the information area at the bottom of the Composition Editor. If more than one part is selected, then `*Multiple selection*` is displayed.

You cannot select parts and connections together. They are mutually exclusive. However, if you delete a part connected to other parts, Visual Builder deletes the connections in addition to the part.

When a part is selected, small boxes, called selection handles, are displayed on its corners. If more than one part is selected, the one you selected last has solid selection handles, indicating it is the *anchor part*. The other selected parts have hollow selection handles as shown in the following figure:

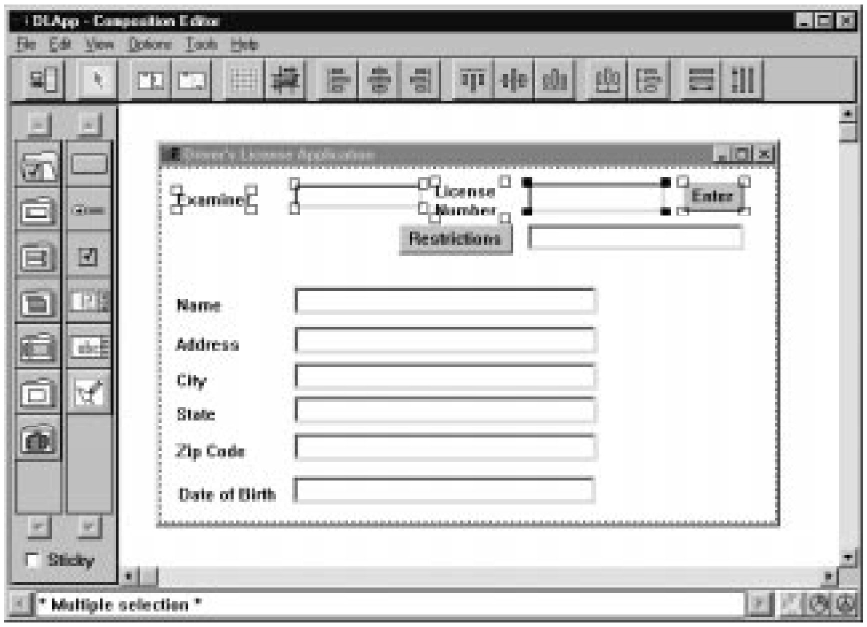


Figure 42. Multiple parts selected with the Entry Field as the anchor part

Some parts are not sizable and, therefore, do not have any selection handles. These parts have their background reverse colored. Parts with this behavior include menus.

The following sections describe how to select and deselect a single part and multiple parts.

Selecting a single part

To select a part you have placed on the free-form surface, click on the part with mouse button 1. If other parts are already selected, they are deselected automatically.

Selecting multiple parts

Selecting multiple parts lets you perform the same operation on several parts at once. To select multiple parts, do one of the following:

- Hold down the Ctrl key in OS/2 or the Shift key in Windows and click mouse button 1 on each additional part you want to select.
- Hold down mouse button 1 instead of clicking it. Then move the mouse pointer over each additional part you want to select. After you select the parts, release mouse button 1. (This method works only in OS/2.)

Note: Depending on the operation you want to perform, remember to consider which part you want to be the anchor part because that is the part you want to select last. For example, if you select two parts because you want to match the width of one part to the width of the other, the part you select last is the anchor part, the part whose width is used for the operation.

Deselecting parts

To deselect a part after you have selected it, do one of the following:

- Hold down the Ctrl key in OS/2 or the Shift key in Windows and click on the selected part with mouse button 1.
- Click mouse button 1 in a clear spot on the free-form surface.

When the selection handles disappear, you know the part is no longer selected.

To deselect multiple parts, do the following:

1. Hold down the Ctrl key in OS/2 or the Shift key in Windows.
2. Click and release mouse button 1 on a selected parts.
3. Repeat the previous step until all parts you want to deselect have been deselected.

Manipulating parts

Once a part is added to the free-form surface, you can manipulate it in a number of different ways. The following sections explain each of those ways.

Displaying pop-up menus

To display the pop-up menu of a part, click on the part with mouse button 2. The pop-up menu displays the operations you can perform on that part.

A part does not have to be selected for you to display its pop-up menu. The pop-up menu displayed is for the part the mouse pointer is over when mouse button 2 is clicked, even if another part is selected.

Copying Parts

To copy parts by dragging them, do the following:

1. Select all the parts you want to copy. If you only want to copy one part, you do not have to select it.
2. Move the mouse pointer over the part you want to copy or one of the selected parts.
3. Hold down the Ctrl key and mouse button 2 in OS/2 or the Ctrl key and mouse button 1 in Windows.
4. Drag a copy of the part or parts by moving the mouse pointer to a new position. An outline of the part or parts is displayed to help you with positioning. When you are copying multiple parts, the outlines of each part move together as a group.
5. Release the Ctrl key and mouse button when the part or parts are where you want them to be. A copy of the part or parts appears where you positioned the outline or outlines.

Copying parts using the clipboard

To copy parts by using the clipboard, do the following:

1. Select all the parts you want to copy.
2. From the **Edit** pull-down menu, select **Copy**. A copy of each selected part is placed on the clipboard.
3. Select **Paste** from the **Edit** pull-down menu when you are ready to use the parts. The mouse pointer turns to crosshairs to show it is loaded with the copied parts.
4. Position the mouse pointer where you want the parts to be copied.
5. Click mouse button 1. Copies of the parts are pasted at the position of the mouse pointer.

Parts that you copy remain on the clipboard until you copy something else. Therefore, you can continue to paste copies of those parts by selecting **Paste**, positioning the mouse pointer, and clicking mouse button 1.

If you select **Paste** and then decide against pasting the parts, you can unload the mouse pointer by either selecting the Selection tool on the tool bar or by selecting **Tools→Selection tool** on the menu bar.

Deleting Parts

To delete one or more parts, do the following:

1. Select all of the parts you want to delete. If you are deleting just one part, you do not have to select it.
2. Position the mouse pointer over the part you want to delete or one of the selected parts.
3. Click mouse button 2.
4. From the part pop-up menu, select **Delete**. The part or parts are deleted.

You can also delete a part by pressing the Delete key after selecting the part.

Any connections between the part you are deleting and other parts are also deleted. Visual Builder displays a message to alert you of this. However, the **Edit→Undo** function also restores any connections removed when you deleted the part.

Editing Text Strings

Some visual parts, such as push buttons and menus, have text strings. To directly edit a part's text string, do the following:

1. Hold down the Alt key.
2. Click mouse button 1 on the text string.
3. Edit the text string.
4. When you have finished, do either of the following:
   - Click mouse button 1 anywhere outside of the text string.
   - Press Shift+Enter.

You can also use this *direct editing* technique to edit the names of nonvisual parts. The name of a nonvisual part is displayed directly below its icon.

Renaming parts on the free-form surface

When you use parts in the Composition Editor, Visual Builder gives those parts a name based on the names given to the parts on the parts palette or the names you specify when you place parts on the free-form surface. For example, the first push button part that you use is named PushButton1. When you select this part, the information area at the bottom of the Composition Editor shows the message “PushButton1 selected.” The second push button you use is named PushButton2, the third is named PushButton3, and so forth. These default names are assigned to help Visual Builder distinguish one part from another, as well as the connections between parts, when you generate the code to build your application.

If you want to give parts names that are more descriptive or meaningful to your application, you can do so as follows:

1. Move the mouse pointer over the part whose name you want to change.
2. Click mouse button 2 to display the pop-up menu for the part.
3. Select **Change name**. A Name Change Request window is displayed. Figure 43 shows a Name Change Request window for a push button part.

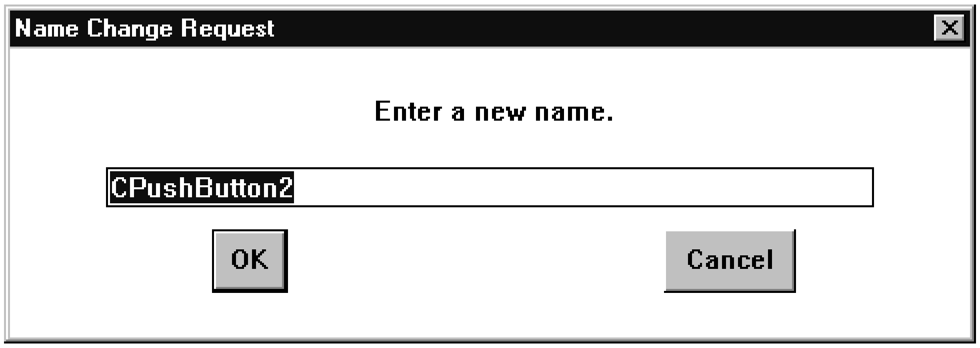


Figure 43. Name Change Request window for a push button part

4. Type a new name in the entry field. This name must be a valid COBOL name.
5. Select **OK**. Visual Builder changes the name of the part to the name that you typed in the entry field.

You can also change a part's name by opening the part's settings notebook and changing the name in the **Subpart name** field.

Arranging parts

You can arrange parts on the free-form surface in a number of different ways. The following sections explain each of those ways.

Moving parts

To move a part, move the mouse pointer over the part, hold down mouse button 2 in OS/2 or mouse button 1 in Windows, and move the mouse pointer to drag the part to the new position.

You can move several parts at once by first selecting all the parts you want to move and then dragging one of the parts as described. All of the selected parts will move together, maintaining their position relative to each other.

Positioning parts on the grid

The free-form surface has a grid you can use to position parts. In addition, parts that can contain other parts (for example, any Composers part, such as a frame window) have a grid associated with them. You can use this grid to align and evenly space subparts Composers parts contain.

To position the upper-left corner of parts to the nearest grid coordinate, do the following:

1. Select all the parts you want to position to the grid.

   Note: If the parts you select are subparts, they are positioned to the grid set up inside the Composers part, not the grid for the free-form surface.

2. Select ▦ , the **Snap To Grid** tool.

You can automatically position a part to the nearest grid coordinate when it is added to the free-form surface or a Composers part by selecting **Snap on drop** from the **Options** pull-down menu.

Specifying grid spacing

To specify the grid spacing, do the following:

1. From the pop-up menu of a Composers part or the free-form surface, select **Set grid spacing**.
2. Specify the horizontal and vertical distance between the lines of the grid in pixels.

Showing and hiding the grid

To toggle between showing and hiding the grid for the free-form surface, do one of the following:

- If no parts are selected, you can select ▦ , the **Toggle Grid** tool to toggle the grid for the free-form surface.
- If a Composers part is selected, selecting the **Toggle Grid** tool toggles the grid for the Composers part instead of the free-form surface.

<u>Toggling between showing and hiding the grid for a Composers part</u>

To toggle between showing and hiding the grid for a Composers part, do one of the following:

- Select the Composers part and the select ▦ , the **Toggle Grid** tool.

- From the Composers part's pop-up menu, select **Toggle Grid**.

Sizing parts

To change the size of a part, select it and use mouse button 1 to drag one of the selection handles to the new position. An outline of the part is displayed under the mouse pointer to show you the new size of the part.

You can size several parts at once by first selecting all the parts you want to size.

To size a part in only one direction, press and hold the Shift key while using mouse button 1 to size the part. Holding down the Shift key prevents one dimension of the part from changing while you resize the other dimension. For example, to change the width of a part but prevent its height from changing, hold down the Shift key while changing the width.

You can also size a part to the grid coordinates by selecting **Snap on size** from the **Options** pull-down menu.

Note: The size and position of a window subpart at run time is determined by the size and position specified during construction. Window subparts can be moved to the lower right while connecting to other parts.

Matching part sizes

To size parts to the same width or height of another part, do the following:

1. Select all the parts you want to size, making sure the last part you select is the part whose size you want the others to match.
2. Select one of the following sizing tools from the tool bar:

   Match Width **Match Height**

   The size of all the parts you selected, with the exception of the last part, changes to match the size of the last part selected.

Aligning parts

To align parts to the same position as another part, do the following:

1. Select all the parts you want to align, and then select the part you want the others to be aligned with.
2. Select one of the following alignment tools from the tool bar:

   Align Left **Align Top**

Align Center	**Align Middle**
Align Right	**Align Bottom**

Spacing subparts within Composers parts

To evenly space subparts within their Composers part, do the following:

1. Select all the parts you want to evenly space.
2. Select one the following spacing tools from the tool bar:

   Distribute Horizontally

   Distribute Vertically

Spacing parts within a bounding box

To evenly space parts within the unseen bounding box that contains the selected parts, do the following:

1. Select all the parts you want to evenly space. You must select a minimum of three parts.
2. From the pop-up menu of one of the selected parts, select **Layout→Distribute**, and then one of the following:

   Horizontally in bounding box
   Evenly distributes the selected parts within the region bounded by the leftmost edge and rightmost edge of the selected parts.

   Vertically in bounding box
   Evenly distributes the selected parts within the region bounded by the topmost edge and bottommost edge of the selected parts.

For more information on tool bar tools, see “The tool bar” on page 44.

Changing settings for a part

The settings notebook of a part provides a way to display and set attributes and options for the part. To ensure that initial values are set the way you expect, always explicitly set them.

Opening the settings notebook for one part

To open the settings notebook for a part, move the mouse pointer over the part and do one of the following:

- Double-click mouse button 1.
- Click mouse button 2 and select **Open settings** from the part's pop-up menu.

Opening settings notebooks for multiple parts

You can open the settings notebooks for multiple parts by doing the following:

1. Select the parts whose settings you want to change.
2. Move the mouse pointer over one of the selected parts.
3. Click mouse button 2.
4. Select **Open settings** from the pop-up menu.

   Visual Builder opens a settings notebook for each of the selected parts.

Navigating through a settings notebook

You can navigate through the notebook pages in the following ways:

- To turn the pages of a notebook, use the small left- and right-arrow buttons at the lower-right corner of each page.
- To move to a different settings category, select one of the tabs to the right of the pages.

  Note: When a category has more than one page, the page number and total number of pages within the category are displayed at the bottom of the page.
- If all of the category tabs do not appear on the pages of the notebook, small left- and right-arrow push buttons are displayed to the left of the category tabs, and small up- and down-arrow push buttons are displayed above and below the category tabs, Use these buttons to move through the available category tabs.

About the settings pages

The following list contains a description of each of the pages a settings notebook might contain:

General
A page for setting the name of the part, any static text that might appear on the part, and other part-specific settings. For example, the **General** page for an CMenuItem part contains a group box for setting the Command Key for the menu item. Refer to the online help for descriptions of specific settings for parts.

Control
This page allows you to specify information for the part in its role as a control part, such as a window ID, and whether the part is available for the user to select.

Styles
Select the styles you want initially set for the part. Some styles are mutually exclusive.

Display Type
Specify the picture clause you want to use for alphanumeric data.

Validation
Entry fields can be validated before data is returned to other parts. This page specifies the validation rules to use.

Color
A page that allows you to change the color of the part.

Changing the color

To change the color, do the following:

1. In the **Color area** group box, select the area, such as foreground or background, you want to change.
2. Do one of the following:
   - If you want to specify red-green-blue values, select the **RGB** check box and specify values in the fields in the **RGB values** group box.
   - If you want to select a color by its name, deselect the **RGB** check box and select a color from the **Colors** drop-down list box.
3. Select either the **Apply** button to see how this color looks for your part without saving the change or the **OK** button to close the settings notebook and save the color change.

Size/Position
This page allows you to specify the size and position of a part.

Specifying the size and position of a part

To specify the size and position of a part, do the following:

1. In the **x** and **y** fields, specify the initial X and Y coordinates for the part. These coordinates determine the position of the part's upper-left corner.
2. In the **width** and **height** fields, specify the number of pixels for the width and height of the part.

Note: The size and position of non-window subparts is determined by their visual placement in the containing part. This overrides any settings in the subpart. Window subparts are visually positioned and sized by themselves.

Font
This page allows you to specify the font to be used for the part.

Changing the font for a part

To change the font for a part, do one of the following:

- If you know the name and size of the font you want to use, you can enter them in their respective fields.
- If you do not know the name and size of the font you want to use or if you want to change the style or emphasis, select the **Edit** button. Visual Builder displays

a standard font dialog from which you select the name, size, style, and emphasis you want to use for the part's font.

Note:

1. Fonts are system dependent.
2. A change in font may require a part to be resized. At run time, an oversized cursor prevents data entry.

Activating settings changes

After you make changes to the settings in the settings notebook, you can activate them in the following ways:

- Select the **OK** button to immediately activate and save the changes you made and to close the settings notebook.
- Select the **Apply** button to apply the changes you have made and keep the settings notebook open.

  This allows you to see whether you need to modify any of the changes you made. The changes remain applied until you change them again.

Select the **Cancel** button to close the settings notebook. If you made changes and selected the **Apply** push button, the changes are saved.

Select the **Help** button for descriptions of the settings in the settings notebook.

Using the generic settings notebook

When you create a part, Visual Builder provides a settings notebook for your part. The settings notebook for your part has one page, which contains the following types of settings:

- An entry field for each attribute with a set method
- A check box for each Boolean attribute

If your part has no attributes, the page displays a message saying that there are no values to set.

Note: For most nonvisual parts, you can use the generic settings notebook to provide initial values for attributes. For nonvisual data parts, provide initial values as follows:

1. Connect the *ready* event of the part you are editing to the attribute of the nonvisual data part you want to initialize. The resulting connection appears as a dashed line because the connection is incomplete and requires a parameter.
2. Double-click on the dashed line to open the settings window of the connection.
3. Click on the **Set parameters** button. The **Constant parameter settings** notebook appears.

4. Enter the appropriate value for the attribute listed.
5. Click on the **OK** button to save your entry and close the notebook.
6. Click on the **OK** button to close the settings window of the connection. The dashed line becomes a solid line, signifying the connection is complete.

When you nest your part in another part, follow these steps to modify the attribute settings for that particular instance of your part:

1. Place your part in the Composition Editor.
2. Move the mouse pointer over your part and click mouse button 2.
3. Select **Open settings** from the pop-up menu.

   Visual Builder displays the settings notebook for your part.
4. Fill in the entry fields with appropriate values.
5. Click on the **OK** button to set those values and close the settings notebook.

Listing parts within a composite part

The Parts List window provides a way to display an ordered list of the parts nested in a composite part. At first, parts are listed in the order they were dropped in the composite part. If you then change the tabbing order of parts that have tabbing set, Visual Builder rearranges the list to reflect the updated tabbing order. For more information on tabbing, see “Setting the tabbing order” on page 135.

Note: In Windows, make sure your system palette is set to 256 colors or fewer before trying to list parts.

To list parts nested in a composite part, do the following:

1. Open the composite part.
2. Click on the free-form surface with mouse button 2. The part’s pop-up menu appears.
3. Select **View parts list**. The Parts List window opens. At first, the Parts List window displays only the immediate subparts of the selected part. An ⊞ expansion icon appears next to each part that contains subparts of its own. To see those parts, select the expansion icon.

The parts list for a default COBOL Visual Builder project part is shown in Figure 44 on page 134.

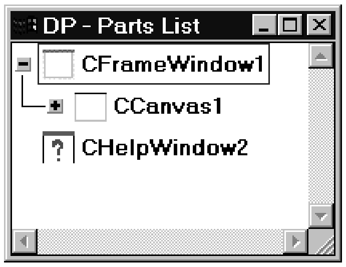


Figure 44. Parts list for a sample part

Changing depth order within a composite part

Depth order is the order in which parts are stacked on the application desktop. Parts lower in the depth order overlay at least a portion of parts higher up. An example of this is a push button on a canvas. The canvas appears higher (or first) in the depth order; the push button, which lies on top of the canvas, appears lower (or later).

Visual Builder assigns the depth order as parts are dropped. Depth order is not linear, but hierarchical, depending on the arrangement of Composers parts.

You can change the depth order of parts in a composite part with a single parent by dragging items in the parts list. To change the depth order, do the following:

1. Open the parts list for the composite part by selecting **View parts list** from the composite part's pop–up menu. Refer to “Listing parts within a composite part” on page 133 for more information on part lists.
2. To move more than one part, do one of the following:
   - If the parts are adjacent in the list, select the first part in the group to be moved. Then hold down the Shift key and click on the last part in the group to be moved. All parts between the first and last parts selected are now highlighted.
   - If the parts are not adjacent in the list, select the first part. Then hold down the Ctrl key while selecting the other parts.

   If you want to move only one part, you do not need to select it first.
3. Using mouse button 2, drag the selected parts to their new location in the depth order.

If tabbing has been set for any of the parts moved, changing the depth order also changes the tabbing order. To find out more about tabbing order, see “Setting the tabbing order” on page 135.

Performing other operations on parts in the Parts List window

You can perform some of the same operations on parts in the parts list that you can perform on the parts on the free-form surface. Visual Builder provides pop-up menus that contain the enabled operations for each part in the parts list.

To perform an operation on a part in the parts list, do the following:

1. Move the mouse pointer over the part.
2. Click mouse button 2 to open the part's pop-up menu.
3. Select the operation you want to perform.

Setting the tabbing order

The tabbing order is the order in which the input focus moves from part to part as the user presses the Tab key. The tabbing order can also indicate the order in which the input focus moves among parts within a tab group as the user presses the arrow keys. Tabbing order is related to depth order, as discussed in "Changing depth order within a composite part" on page 134.

The tabbing order can only be set or displayed for parts placed within a Composers part. For example, if you place a row of push buttons in a frame window, you can set the tabbing order for the push buttons. Consider the part shown in Figure 45.

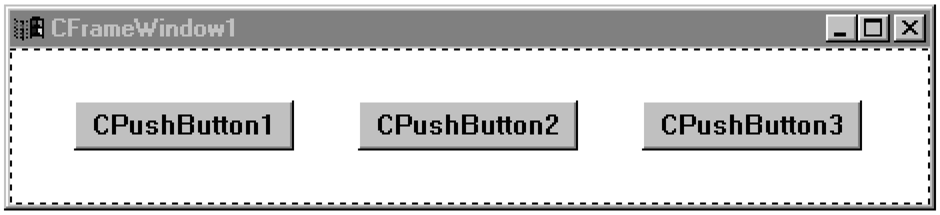


Figure 45. Frame window with push buttons

The initial tabbing order is determined by the order in which you place the parts on a Composers part. Also, the first part in the tabbing order receives the initial input focus. For example, if the first part in the tabbing order is a push button, that push button receives the initial input focus when the application starts.

To display the tabbing order, open a parts list for the Composers part that contains the push buttons. Within the parts list, you can change the positions of parts in the tabbing order.

Changing the tabbing order

Because the order in which parts are placed on a Composers part determines the tabbing order, you will probably need to change the order as you add or rearrange parts. For example, suppose you decide to rearrange the three push buttons from the example in the preceding section so that PushButton3 is between PushButton1 and PushButton2, as shown in Figure 46.

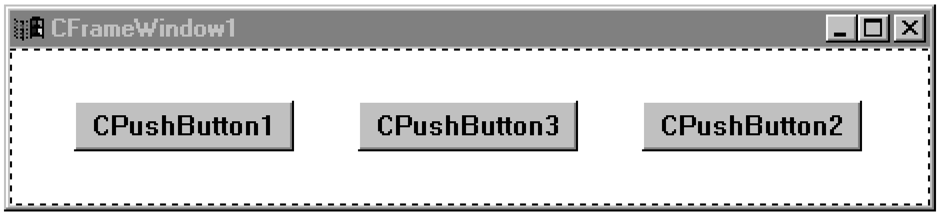


Figure 46. Rearranged push buttons

The tabbing order of these push buttons is PushButton1, PushButton2, PushButton3, even though PushButton3 is now between PushButton1 and PushButton2.

To change the position of a part within the tabbing order, do the following:

1. Open a parts list for the CCanvas part that contains the push buttons.
2. Move the mouse pointer to the part in the list whose position you want to change.
3. Press and hold mouse button 2.
4. Drag the part icon to its new position.
5. Release mouse button 2.

The changed tabbing order is shown in Figure 47.

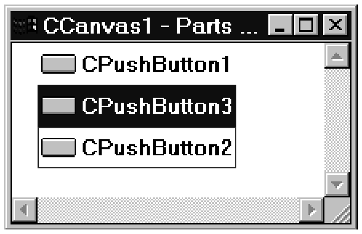


Figure 47. Resequenced tabbing order

You cannot move a subpart to a new Composers part by changing the tabbing order. You must do this by moving the parts themselves in the Composition Editor.

Setting tab stops and groups

If you want the user to be able to move the input focus to a part using the Tab and backtab keys, do the following:

1. Select the part.
2. Open the part's pop-up menu.
3. Select **Set tabbing→Tab stop**.

If you want the user to be able to move the input focus to a part with the keyboard arrow keys, do the following:

1. Select the part.
2. Open the part's contextual menu.
3. Select **Set tabbing→Group**.

All parts in the tabbing order below the part with **Group** selected are included in the group.

To start another group, select **Set tabbing→Group** for the part you want to be the first part in the new group. If a part has both **Group** and **Tab stop** selected, a user can tab to the first part in the group and then use the arrow keys to move to the other parts in the group.

Special considerations for radio buttons and entry fields

When you put radio buttons in groups, they become mutually exclusive within their group. For example, suppose you have four consecutive radio buttons in your list and you select **Group** for RadioButton1 and RadioButton3. In this case, RadioButton1 and RadioButton2 become mutually exclusive in their group, with RadioButton3 and RadioButton4 mutually exclusive in their group, as well. Tab stops are also set so a user can tab between the two groups.

Consider setting a tab stop on each entry field that a user can type in to allow the user to move the input focus from one entry field to another. Read-only entry fields do not need a tab stop, and arrow keys only move the cursor within an entry field; only the Tab key, backtab key, and mouse can change the input focus from one entry field to another.

Style guidelines for setting groups and tab stops

The following are some typical style guidelines for setting groups and tab stops:

- The position of the parts in the tabbing order should be the same as the order in which they are displayed in the window, from left to right and then top to bottom.
- Parts that are not in groups, such as entry fields and list boxes, should have **Group** and **Tab stop** selected.
- Each group of related parts, such as check boxes and radio buttons, should be put within an outline box or a group box. If there is only one group of related parts, such as push buttons, you do not need to put them within an outline box or group box. Select only **Tab stop** for these parts.

- Parts that should not receive input focus, such as static text parts, should not have either **Group** or **Tab stop** selected.

Editing parts placed on the free-form surface

Suppose you create a composite part, add it to another composite part you are creating, and then realize that you need to change the first composite part. With Visual Builder you do not have to start over. It provides a way for you to edit the part that needs to be changed right from the free-form surface.

The only exception is the parent parts that Visual Builder provides. Visual Builder does not allow you to modify these parts. This includes all of the parts in the part file VAccess.vcb. If you place one of these parts on either a Composers part or the free-form surface, you can modify the behavior of the part by doing the following:

- If you want to add an action to the part, consider connecting to a method that belongs to the composite part, instead. Write a method if you need to perform an action of limited use—that is, one that you do not anticipate using very often and you do not want derived parts to inherit.
- If you want to add a new feature you plan to use often, create a new part derived from the parent part. For example, to add a new feature to an CEntryField part, create a new visual part whose parent is a canvas and contains a CEntryField part. You can then add as many new features to your new part as you need. You can also include features of the CEntryField part to your new part by promoting them to your new part.

If you need to edit a part added to the part you are editing, do the following:

1. If you have not already done so, load the part file containing the part you want to edit.

   Note: See “Loading part files” on page 33 if you need information about loading part files.

2. Move the mouse pointer over the part you want to edit.
3. Click mouse button 2. The part’s pop-up menu appears.
4. Select **Edit part**.

   Visual Builder displays the appropriate editor for the part, as follows:

   - If you are editing a visual part, Visual Builder displays the Composition Editor
   - If you are editing a nonvisual, Visual Builder displays the Part Interface Editor.

5. Edit the part.

If you want to promote any of the features of the parts used to create the composite part you are editing, doing so now keeps you from having to edit this part again later. See “Promoting a part’s features” on page 139 if you need more information about doing this.

6. Select **File→Save** to save the part.
7. Close the editor by doing one of the following:
   - Double-click on the system menu icon.
   - Select **File→Exit**.

   The editor you are using disappears and you are returned to the Composition Editor you were using previously. However, Visual Builder has not applied the changes you made to the part you just edited, so those changes are not visible yet.
8. Select **File→Save** to save the original part.
9. Close the Composition Editor for the original part that you were editing, as described previously.
10. Reopen the original part you were editing by double-clicking on the part's name in the Visual Builder window.

    You should now be able to see the changes you made to the part that you edited.

Promoting a part's features

"Guidelines for placing parts on the free-form surface" on page 119 discusses the relationship between parts in the Composers category and parts placed on top of them, called subparts. When you create a visual part that consists of a part from the Composers category that contains subparts, you can then place that visual part on top of another part from the Composers category. However, if you do this, the features of the subparts in the visual part that you created are not automatically available. You must promote these features to use them in connections.

For example, suppose you create a visual part called Buttons whose parent class is CCanvas. This part consists of a canvas part that contains three push button parts. Here is what the Buttons part looks like:

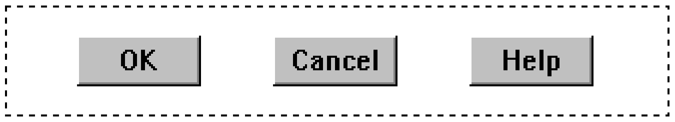


Figure 48. Buttons part

Suppose you create another visual part whose parent class is CFrameWindow and then add the Buttons part to the frame window part. Here is what the frame window part with the Buttons part looks like:

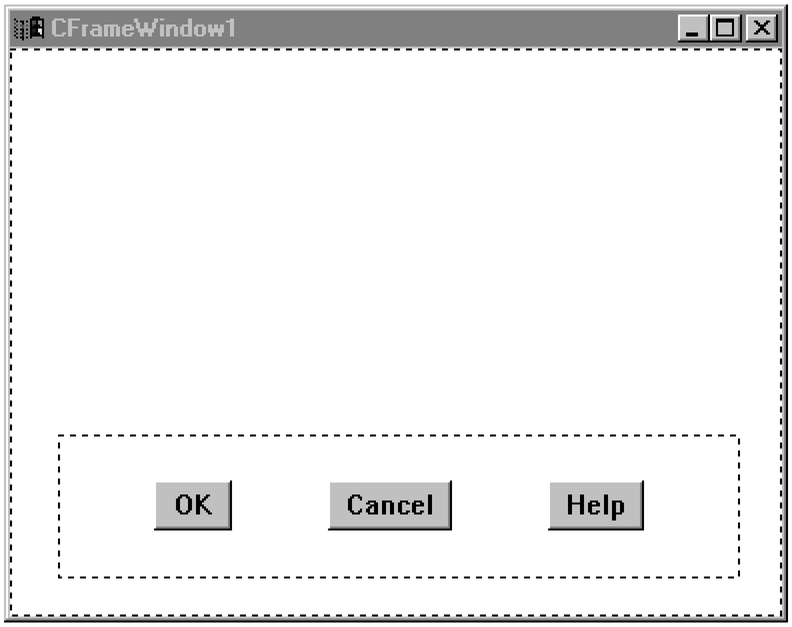


Figure 49. Frame window with Buttons part

Now, suppose that you want to connect the press feature of the **Cancel** push button to the close feature of the frame window so the window closes whenever the **Cancel** push button is selected. However, features of the push button parts are not available for connections because the push buttons are subparts of the canvas.

When you nest a part such as Buttons, only the features of the Buttons part's parent classes (CCanvas, CFrameWindow, and so forth) are available in the connections menu for the Buttons part. To use the features of the push button parts, you must promote them to the Buttons part. You can do this either before or after you add the Buttons part to the frame window.

To promote features of several subparts, we recommend using the **Promote** page of the Part Interface Editor. For information about promoting a part's features from the Part Interface Editor, see "The Promote page" on page 69.

Tearing off an attribute

Select **Tear-Off Attribute** from a part’s pop-up menu to work with an attribute as if it were a stand-alone part. The torn-off attribute is not actually a separate part, but a variable that either represents the attribute itself or points to it.

When you select **Tear-Off Attribute**, Visual Builder displays the list of attributes for the part you are tearing from. After you select an attribute from the list, you can drop the torn-off attribute on the free-form surface. Visual Builder creates an attribute-to-attribute connection between the original part and the torn-off attribute. You can then make other connections to or from the torn-off attribute. See Chapter 9, “Learning to use connections” on page 153 if you need information about attribute-to-attribute connections.

You might want to tear off an attribute to do the following:

- Allow direct access from one part that is nested inside of another
- Enable direct access to an part's events and actions

Undoing and redoing changes in the Composition Editor

If you change something in the Composition Editor and then decide you should have left things as they were, select **Undo** from the **Edit** pull-down menu to restore the part to its previous state. You can undo as many operations as you want, up to when you opened the Composition Editor.

If you undo an operation and then decide you did the right thing in the first place, select **Redo** from the **Edit** pull-down menu. **Redo** restores the part to the state it was in before the last **Undo**, including any connections deleted.

If you are not sure which operations you want to undo or redo, select **Undo/Redo list** from the **Edit** pull-down menu to display two lists of operations, one for undoing and one for redoing. From these lists, you can select an operation and then select the **Undo** or **Redo** push button. The operation that you select and all of the operations listed below it are undone or redone.

Note: **Undo**, **Redo**, and **Undo/Redo list** only affect operations you perform on the free-form surface and parts palette in the Composition Editor. They have no affect on any of the functions in the **File** pull-down menu, such as **Save**, **Save as**, and **Save and generate**, which you cannot undo.

Sharing parts with others

The most effective parts can be reused with little effort by others that are previously not familiar with 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 (.VCB) for immediate use in Visual Builder. This method is preferred for distributing visual parts.
- Providing part information files (.VCE) for import into Visual Builder. This method works for almost any type of part but must be used to distribute program parts.

Developing Applications

In this chapter, the term *part consumer* refers to the recipient of the parts you distribute.

Providing part files (.VCB)

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 users can modify the parts.

If you want to distribute primitive visual or composite parts, you must provide part information files (.VCE) instead. For more information, see “Providing part information files (.VCE).”

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 user:
    - A part file that contains the parts to be distributed
    - Any additional code files (.CPV or .CBV) 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 user loads the part files and generates source code.

Providing part information files (.VCE)

You can share nonvisual parts, class interface parts, or program groups through part information files. One advantage to this method is that you can prevent the part user from modifying the parts.

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.
2. Create and assign any icons needed for the new parts.
3. Supply the following to the part user:
    - The part information file that contains the parts to be distributed
    - Any files (.CPV, .CBV, .IMP, .DLL, .RCI, .ICO, .BMP, .ODX, .RCH, .RCS, .CPH, .DEF) needed to 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 user imports the part information file into Visual Builder to create part files (.VCB). If you provided a DLL with the part information file, no code generation or further compilation is required.

For information about coding your part information files, refer to “Describing part interfaces in part information files” on page 223.

Providing enumerations and types

You can also use part information files to provide *enumerations* and *types*.

Visual Builder uses types to match interfaces and generate code. Types represent ordinary data declarations. Visual Builder replaces the type name with the COBOL data name when it generates code. You can choose between declaring the full structure in a part information file, or declaring the higher level record items in a COPY file that is referenced by the part information file.

Enumerations are special types that are handled a little differently by Visual Builder. They are integer types with 88-level record items.

For information about coding enumerations and types in your part information files, refer to “Describing part interfaces in part information files” on page 223.

Adding categories and parts to the parts palette

You can modify the parts palette at any time and from any of the Visual Builder editors or from the Visual Builder window.

One reason to modify the parts palette is so you can quickly and easily place parts you have created and use often on the free-form surface. Otherwise, you have to place them by selecting **Options→Add part**, which requires you to know the exact class name of the part you are adding.

Another reason to modify the parts palette is to give everyone who is working on the same project access to the same set of standardized parts. Your company could have a parts builder who builds these standardized parts and puts them in a category on the parts palette for you to use.

Group parts with similar behavior in the same category. By looking at the parts palette you can see how we grouped the parts we provided into categories based on their behavior. For example, all parts used for data entry are in one category, all parts that contain and display lists are in another category, and so forth.

Preparing icons for the parts palette

Each category and part on the Visual Builder parts palette is represented by a bitmap so you can recognize it visually. With Visual Builder, you can create and use your own bitmaps when you extend the parts palette. If you do not, you can still extend the parts palette and accept the default category Icon, , and the default part Icon, .

Preparing a resource DLL (OS/2 version only)

This example uses a **Miscellaneous** category and a **MainWindow** part, which are stored in a sample.dll file as resources numbers 800 and 802, respectively.

To prepare bitmaps for use with Visual Builder, do the following:

1. Create your icons. One way to do this is to use the OS/2 icon editor, which is available in the operating system toolkit.

   Bitmaps used on the parts palette must be no larger than standard icons for the display resolution being used. For VGA displays on OS/2, use the Independent VGA form (32x32). For higher display resolutions on OS/2, use the 8514-16 colors form (40x40).

2. Create a resource DLL that contains your icons. Use files similar to the following:

   - sample.cbl

     any COBOL program

   - sample.rc

     ```
     icon 800 Miscellaneous.ico
     icon 801 MainWindow.ico
     ```

   - sample.def

     ```
     library sample
     description 'Icons for user-extended palette'
     ```

   - sample.mak

     ```
     sample.dll: sample.obj sample.def sample.res
       cob2 sample.obj sample.def /dll
         rc sample.res sample.dll

     sample.obj: sample.cbl
       cob2 -c sample.cbl

     sample.res: sample.rc
       rc -r sample.rc
     ```

Once you have the files ready, type the following in a command window to build the resource DLL:

```
nmake sample.mak
```

3. Place the resource DLL in a directory in your LIBPATH statement.

Your icons are now ready for use with Visual Builder.

Preparing a resource DLL (Windows version only)

This example uses a **Miscellaneous** category and a **MainWindow** part, which are stored in a sample.dll file as resources numbers 800 and 802, respectively.

To prepare bitmaps for use with Visual Builder, do the following:

1. Create your icons using an icon editor.
2. Create a resource DLL that contains your icons. Use files similar to the following:
   - sample.cbl

     any COBOL program
   - sample.rc

800 icon	Miscellaneous.ico
802 icon	MainWindow.ico
   - sample.mak

     ```
     sample.dll: sample.obj sample.exp sample.res
        cob2 /dll sample.obj sample.exp sample.res

     sample.exp: sample.obj
        echo LIBRARY sample > sample.def
        echo EXPORTS >> sample.def
        CPPFILT -Q -B -P sample.obj >> sample.def
        ilib /q /def:sample.def /gi:sample.lib

     sample.obj: sample.cbl
        cob2 -c sample.cbl

     sample.res: sample.rc
        irc sample.rc
     ```

   Once you have the files ready, type the following in a command window to build the resource DLL:

   ```
   nmake sample.mak
   ```

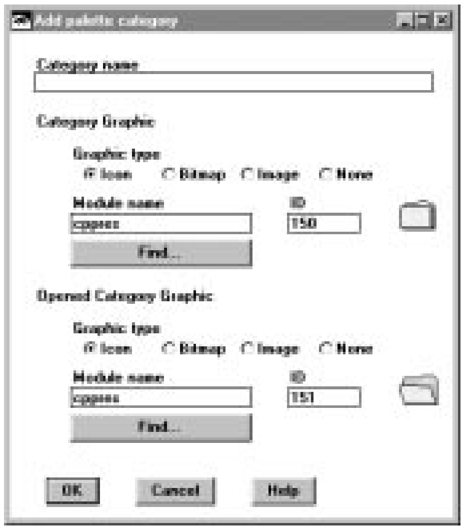
3. Place the resource DLL in a directory in your PATH statement.

Your icons are now ready for use with Visual Builder.

Adding a category to the parts palette

Once you have prepared an icon in a resource dynamic link library (.DLL), you are ready to extend the parts palette. To add a category to the parts palette, do the following:

1. In the Composition Editor, select **Modify palette→Add new category** from the **Options** pull-down menu. The Add Palette Category window is displayed as follows:



Category name:

Category Graphic

Graphic type: [✓ Icon] Bitmap Image None

Module name: cppres

ID: 150

Opened Category Graphic

Graphic type: [✓ Icon] Bitmap Image None

Module name: cppres

ID: 151

Figure 50. The Add Palette Category window

   Notice that the default category icon, , is specified. It is stored as resource ID `150` in the iwzbv33r.dll resource file provided with Visual Builder.

2. Enter the name you want for your category in the **Category name** field.
3. Enter `iwzbv33r` or the name of your resource dynamic link library (.DLL) in the **Module name** field.

   Note: Do not type the .DLL file extension in the **Module name** field.

4. Type `150` or the resource ID of the icon in your resource dynamic link library (.DLL) in the **Resource ID** field.

   After entering the resource ID number, move the cursor to another component in the window, such as the **Module name** field, if you want to see the graphic to be used before continuing.

5. Select the **OK** button.

Your category with the icon specified is added to the parts palette.

Note: If you do not specify a dynamic link library (.DLL), Visual Builder uses the default icon. If you specify a dynamic link library (.DLL) but Visual Builder cannot find it, Visual Builder uses the question mark icon,

.

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

- The dynamic link library (.DLL) exists and is in the current directory or any directory specified in the LIBPATH (for OS/2) or PATH (for Windows) environment variable.
- The dynamic link library (.DLL) file name is correct.
- The resource ID for the icon (in the .RC file) exists in the dynamic link library (.DLL).

Specifying a unique icon for a part

You can specify a unique icon for a part you add to the parts palette, but you must do so before you add it to the parts palette. To give your part a unique icon, do the following:

1. Open the part.
2. Switch to the System Interface Editor.
3. Enter the name of the dynamic link library (.DLL) file containing the icon you want to use in the **DLL Name** field.
4. Enter the resource ID number for the icon in the **Resource ID** field.

   If you enter a valid dynamic link library (.DLL) file name and resource ID number, Visual Builder displays the icon below the **Resource ID** field. This enables you to verify the icon before adding it to the parts palette.

   Note: If you do not specify a dynamic link library (.DLL), Visual Builder uses the default icon. If you specify a dynamic link library (.DLL) but Visual Builder cannot find it, Visual Builder uses the question mark icon,

   .

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

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

5. Select **File→Save** to save the resource dynamic link library (.DLL) and resource ID information in the System Interface Editor.

Adding a part to the parts palette

You can add a part to any category on the parts palette using any of the following methods:

- Adding a part selected in the Visual Builder window
- Adding the part you are currently editing
- Adding any part whose part file (.VCB) is loaded

Adding a part selected in the Visual Builder window

To add a part to the parts palette from the Visual Builder window, do the following:

1. Load the part file (.VCB) containing the part you want to add to the parts palette, if it is not already loaded.
2. Select the part file (.VCB) containing the part you want to add to the parts palette.
3. Select the part you want to add.

   Note: You can add multiple parts by holding down the Ctrl key and clicking on each part you want to add.
4. Select **Part→Add to palette**. Visual Builder displays the Add to Palette window, as shown in Figure 51.

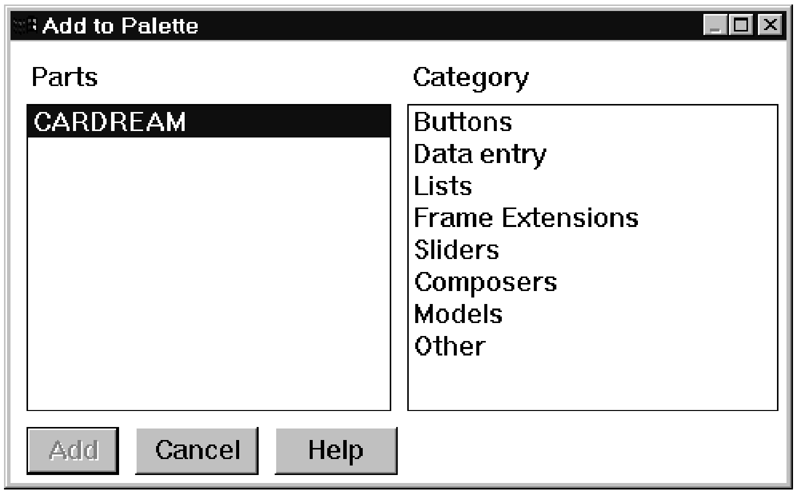


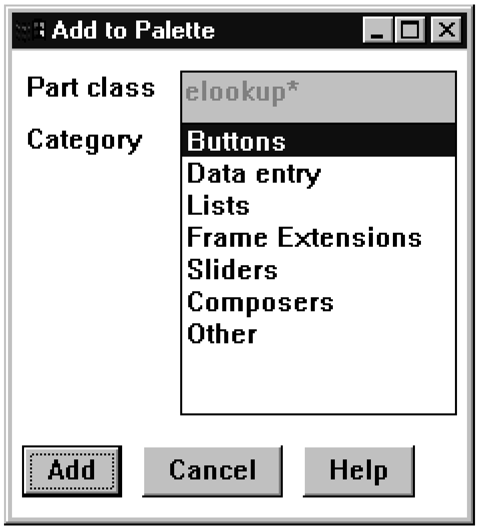
Figure 51. Add to Palette window

5. Select the part you want to add.
6. Select the category you want to add the part to.
7. Select the **Add** button. Visual Builder adds the part to the parts palette in the selected category.

Adding the part you are currently editing

You can add the part you are currently editing to the parts palette from either the Composition Editor, the System Interface Editor, or the Part Interface Editor. To add the part you are currently editing to the parts palette, do the following:

1. Double-click on the name of the part in the Visual Builder window. Visual Builder opens the part in the Part Interface Editor or the Composition Editor.
2. Select **File→Add to palette**. Visual Builder displays the Add to Palette window, as shown in Figure 52.


Add to Palette

Part class: elookup*

Category:
- Buttons
- Data entry
- Lists
- Frame Extensions
- Sliders
- Composers
- Other

Figure 52. The Add to Palette window

The **Part name** field shows the name of the part you are editing. This is the part to be added to the parts palette. You cannot change the name of the part displayed in this field.

3. Select the category you want to add the part to.
4. Select the **Add** button. Visual Builder adds the part to the category you selected on the parts palette. To see this, switch to the Composition Editor and select the category. The icon for the part is displayed in the parts column.

Adding any part whose part file (.VCB) is loaded

You can add any part to the parts palette as long as its part file (.VCB) is loaded in the Visual Builder window. The following steps explain how to do this:

1. In the Composition Editor, select **Modify palette→Add new part** from the **Options** pull-down menu. The Add to Palette window is displayed as shown in Figure 53 on page 150.

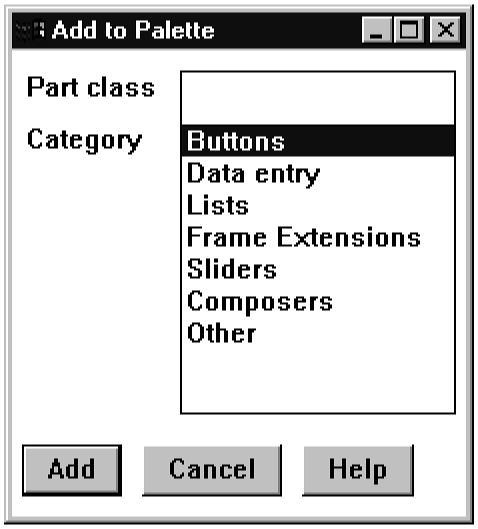


Figure 53. The Add to Palette window

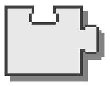
To add a part to the parts palette, do the following:

a. Type in the part name in the **Part class** field or the class name of the part you want to add.

b. Select the name of the category to which you want to add your part.

c. Select the **Add** button.

   Your part is added to the parts palette in the specified category.

Notice the part you just added uses the same icon as the part it inherits from. If you inherit from a part whose part file (.VCB) is not loaded or for which you have not provided a resource dynamic link library (.DLL), Visual Builder uses the default part icon,  .

Deleting a category or part from the parts palette

To delete a part from the parts palette, do the following:

1. Select the part on the parts palette.
2. Select **Modify palette→Delete category** from the **Options** pull-down menu. The selected part is deleted from the parts palette.

To delete a category from the parts palette, do the following:

1. Select the category on the parts palette.
2. Select **Modify palette→Delete category** from the **Options** pull-down menu. The selected category and all of the parts in it are deleted from the parts palette.

Saving parts palette changes

Visual Builder automatically saves all parts palette changes for you. When you create a new category or part, Visual Builder stores information about the category or part in a file named vbpalet.dat, which is stored in your startup directory (or in your target directory, if you are using WorkFrame). This file is written automatically.

Once you add or delete categories or parts, the vbpalet.dat file is read each time you start Visual Builder. The information this file contains causes any categories or parts that you have added to be included on the parts palette. It also prevents any categories or parts that you have deleted from appearing on the parts palette.

If you update the icon associated with a part, the parts palette is updated the next time you select the category in which the icon appears.

Removing a category or part that you just added

The vbpalet.dat file also allows you to undo and redo any changes you make to the parts palette, but only during the current Composition Editor session. For example, after adding a category or part, you can select **Edit→Undo** to remove the part or category you just added. Selecting **Edit→Redo** would put the part or category back on the parts palette, again.

Once you close the Composition Editor, you can no longer undo or redo any changes. However, you can still add categories and add parts, as well as delete categories and parts.

Chapter 9. Learning to use connections

This chapter describes the types of connections you can make and how to make them. Each connection description provides the following information:

- A definition of the connection
- The color of the connection
- Whether the connection is unidirectional or bidirectional
- Whether the connection requires you to supply values to complete it

Attribute-to-attribute connection

An *attribute-to-attribute connection* links two attribute values together. The purpose of this type of connection is to cause the value of one attribute to change when the value of another attribute changes.

An attribute-to-attribute connection uses a bidirectional, dark blue line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source.

Note: In Windows NT, this connection appears with small diamonds at either end.

When your part is instantiated, the target attribute is set to the value of the source attribute. Attribute-to-attribute connections never take parameters.

Note: Do not create attribute-to-attribute connections in which the source attribute is not initialized before the *ready* event is signaled. You may get a system error in the resulting application and the application may fail. See “The *ready* event” on page 64 for more information.

In Figure 54, an attribute of the DBClass1 nonvisual part is connected to an attribute of the entry field. This connection causes the value of the entry field's attribute to change whenever the value of the DBClass1 attribute changes, and vice versa.

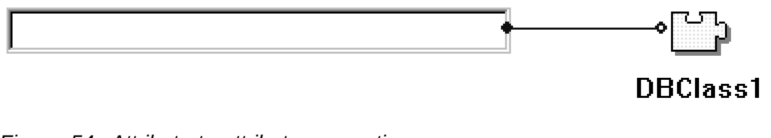


Figure 54. Attribute-to-attribute connection

The effect of attribute completeness on connections

It is important to know the completeness of attributes you are connecting. Otherwise, you might not achieve the results you anticipate.

Figure 55 shows the results of connecting attributes with different behaviors. See “The Attribute page” on page 57 for descriptions of the different kinds of behavior attribute display.

Figure 55. Source and target considerations for attribute types

If the source is a...	**And the target is a full attribute...**	**And the target is a no-set attribute...**	**And the target is a no-event attribute...**
full attribute	All attribute behaviors are available to both the source and target attributes.	Visual Builder automatically reverses the connection.	The target attribute cannot notify the source attribute when the target attribute’s value changes.
no-set attribute	The source attribute initializes the target attribute. The target attribute is updated whenever the source attribute’s value changes.	This is an invalid connection.	The source attribute initializes the target attribute. The target attribute is updated whenever the source attribute’s value changes, but the target attribute cannot notify the source attribute when the target attribute’s value changes.
no-event attribute	The source attribute initializes the target attribute; no event notification occurs.	Visual Builder automatically reverses the connection.	The source attribute initializes the target attribute; no event notification occurs.

Event-to-attribute connection

An *event-to-attribute connection* enables the occurrence of the source event to trigger a change in the value of the target attribute. To accomplish this, the connection calls the attribute’s set method whenever the event occurs. If the attribute is a no-set attribute, you cannot make the connection. If you open settings on a connection of this type, the target of the connection appears to be an action with the same name as the target attribute.

An event-to-attribute connection uses a unidirectional dark green arrow with the arrow head pointing to the target. If the attribute's set method requires you to supply parameter values, the connection line is initally dashed. This indicates the connection is incomplete. You must provide values for the parameters to turn the connection line solid. A solid line indicates the connection is complete. If the event contains event data, the attribute's set method uses the event data and the connection line is solid.

You can supply the missing parameter value or override any event data present by connecting the parameter to an attribute or action, or by supplying a constant parameter value. See “Supplying parameter data for incomplete connections” on page 163 for more information.

In Figure 56, the *press* feature of the **Refresh** push button is connected to the *contents* attribute of the entry field. Note the line is dashed, meaning a parameter is needed to complete the connection. If you connect the *contents* attribute of the connection with a feature of type VarLengthString of some other part, the connection is completed and the line becomes solid.

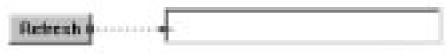


Figure 56. Event-to-attribute connection

Event-to-action connection

An *event-to-action connection* causes an action to start whenever the source event occurs.

An event-to-action connection uses a unidirectional, dark green arrow with the arrow head pointing to the target. If the action requires you to supply parameter values, the connection line is initally dashed. This indicates the connection is incomplete. You must provide values for the parameters to turn the connection line solid. A solid line indicates the connection is complete.

You can supply the missing parameter value or override any event data present by connecting the parameter to an attribute, action, or by supplying a constant parameter value. See “Supplying parameter data for incomplete connections” on page 163 for more information.

Also, the action targeted by this connection can have a return parameter. If it does, you can treat the return parameter as a no-set attribute of the connection and use it as the source of another connection. The return parameter appears in the connection menu for the connection as *actionResult*.

In Figure 57, the *press* feature of the **Add** push button is connected to the *addItemEnd* action of the multiline edit control. Note the line is dashed, meaning a parameter is needed to complete the connection. Once you supply a parameter, the line becomes solid.

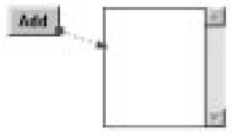


Figure 57. Event-to-action connection

Attribute-to-action connection

An *attribute-to-action* connection causes an action to start whenever the event identification that is associated with the attribute is triggered. This connection is similar to an event-to-action connection because the connection invokes the action's method whenever the attribute's event is triggered, and if the method has parameters, passes the attribute data as the first parameter.

An attribute-to-action connection uses a unidirectional, dark green arrow with the arrow head pointing to the target. If the action requires more than one input parameter, the connection line initially appears dashed to show it is incomplete.

The attribute's data is passed as the first parameter of the action if no parameter is explicitly specified. You can supply any other missing parameters or override the attribute data present by connecting the parameter to an attribute or action, or by supplying a constant parameter value. See "Supplying parameter data for incomplete connections" on page 163 for more information.

The action targeted by this connection can have a return parameter. If it does, you can treat the return parameter as a no-set attribute of the connection and use it as the source of another connection. The return parameter appears in the connection menu for the connection as *actionResult*.

In Figure 58, the *contents* attribute of the entry field is connected to the *addItemEnd* action of the list box.

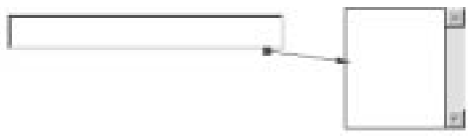


Figure 58. Attribute-to-action connection

Parameter connections

A parameter connection supplies a parameter to an action by passing either an attribute's data or the return data from an action. This connection looks similar to an attribute-to-attribute connection; it uses a bidirectional line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source. The difference you see on your screen is parameter connections are violet instead of dark blue.

In addition, the parameter names are included in the part's pop-up menu. Therefore, if you are in doubt about a connection you want to make, you can browse a part's features to see the parameter names.

The parameter is always the source of the connection because the parameter cannot store any parameters. If you connect an attribute or action to a parameter, Visual Builder reverses the direction of the connection to make the parameter the source.

Whenever the parameter needs data, code generation supplies one, as follows:

- If the parameter is connected to an attribute, the connection calls the attribute's get method to get the attribute's data and either moves the result to the parameter or sets the parameter to the result, depending on the parameter's type.
- If the parameter is connected to an action with a result, the connection code calls the action and either moves the result to the parameter or sets the parameter to the result, depending on the parameter's type.
- You can supply a constant parameter value as a setting within the connection. See "Supplying parameter data for incomplete connections" on page 163 for more information.

If you connect a parameter to two different attributes, the first attribute you connect the parameter to has precedence over the second. You can change this, if necessary, by reordering the connections or deleting one of the connections.

Visual Builder uses a dashed line to give you a visual cue when a parameter connection is needed. For example, if you connect an event to an action requiring parameter values, the connection line between the event and the action may be dashed. This is always the case when the event has no data. See "Supplying parameter data for incomplete connections" on page 163 for more information.

In the Visual Builder you use types to ensure data declarations are consistent and correct linkages are used in code generation. Make sure the source and target agree with each other and comply with COBOL MOVE rules. To view the types available, do one of the following ways:

- Select **Options→Show type list** in the Visual Builder window. As you select visual or nonvisual parts, the types available in the selected parts appear in the **Loaded Type Information** box.
- You can export the interface of a part and view the type declarations of the attributes. To export the interface of a part, open the pop-up menu for the part in the Visual Builder window and select **Export interface**. After you export the interface in part information file, you can use your favorite text editor to view the contents of the part information file.

  You can add your own types through part information files (.VCE). Refer to Appendix A, "Creating part information files" on page 223 for more information on creating part information files.

If you need to make changes to a connection, you can open the settings window for a connection to change the source, target, or direction of a connection. Refer to "Changing settings for a connection" on page 166 for details on making changes to a connection.

Connection type summary

The following table summarizes the types of connections that Visual Builder provides:

If you want to...	Use this connection type	Color	Arrows	Return value allowed?
Change one data value when another changes	attribute-to-attribute	Dark blue	●——●	No
Change a data value whenever an event occurs	event-to-attribute	Dark green	- - - -► OR ——►	No
Call an action whenever an event occurs	event-to-action	Dark green	- - - -► OR ——►	Yes[2]
Call an action whenever a data value changes	attribute-to-action	Dark green	- - - -► OR ——►	Yes[2]
Supply data to a parameter	parameter	Violet	●——●	No

Making the connections

In this section, you learn how to make attribute-to-attribute, event-to-attribute, event-to-action, and attribute-to-action connections.

Determining the source and target

A connection is directional; it has a source and a target. The direction in which you draw the connection determines the source and target. The part on which the connection begins is the *source* and the part on which it ends is the *target*.

When you make an event connection, Visual Builder draws an arrow on the connection line between the two parts. The arrow points from the source to the target. If information can pass through the connection in both directions, as it can in an attribute-to-attribute connection, a hollow circle (OS/2) or hollow diamond (Windows) indicates the source and a solid circle (OS/2) or solid diamond (Windows) indicates the target.

Often, it does not matter which part you choose as the source or target, but there are connections where direction is important.

- With an attribute-to-action, event-to-action, or event-to-attribute connection, the event is always the source and the action or attribute is always the target. In the case of an attribute-to-action connection, the source event is signaled when the

[2] The return value is supplied by the connection's *actionResult* attribute.

attribute changes value. If you try to make an action-to-attribute, action-to-event, or attribute-to-event connection, Visual Builder automatically reverses it for you.

- For attribute-to-attribute connections, if only one of the attributes has a set method, Visual Builder makes that attribute the target. This is done so the attribute with the set method can be initialized when the application starts.
- When you make attribute-to-attribute connections, the order in which you choose the source and target is important. The source and target attribute data are probably different when your part is first initialized. If they are, Visual Builder resolves the difference by changing the value of the target attribute to match that of the source attribute. Thereafter, if both attributes have set methods, the connection updates either attribute if the other changes.

Browsing a part’s features

Sometimes it is useful to browse a part’s features before using them in a connection. For example, you might want to look at an attribute to see if it has a set method so it can update itself when it receives new data from another attribute.

By using **Browse part features**, you can see all of a part’s features in one window and browse, but not change, the information about each feature. To modify a feature, use the Part Interface Editor. To modify a feature of a nonvisual data part, change the copy file and reimport it. If you do change the copy file and reimport it, you must regenerate and rebuild all the part using the new nonvisual data part.

There is an important distinction between browsing a part’s features and displaying its features for making a connection. When you browse a part’s features, you see all of its features, even if some of them are not available for connections. (This includes inherited features if the base parts are loaded into Visual Builder.) When you display a part’s connection menu, however, you see only those features available for connections.

To browse the features of a part, do the following:

1. Move the mouse pointer over the part and click mouse button 2. Visual Builder displays the part’s pop-up menu.
2. Select **Browse part features**.

   Visual Builder displays a browse window containing three columns: one for actions, one for attributes, and one for events. For example, Figure 59 on page 160 shows the browse window Visual Builder displays for browsing the features of a push button:

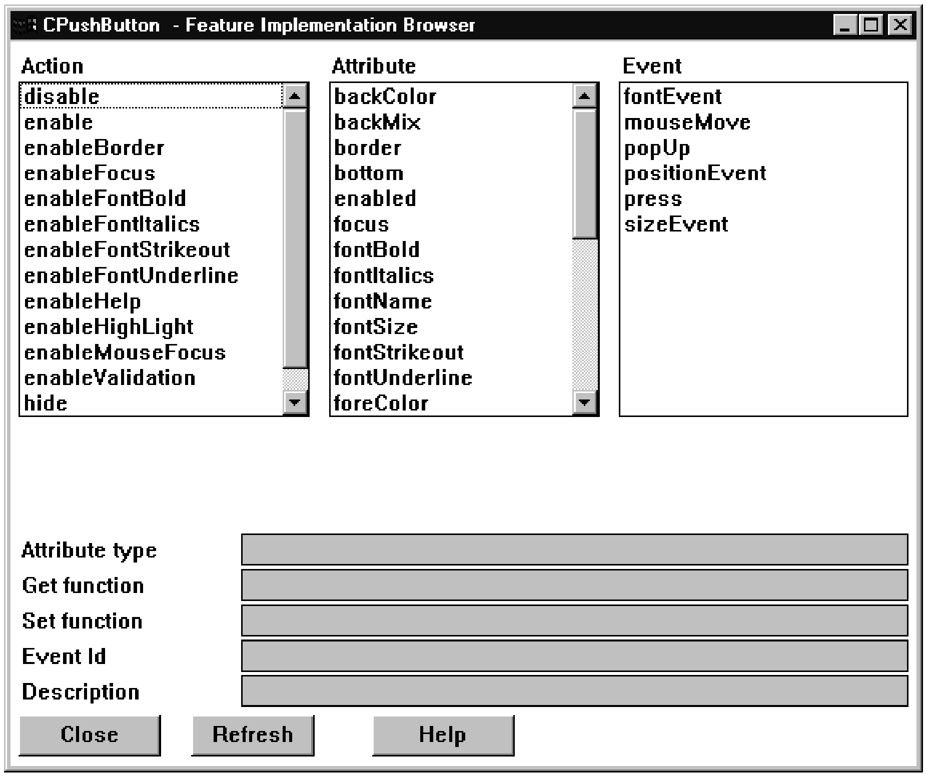
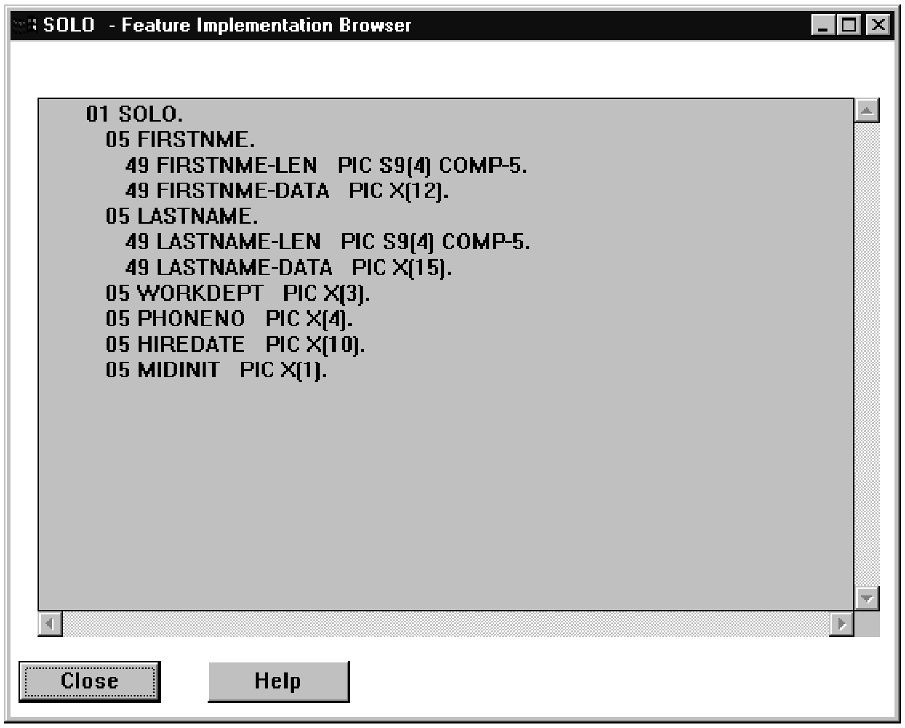


Figure 59. Browse Part Features window

Note: When you select **Browse part features** for a window similar to Figure 60 on page 161 appears.



```
01 SOLO.
  05 FIRSTNME.
    49 FIRSTNME-LEN   PIC S9(4) COMP-5.
    49 FIRSTNME-DATA   PIC X(12).
  05 LASTNAME.
    49 LASTNAME-LEN   PIC S9(4) COMP-5.
    49 LASTNAME-DATA   PIC X(15).
  05 WORKDEPT   PIC X(3).
  05 PHONENO   PIC X(4).
  05 HIREDATE   PIC X(10).
  05 MIDINIT   PIC X(1).
```

Figure 60. Feature Implementation Browser window to browse the part features for a data part

3. Select the feature you want to browse.

   Visual Builder displays information about the feature you select in the entry fields below the feature columns. Different sets of entry fields are displayed depending on whether you select an action, an attribute, or an event.

   The information Visual Builder displays when you browse a part's features is the same as the information you would see in the Part Interface Editor. See “The Part Interface Editor” on page 56 to learn about the information Visual Builder displays for features.

4. To read more information about the features listed, select the **Help** button.
5. When you have finished browsing the features, select the **Close** button to close the browse window.

Connecting features to features

Follow these steps to connect features:

1. Position the mouse pointer over the *source*, the part or connection you want to connect from, click mouse button 2, and select **Connect** from its pop-up menu.

   A menu appears showing the names of the most commonly used attributes, actions, and events, called the *preferred features*. If the source is a part, there is usually a **More** selection at the bottom of the list.

   If the **More** selection is not there, this means the list contains all of the available features, not just the preferred ones, and there are no more features.

2. Do one of the following:

   - If the feature you want appears in the list, select it.
   - If the feature you want does not appear in the list, but the **More** selection is available, select **More** and then select the feature you want from the complete list of features.
   - If the feature you want does not appear in either the preferred list or the expanded list displayed when you select **More**, you can edit the part to add the feature you need. For more information about this, see "Editing parts placed on the free-form surface" on page 138.

     Note: You can not edit parts in VAccess.vcb. To add features to a Visual Builder-supplied part, first create your own part by placing a Visual Builder-supplied part on a canvas. Then add the features you want or promote the features you want to access.

     If, at this point, you decide not to complete the connection, do one of the following:

     – If a pop-up menu is displayed, move the mouse pointer away from the connection menu and click mouse button 1.

     – If a window showing all of the features is displayed, select the **Cancel** button at the bottom of the window.

     The menu or window disappears and the connection is not completed.

3. Position the mouse pointer over the part or connection you want to connect *to*, called the target.

   While moving the mouse, notice a dashed line trails from the mouse pointer to the source of the connection.

4. Click mouse button 1 and a pop-up menu appears, again showing a list of features.

5. Select a name from the pop-up menu or from the **More** list. The same instructions regarding the presence of **More** apply as described previously.

A colored connection line appears when both ends of the connection have been made. The color indicates the connection's type, based on the selections you made in the pop-up menu. See "Connection type summary" on page 158 for a table showing the colors used for each connection type.

If the line is dashed, it requires parameters, as described in the next section.

Supplying parameter data for incomplete connections

Event-to-action, attribute-to-action, and event-to-attribute connections sometimes require parameters, or input arguments. If a connection requires parameters not yet specified, it appears as a dashed arrow indicating it is incomplete. When you have made all the necessary parameter connections, the connection line becomes solid indicating the connection is complete.

Note: Do not make parameter-to-parameter connections. Visual Builder does not prevent you from doing this, but the generated COBOL code for your application may not be valid.

The following sections describe how to complete connections when input parameters are required.

Supplying parameter data using a connection

One way to supply parameters is to make connections from the dashed connection lines to the parts supplying the data to the parameters. Most of the time, the data you need are those of attributes of other parts you are working with in the Composition Editor. Sometimes, however, the data you need is the return data from an action.

To supply a parameter, do the following:

1. Start a new connection using the dashed connection line requiring the parameter as the source.
2. For the target, select the attribute or *actionResult* feature from an action to provide the data the parameter needs.

When you make a connection, Visual Builder provides a visual cue to help you position the pointer correctly. When you have the pointer directly over the connection line, a small hollow box appears.

Figure 61 on page 164 shows an incomplete event-to-action connection. When a user selects the **Add** button, its *press* feature notifies the *addItemEnd* action of the CListBox part to add something to the list box as the last item in the list. The connection is incomplete because the *addItemEnd* action has a parameter that needs input data, which is the text to add to the list box.

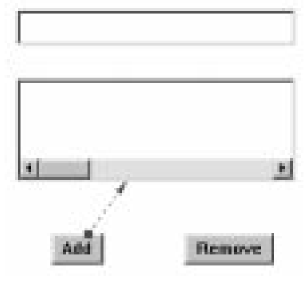


Figure 61. Incomplete connection due to missing parameter value

Figure 62 shows how a parameter connection, in which an attribute of the CEntryField part is used to supply the input data for the parameter, completes the event-to-action connection shown in Figure 61. The *contents* parameter of the connection shown in Figure 61 is connected to the *contents* attribute of the CEntryField part. Therefore, when the **Add** button is selected, its *press* feature notifies the *addItemEnd* action of the CListBox part to add the contents in the CEntryField part as the last item in the list box.

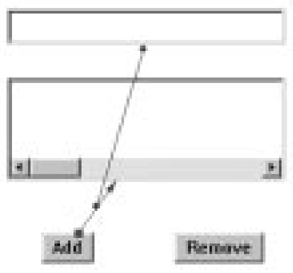


Figure 62. Completing a connection using an attribute as the input data

The return parameter, if any, of an action displays as the *actionResult* attribute of the connection. For example, you can connect the *actionResult* attribute to an attribute of the same part or another part.

Note: Make sure the target of the parameter connections has the same type as the parameter. Otherwise, the generated code may be invalid.

Supplying parameter data using a literal

When connections need parameters and you want to provide these parameters as a literal, use the settings window of the incomplete connection, as follows:

1. Select **Open settings** from a connection's pop-up menu.

A quicker way to open the settings window is to double-click on the connection line.

The settings window of the connection is displayed.

2. Select the **Set parameters** button. The **Constant Parameter Data Settings** window is displayed showing the parameters.
3. Enter the literal you want to use.

   Enter the literal just as you would if you were coding it by hand. Visual Builder places the literal in a MOVE statement to the parameter. For example, to provide a value for a *contents* parameter, enter the text string (in quotes) you want the parameter to receive.
4. Do one of the following:
   - Select the **OK** button to apply the values and save them.
   - Select the **Apply** button if you want to see what effect these values have before saving them.
   - Select the **Cancel** button to remove the notebook without saving any of the parameter values you entered.

   You can select **Help** for additional information about entering parameter values.

Note: You should only provide literals for parameters whose type represents an elementary data item or is VarLengthString. Visual Builder does not prevent you from entering values for any data item, but COBOL MOVE rules determine the result in generated code.

Preventing missing or invalid parameters

When an action connection requires arguments, be sure you make the correct number of parameter connections. Also, be sure you make the parameter connections before you generate code for your part. If you use the return parameter of one connection as input to another, make sure the connections appear in the correct order. See “Manipulating connections” to learn how to reorder connections.

Manipulating connections

Once a connection is made, you can manipulate it by doing the following:

- Changing settings for a connection
- Reordering connections
- Deleting connections
- Showing and hiding connections
- Rearranging connections

Changing settings for a connection

The settings window of a connection provides a way for you to select different features as the source and target of the connection. If a method is the target of the connection, this window enables you to specify or select a different method as the target.

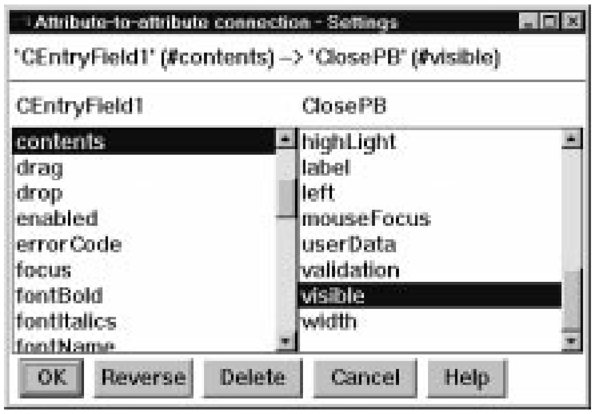
To open the settings window for a connection, move the mouse pointer over the connection and do one of the following:

- Double-click mouse button 1.
- Click mouse button 2 and select **Open settings** from the connection's pop-up menu.

Visual Builder displays different connection settings windows depending on whether the target of the connection is an attribute or action. The following sections describe these two windows.

Changing settings for attribute-to-attribute connections

The following figure shows the window Visual Builder displays when you open the settings window for an attribute-to-attribute connection:



Attribute-to-attribute connection - Settings

'CEntryField1' (#contents) -> 'ClosePB' (#visible)

CEntryField1	ClosePB
contents	highLight
drag	label
drop	left
enabled	mouseFocus
errorCode	userData
focus	validation
fontBold	visible
fontItalics	width
fontName	

Figure 63. Attribute-to-attribute connection settings window

The connection settings window for an attribute-to-attribute connection contains two columns of attributes. The left column contains the attributes belonging to the source part. The right column contains the attributes belonging to the target part, excluding any no-set attribute attributes.

To change the attribute to be used as either the source or target of the connection, select an attribute from the list.

This connection settings window has the following buttons:

OK
Removes the connection settings window and puts any changes made into effect.

Reverse
Reverses the order of the connection; the source part becomes the target and the target part becomes the source.

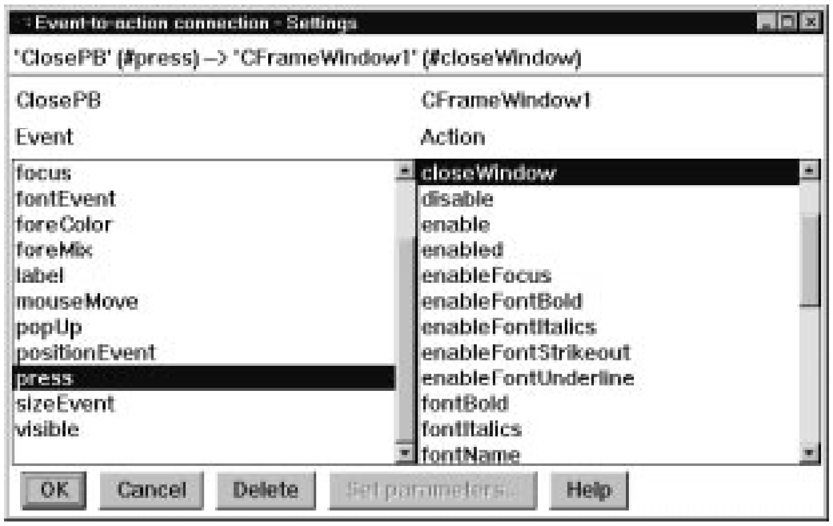
Delete
Deletes the connection.

Cancel
Removes the connection settings window without putting any changes into effect.

Help
Provides information about the window.

Changing settings when an action is the target

When you open a settings window for a connection whose target is an action, Visual Builder displays the following window:


Event-to-action connection - Settings

'ClosePB' (#press) --> 'CFrameWindow1' (#closeWindow)

ClosePB	CFrameWindow1
Event	Action
focus	closeWindow
fontEvent	disable
foreColor	enable
foreMix	enabled
label	enableFocus
mouseMove	enableFontBold
popUp	enableFontItalics
positionEvent	enableFontStrikeout
press	enableFontUnderline
sizeEvent	fontBold
visible	fontItalics
	fontName

Figure 64. Connection settings window with an action as the target

The connection settings window contains two columns of features. The features in the left column belong to the source part; these features are the same type of feature as the one currently selected for the source part. For example, if the feature selected for the source part is an event, this column contains a list of the events belonging to the source part, including events associated with attributes of the source part.

Likewise, the features in the right column belong to the target part; these features are all actions or attributes with set methods.

The top line above the lists contains the features currently selected. To change the feature to be used as either the source or target of a connection, select a feature from the list.

This connection settings window has the following buttons:

OK
Removes the connection settings window and puts any changes made into effect.

Cancel
Removes the connection settings window without putting any changes into effect.

Delete
Deletes the connection.

Set parameters
Opens another window in which you can specify data for parameters of the action.

Help
Provides information about the window.

Reordering connections

If you make several connections from the same part, they run in the order in which you made the connections. To ensure the correct flow of control when you generate the source code, you may need to reorder the connections. If so, do the following:

1. Select the source part.
2. From the source part's pop-up menu, select **Reorder connections from**.

   Visual Builder displays the **Reorder Connections** window showing a list of your connections.
3. With the mouse pointer over the connection you want to reorder, press and hold as follows:
   - mouse button 1 in Windows.
   - mouse button 2 in OS/2.
4. Drag the connection to the place in the list where you want the connection to occur.
5. Release the mouse button.
6. Repeat these steps until the connections are listed in the order in which you want them to occur.
7. Close the window.

Deleting connections

You can delete a connection in either of the following ways:

- From the connection's pop-up menu.

  Note: You do not have to select a connection to delete it using this method.

  To delete a connection from its pop-up menu, do the following:

  1. Click on the connection with mouse button 2 to display its pop-up menu.
  2. Select the **Delete** button.
- From the connection's settings window

To delete a connection from its settings window, do the following:

1. Open the settings window for the connection by doing one of the following:
   - Double-clicking on the connection
   - Clicking on the connection with mouse button 2 to display its pop-up menu and selecting **Open settings**
2. Select the **Delete** button.

Showing and hiding connections

You can show and hide connections by using , the **Show Connections** tool, and , the **Hide Connections** tool on the Tool bar. These tools show or hide all connections the selected part or parts have as their source or target.

If you hide connections, the Composition Editor free-form surface is less cluttered, making it easier for you to work.

If no parts are selected, these tools show and hide all of the connections on the free-form surface. If at least one part is selected, these tools show and hide the connections from or to the selected part(s).

Another way to show and hide connections is to move the mouse pointer over a part, click mouse button 2, and select the **Browse connections** choice from the part's pop-up menu, which displays a cascaded menu. The choices in the menu affect only connections going to and from the part the mouse pointer was over when you displayed the pop-up menu.

The Browse connections cascaded menu contains the following choices:

Show to
Shows all connections for which the part is the target.

Show from
Shows all connections for which the part is the source.

Show to/from
Shows all connections for which the part is either the source or the target.

Show all
Shows all connections that have been made, regardless of where the mouse pointer is when you click mouse button 2.

Hide to
Hides all connections for which the part is the target.

Hide from
Hides all connections for which the part is the source.

Hide to/from
Hides all connections for which the part is either the source or the target.

Hide all
Hides all connections that have been made, regardless of where the mouse pointer is when you click mouse button 2.

Rearranging connections

You can rearrange a connection by doing the following:

- Selecting connections
- Deselecting connections
- Changing the source and target of connections

Selecting connections

You select connections in the same way that you select parts. When you select a connection, three boxes called selection handles appear on it to show it is selected: one at each end and one in the middle. You can use these boxes to change either of the following:

- The end points of the connection, as described in "Changing the source and target of conections" on page 171.
- The shape of the connection line by dragging the middle box to another location. This helps you distinguish among several connection lines located close together.

Selecting a single connection

1. Move the mouse pointer over the connection you want to select.
2. Click mouse button 1 to select the connection. The connection is selected.

Selecting multiple connections in OS/2

If you want to select several connections, do one of the following:

- To select multiple connections using just the mouse, do the following:
    1. Move the mouse pointer over one of the connections you want to select.
    2. Hold down mouse button 1 instead of clicking it.
    3. Move the mouse pointer over each connection you want to select.

       The selection boxes appear on each connection the mouse pointer passes over to show they are selected.
    4. After the connections are selected, release mouse button 1.
- To select multiple connections using both the mouse and the keyboard, do the following:
    1. Hold down the Ctrl key.

2. Move the mouse pointer over a connection.
3. Click mouse button 1 while the mouse pointer is over the connection line.
4. Without releasing the Ctrl key, repeat the preceding steps until all connections you want to select are selected.

Selecting multiple connections in Windows

To select several connections, do the following:

1. Hold down the Shift key.
2. Move the mouse pointer over a connection.
3. Click mouse button 1 while the mouse pointer is over the connection line.
4. Without releasing the Shift key, repeat the preceding steps until all connections you want to select are selected.

Deselecting connections

If you want to deselect a connection without selecting another part or connection, do the following:

1. Move the mouse pointer over the connection line.
2. Hold down one of the following keys:
   - In OS/2, hold down the Ctrl key.
   - In Windows, hold down the Shift key.
3. Click mouse button 1.

Changing the source and target of conections

Visual Builder gives you the ability to change what a connection is pointing to (the target) or pointing from (the source). Of course, you could always just delete the connection and create a new one. However, the following steps show you a quicker way to do this.

Moving either end of a connection

1. Select the connection.
2. Move the mouse pointer over the filled square appearing on the ends of the connection.
3. Press and hold mouse button 2.
4. Move the mouse pointer to the new part or connection.
5. Release the mouse button.

Depending on the connection type, you may get a pop-up menu asking you for new information for the connection.

What you can change

You can change the source end of any connection. In most cases, you can also change the target end. However, depending on the feature you connect to when you make the change, you might get a different type of connection than the one you started with. If you change the target part of a *feature*-to-action connection to a part that does not support the target action, the connection menu appears, and you can select a new action.

Chapter 10. Adding menus to Visual Builder applications

Menus are a common navigation tool in GUI applications. You can create menu bars and pop-up menus using Visual Builder. This chapter describes the types of menus you can construct and guides you through the process of adding menu bars and pop-up menus to a window. To illustrate the process, you will construct a window with a menu bar and a pop-up menu. Figure 65 shows the completed window.

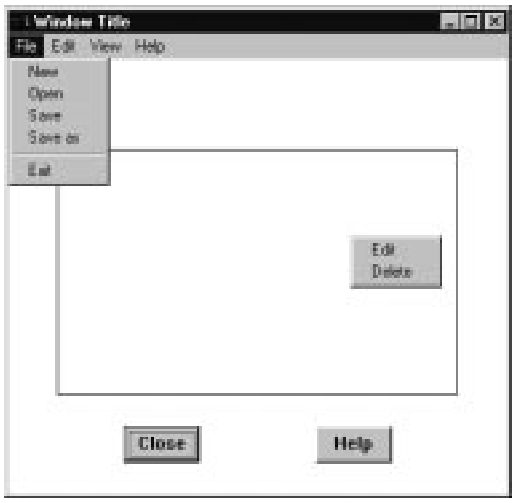


Figure 65. Window with menu bar and pop-up menu

Types of menus and menu items

In Visual Builder, you use the same set of menu parts to build several different menu types:

Menu bars
A menu part attached to a window. It appears horizontally under the window's title bar.

Pop-up menus
A menu part connected to the *popUp* event for a part, such as an entry field. It appears vertically when the user selects the part and presses mouse button 2.

Cascaded menus
A menu part attached to another menu part, as follows:

- If the cascaded menu part is attached to a menu bar, the cascaded menu appears below the menu bar as a pull-down menu.
- If the cascaded menu part is attached to a pop-up menu, the cascaded menu appears beside the pop-up menu.

The Frame Extensions category

The Frame Extensions category contains parts used to create menus. The following parts make up this category:

CMenu

CMenuItem

CMenuCascade

CMenuSeparator

The CMenu part is the fundamental part in this category. This part serves as the basis for all menu types. CMenu parts can contain CMenuCascade parts and CMenuItem parts. Connections made to the CMenu part define whether the menu will be a menu bar or a pop-up menu. The examples in this chapter describe those defining connections.

Creating a menu bar

To begin constructing the window in Figure 65 on page 173, create a new COBOL Visual Builder project, as described in Chapter 5, “Starting Visual Builder” on page 79. Use the following settings for the project:

Project Title
Menu project

Source file directory
MENUPROJ

Project File name
MENUPROJ

Project Target name
MENUPROJ

Once the project is created, the **WorkFrame V3.5 Project - Menu project** window appears. Double-click on the MENUPROJ.VCB part file to start Visual Builder, load the part file, and open the Composition Editor. Creating any of the three menu types requires the same first step: adding a CMenu part. The connections you make to the CMenu part defines which type of menu it becomes.

Adding the CMenu part

The first step in creating a menu bar is to add the CMenu part:

1. Select , the Frame Extensions category, from the left side of the parts palette.
2. Select , the CMenu part, from the right side of the parts palette and drop it on the free-form surface next to the window.

Adding the CMenuCascade parts

In this example, the **File** menu bar choice has some menu items. Follow these steps to create menu items on the **File** menu cascade:

1. Select , the CMenu part, from the right side of the parts palette and drop it on the free-form surface next to the first CMenu part.
2. Select , the CMenuItem part, from the right side of the parts palette.
3. Select **Sticky** from the bottom of the parts palette.
4. Move the mouse pointer over the new CMenu part and drop five CMenuItem parts on the CMenu part.
5. Select , the **Selection Tool**, to unload the mouse pointer.
6. Edit the text of the menu cascade parts. To do this, select each part and press Alt+mouse button 1. The default text is highlighted and you can enter a new name. When you are finished typing, press Shift+Enter. The text of the part is changed. Name the top-most menu cascade part **New**, the next one down **Open**, the third **Save**, the fourth **Save as**, and the last one **Exit**.

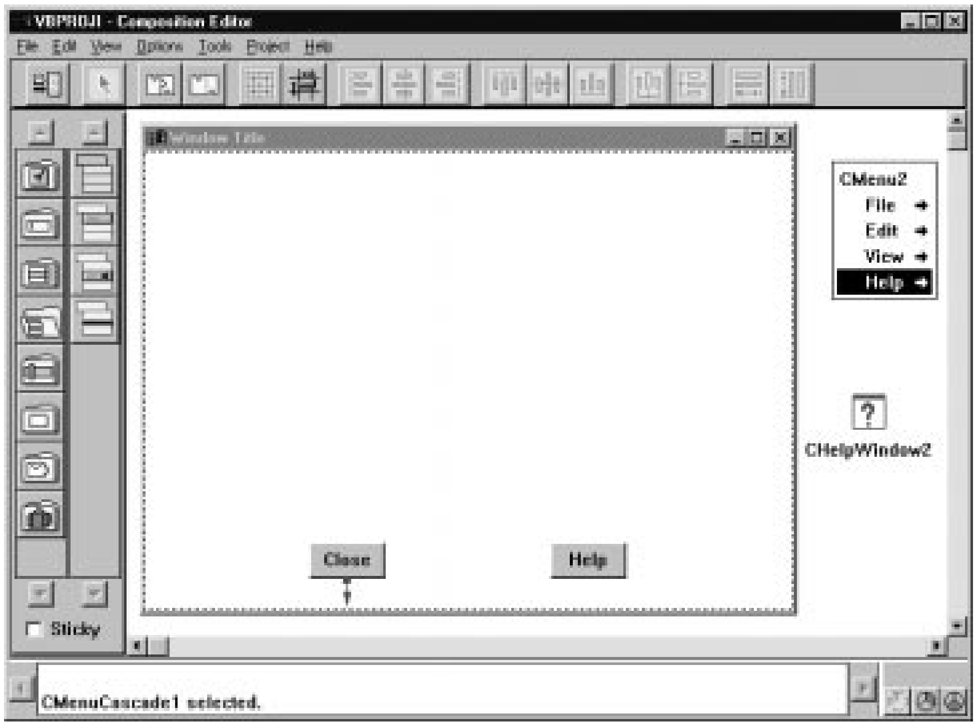
In our example, we have four menu bar choices: **File**, **Edit**, **View**, and **Help**. To add the four menu bar choices, do the following:

1. Select , the CMenuCascade part, from the right side of the parts palette. The CMenuCascade part is used because all four menu bar choices contain additional menu items. If you want to connect a menu bar choice directly to an action, use a CMenuItem part.

2. Select **Sticky** from the bottom of the parts palette.
3. Move the mouse pointer over the CMenu part and drop four CMenuCascade parts on the CMenu part.
4. Select , the **Selection Tool**, to unload the mouse pointer.
5. Edit the text of the menu cascade parts. To do this, select each part and press Alt+mouse button 1. The default text is highlighted and you can enter a new name. When you are finished typing, press Shift+Enter. The text of the part is changed. Name the top-most menu cascade part **File**, the next one down **Edit**, the third **View**, and the last one **Help**.

   Note: The order in which you drop in parts is important. The CMenuCascade or CMenuItem parts at the top appear on the left most side of the menu bar. In our example, **File** appears on the left-most side of the menu bar, **Edit** appears second from the left, **View** appears third from the left, and **Help** appears last.

Your part should look similar to Figure 66.


VBPROJ1 - Composition Editor
File Edit View Options Tools Project Help
Window Title
CMenu2
File
Edit
View
Help
CHelpWindow2
Close
Help
Sticky

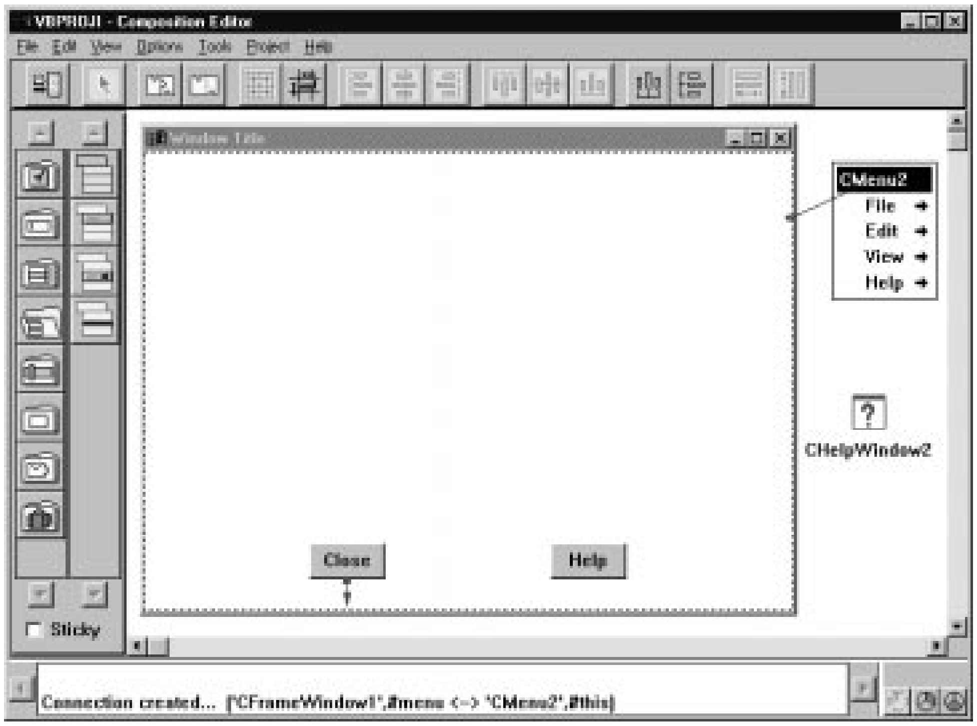
CMenuCascade1 selected.

Figure 66. Composition Editor window showing menu and menu cascade parts

Defining the CMenu part as a menu bar

To make the menu part a menu bar:

1. Connect the *this* attribute of the CMenu part to the *menu* attribute of the frame window part. Although the menu continues to appear vertically on the free-form surface, this connection defines the menu part as a menu bar. Your part should look similar to what is shown in Figure 67.


VBPROJ1 - Composition Editor
File Edit View Options Tools Project Help
Window Title
CMenu2
File
Edit
View
Help
CHelpWindow2
Close
Help
Sticky

Connection created... ('CFrameWindow1',#menu <-> 'CMenu2',#this)

Figure 67. Composition Editor window showing menu part connected to frame window

Because the menu bar is shown outside the frame window, be sure to leave enough space for it below the frame window title. Otherwise, the menu bar might overlay any other parts near the frame window title bar.

Adding menu items to a menu cascade

In this example, the **File** menu cascade choice has some menu items. Follow these steps to create menu items on the **File** menu cascade:

1. Select , the CMenu part, from the right side of the parts palette and drop it on the free-form surface next to the first CMenu part.

2. Select , the CMenuItem part, from the right side of the parts palette.
3. Select **Sticky** from the bottom of the parts palette.
4. Move the mouse pointer over the new CMenu part and drop five CMenuItem parts on the CMenu part.
5. Select , the **Selection Tool**, to unload the mouse pointer.
6. Edit the text of the menu cascade parts. To do this, select each part and press Alt+mouse button 1. The default text is highlighted and you can enter a new name. When you are finished typing, press Shift+Enter. The text of the part is changed. Name the top-most menu cascade part **New**, the next one down **Open**, the third **Save**, the fourth **Save as**, and the last one **Exit**.

Your part should look similar to Figure 68.

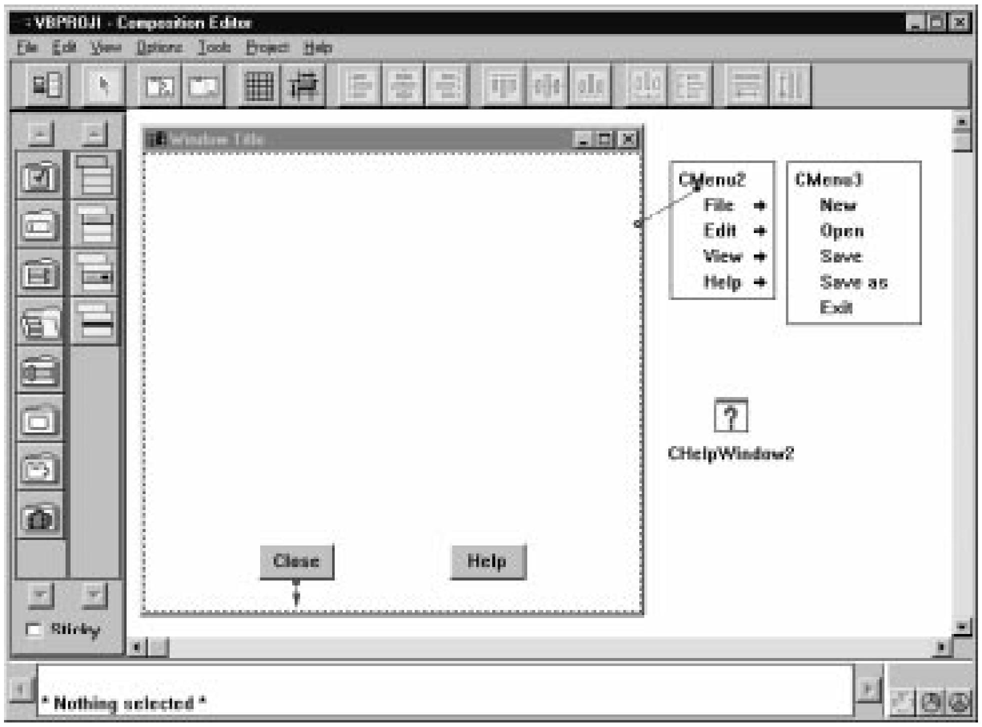


Figure 68. Composition Editor window showing menu item parts in a menu part

Defining a CMenu part as a pull-down menu

After adding the menu items to the new menu part, connect the *this* attribute of the new CMenu part to the *menu* attribute of the **File** menu cascade part. This connection defines the second menu part as the pull-down menu that appears when a user selects the **File** menu bar choice.

Creating a pop-up menu

Pop-up menus are similar to the pop-up menus in the Visual Builder window. For example, in the Visual Builder, move your mouse pointer over any of the three list boxes. Click on mouse button 2 and a menu pops up. This menu is a pop-up menu. In our example, we create a pop-up menu for a list box. To begin creating the pop-up menu, we repeat the same first step we use to create a menu bar: add a CMenu part.

Adding a CMenu part

The first step in creating a pop-up menu is to add the CMenu part:

1. Select , the CMenu part, from the parts palette and drop it on the free-form surface next to the window. Your part should now look similar to Figure 69 on page 180.

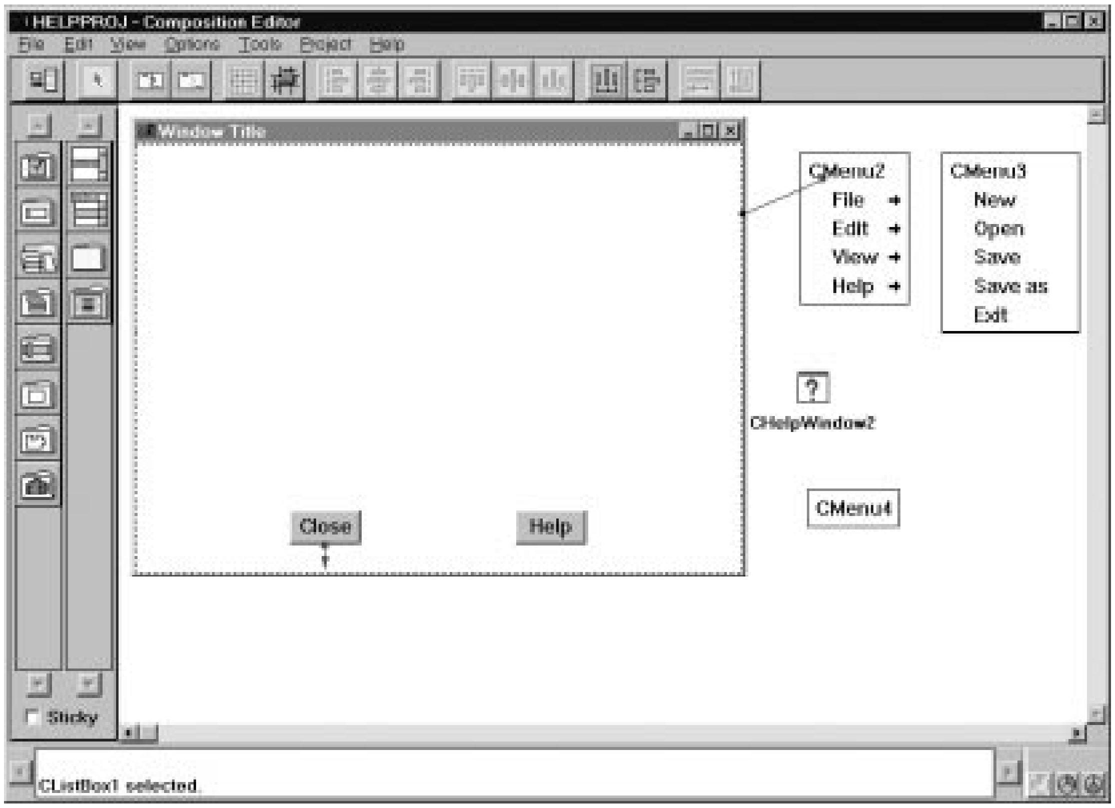


Figure 69. Composition Editor window showing the addition of the third menu part

2. In our example, we are adding pop-up menus to a list box. Select , the Lists category, from the parts the palette.

3. Select , the CListBox part, and drop one list box on the canvas of the frame window.

4. Size and distribute the list box so it takes up about two-thirds of the space on the canvas. Use the distribution tools on the tool bar to space the list box evenly between the right and left edges of the frame window. Refer to “The tool bar” on page 44 for more information on using distribution tools.

   Note: In this example, we will not be filling the list box with data. We are using the list box for illustrative purposes only. To learn how to fill a list box with data, refer to Chapter 11, “Adding containers and list boxes to Visual Builder applications” on page 185.

Defining a CMenu part as a pop-up menu

To make the menu part a pop-up menu, connect the *this* attribute of the CMenu part to the *popUpMenu* attribute of the frame window part. Then connect the *popUp* event of the CListBox part to the *show* action of CMenu part. These two connections define the CMenu part as a pop-up menu. Your part should look similar to Figure 70.

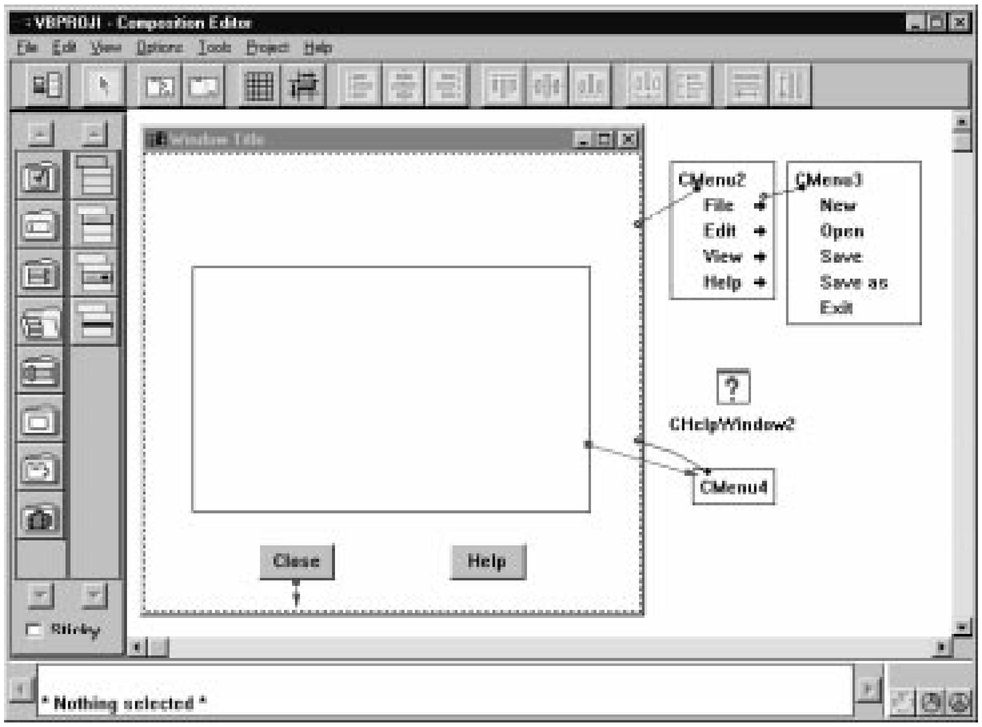


Figure 70. Composition Editor window showing menu part and connections to create a pop-up menu for the list box

Adding menu items to a pop-up menu

In our example, we have two menu items in the pop-up menu: **Edit** and **Delete**. To add the **Edit** and **Delete** menu items:

1. Select , the Frame Extensions category, from the left side of the parts palette.
2. Select , the CMenuItem part, from the parts palette. We use a CMenuItem part because we want to connect the **Edit** and **Delete** menu items to

actions. You can use the CMenuCascade part to create cascade menus on pop-up menus.

3. Select **Sticky** from the bottom of the parts palette.
4. Move the mouse pointer over the new CMenu part and drop two CMenuItem parts on the CMenu part.
5. Select , the **Selection Tool**, to unload the mouse pointer.
6. Edit the text of the menu item parts. To do this, select each part and press Alt+mouse button 1. The default text is highlighted and you can enter a new name. When you are finished typing, press Shift+Enter. The text of the part is changed. Name the first menu item part **Edit** and the next one **Delete**.

Your part should look similar to Figure 71.

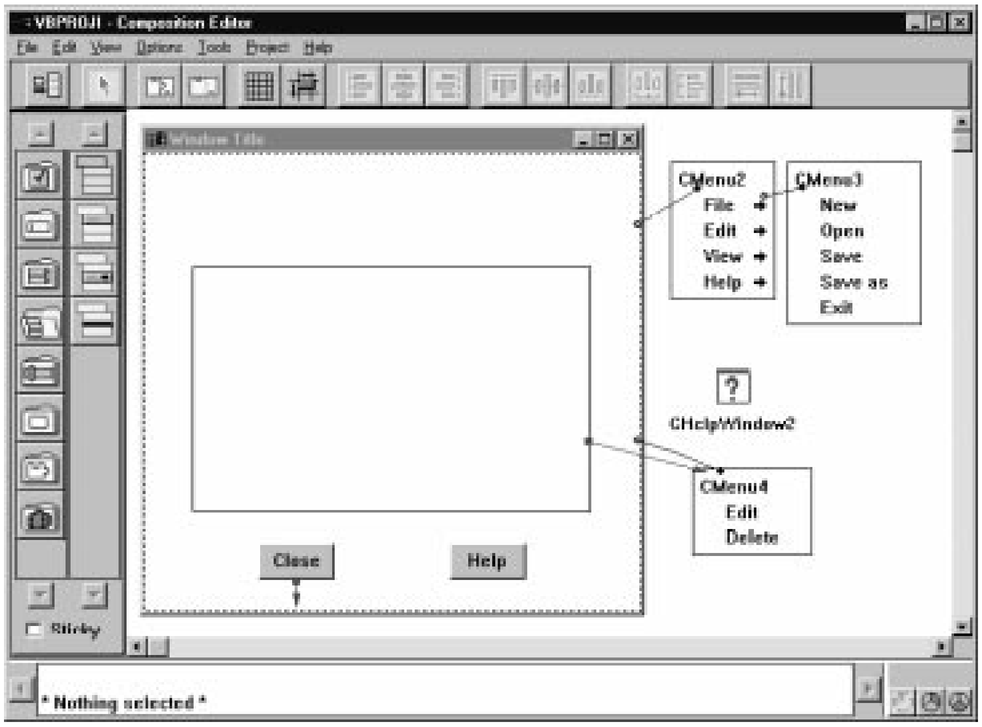


Figure 71. Composition Editor window showing menu parts and connections to create a pop-up menu

If a pop-up menu is connected to an action that depends on the pop-up menu's event data, first connect the *popUp* event of the subpart to an attribute of the part that contains the action. Then, in the action, use the attribute to handle the event data.

Adding menu separators

Menu separators are lines that appear between menu items. They provide a visual break between menu items on the same pull-down menu or pop-up menu. In our example, we place a menu separator between the **Save As** menu item and the **Exit** menu item. This sets the **Exit** menu item apart from the other menu items. To add a separator bar, do the following:

1. Select , the CMenuSeparator part, from the right side of the parts palette.
2. Move the mouse pointer over the second CMenu part containing the **File**, **Open**, **Save**, **Save As**, and **File** menu items. Drop the CMenuSeparator part between the **Save As** and **Exit** menu items. A line appears between the **Save As** and **Exit** menu items.

Connecting menu items to actions

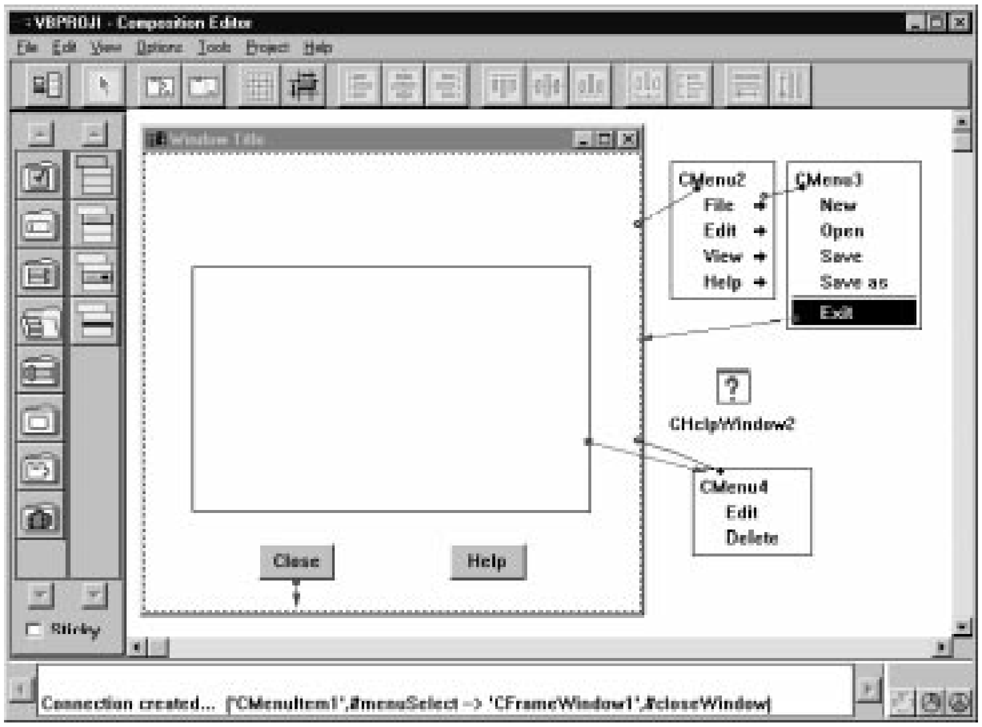
Once you have added menu items, you can connect them to actions in this part or other parts. For example, connect the **Exit** menu item so when it is selected, the window closes, as follows:

From part, feature	To part, feature
Exit,*menuSelect*	FrameWindow,*closeWindow*

The *menuSelect* event occurs when a user selects a menu item. In this case, the user's selection of **Exit** generates a *menuSelect* event to perform the *closeWindow* action on the frame window.

Note: You cannot promote menu item events to the part interface.

The finished part should look like Figure 72 on page 184.



Connection created... ('CMenuItem1',#menuSelect -> 'CFrameWindow1',#closeWindow)

Figure 72. Finished part with all connections and subparts

Building and running the part

Once you have added all the parts and completed all the connections, follow these steps to build and run your part:

1. Select **File→Save and generate→Part source**. This generates the part source.
2. Select **File→Save and generate→Build files**. This generates the build files.
3. Select **Project→Build normal**. This builds the part using the part source and build files created in the previous two steps.
4. Once the build is complete, type `exit` to close the project monitor window. Select **Project→Run**. The window comes up similar to Figure 65 on page 173. Click on the **File** menu to verify the cascade works. Then move the mouse pointer over the list box and click on mouse button 2 to verify the pop-up menu works. Finally, click on **File→Exit** to verify the *closeWindow* action works from the **Exit** menu item.

Chapter 11. Adding containers and list boxes to Visual Builder applications

Containers and list boxes are useful parts to organize large quantities of information visually. You can use containers to organize information containing several different properties, such as a list of employees. This chapter guides you through the construction of two examples: a window with a container and a window with list boxes. The completed windows appear in Figure 73 and Figure 76 on page 192.

Creating container parts

In this section, you learn the steps necessary to set up a container. A completed window looks like Figure 73.

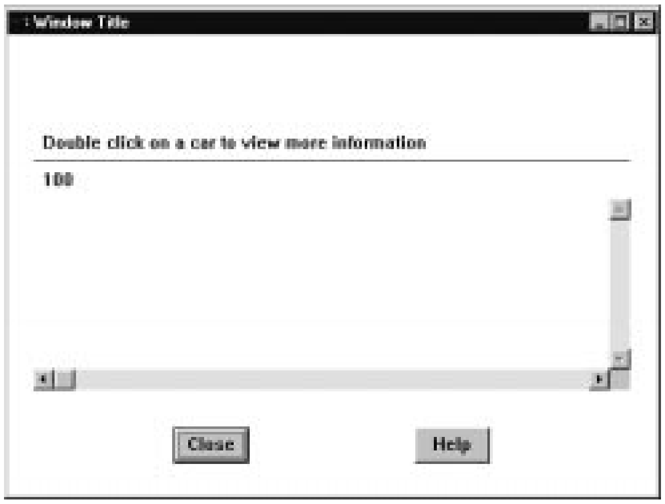


Figure 73. Window with an empty container part

The data required to fill the containers may be provided by nonvisual parts. Before constructing any containers, make sure the nonvisual parts providing data to these containers are loaded into Visual Builder. In the example used in this chapter, an imaginary part called MyNonvisualPart is used. You can create the part using Visual Builder after you have created the project. Refer to “Creating MyNonvisualPart” on page 187 for details.

Container parts

The container parts are found in the Lists category. Containers are constructed using two parts:

CContainerControl

CContainerColumn

The CContainer part is the base in constructing a container. A CContainer may contain instances of CContainerColumn parts.

While your application is executing, you can add and remove objects from the container. A container can display objects in three different formats. These formats are are called view types and are described in “Adding a container part” on page 187. One of these view types (**detailsView**) requires that you use CContainerColumn parts in your CContainer part. Using this view, the CContainerColumn parts are associated to attributes of the objects you want to display in the CContainer part.

For the example used in this chapter, we use a nonvisual part called MyNonvisualPart to provide the data required to fill in a container. The nonvisual part contains the following attributes and types:

Make	VarLengthString
Model	VarLengthString
Year	Integer

The container we construct displays each of the three attributes in it's own column. The following steps are detailed in the rest of this section:

1. Add a CContainer part and associate the nonvisual part to the CContainer.
2. Add three CContainerColumn parts into the CContainer part.
3. You define the first CContainerColumn part to hold data of the *Make* attribute of the MyNonvisualPart part.
4. You define the second CContainerColumn part to hold data of the *Model* attribute of the MyNonvisualPart part.
5. You define the last CContainerColumn part to hold data of the *Year* attribute of the MyNonvisualPart part.

Creating the project

Before proceeding with our example, first create a COBOL Visual Builder project with the following settings:

Project title
Container project

Source file directory
CONTPROJ

Project file name
CONTPROJ

Project target name
CONTPROJ

Once the project is created, the **IBM VisualAge COBOL Project - Container project** window appears. Double click on the CONTPROJ.VCB part file to start the Visual Builder, load the part file, and open the Composition Editor.

Creating MyNonvisualPart

The example described in this section uses a nonvisual part called MyNonvisualPart. To create MyNonvisualPart, do the following:

1. Close the Composition Editor.
2. Select **Part→New** from the Visual Builder window.
3. Specify the **Class name** as MyNonvisualPart.
4. Select the **Part type** as Nonvisual part.
5. Select the **Open** button.
6. Create three attributes, *Make*, *Model*, and *Year*. To create the attributes, repeat the following steps for each attribute:
   a. Type the attribute name in the **Attribute name** field.
   b. Select an attribute type from the **Attribute type** list. “Container parts” on page 185 lists the types to use for each attribute.
   c. Select the **Add with defaults** button.
7. Once you have created the three attributes, save the part. You do not have to generate part source or build files.

Remember to load the part file containing MyNonvisualPart everytime you want to use it.

Adding a container part

To add a container part, do the following:

1. Select , the Lists category, from the left side of the parts palette.
2. Select , the CContainerControl part, from the right side of the parts palette and drop the part onto the default canvas.
3. Resize the container part to match Figure 74 on page 188.

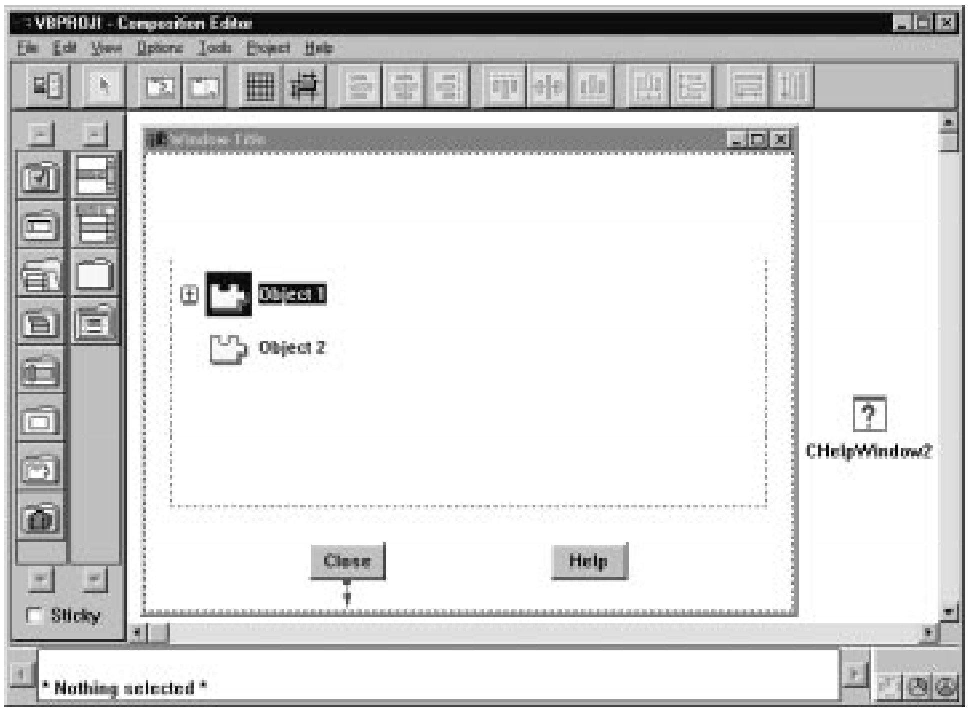


Figure 74. Composition Editor window with a container part

Setting up the container

A container displays specified attributes of an associated part. Each attribute is assigned to a container column in the container column's settings notebook. Settings for a container are set in the container's settings notebook, including the associated part. This section describes how to edit the following settings in the container's settings notebook:

- Title attributes
- View type
- Container item attributes

For information about container columns, see “Adding container columns” on page 190.

Setting the title attributes

You can set the title attributes of a container to display the title of the container with a specific alignment, as well as set the title text and display a title separator. To set the title attributes, do the following:

1. Open the settings notebook for CContainerControl.
2. On the **General** settings page, in the **Title attributes** group, type in a title for the container in the **Title** field. In our example, the title is `Double click on a car to view more information`.

3. To display the title and the title separator, select both **Show title** and **Show title separator**. In our example, we display both.
4. Select an alignment button to align the title. In our example, select **Left** alignment.

Scroll down the **General** settings page to other options of the container part.

Specifying the view type

The view type determines the format the container displays information. The three view type available are:

View-iconView
This view displays items using icons. This view appears similar to the desktop in Windows and OS/2.

View-treeView
This view is similar to the tree view in OS/2 folders. This view displays item along the left-hand side of the container.

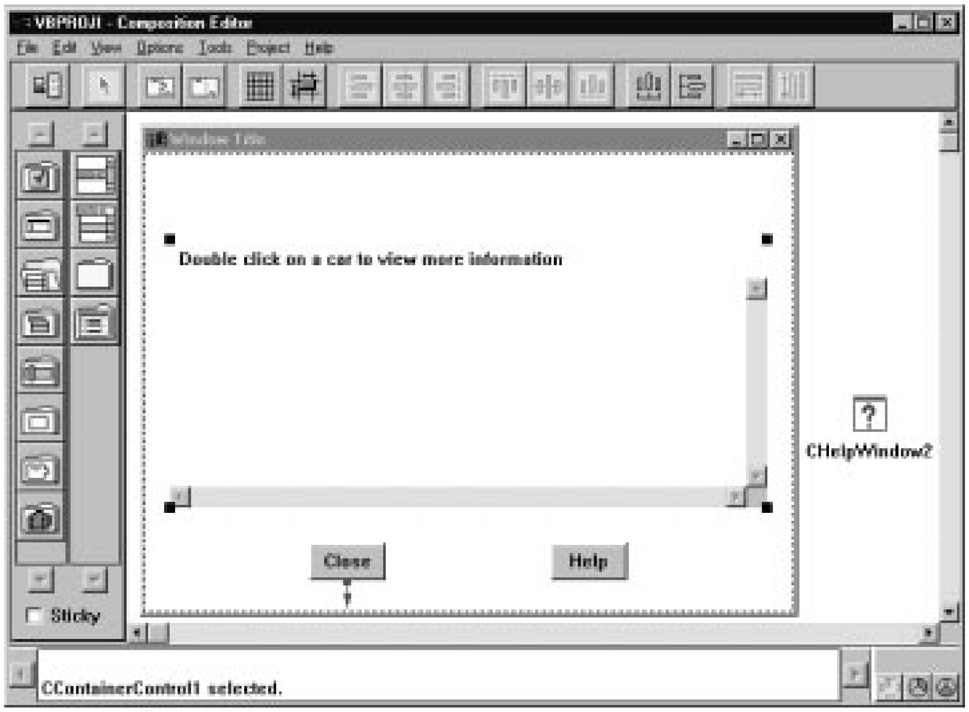
View-detailsView
This view displays information similar to the details view in OS/2 and Windows folders.

Select a view that is appropriate for your purposes. For our example, we select **View-detailsView**.

Specifying the container item attributes

In the **Item type** field, specify the type of the information the container is going to display. In this field, you can specify either a type, or a part name. If you specify a part name, you must use the **View-detailsView** view type. In our example, specify MyNonvisualPart.

After you make all the changes necssary, close the settings notebook. Your part should look like Figure 75 on page 190.


VBPROJ1 - Composition Editor
File Edit View Options Tools Project Help
Window Title
Close
Help
CHelpWindow2
Sticky

Double click on a car to view more information

CContainerControl1 selected.

Figure 75. Composition Editor window with a container column parts in a container part

Adding container columns

If you are using the **View-detailsView** view type, add container columns. A container can have one or more container columns. Each container column holds data of the type of an attribute in the part specified in “Specifying the container item attributes” on page 189. “Setting up a container column” on page 191 describes this set up in more detail.

To add container columns, do the following:

1. Select , the Lists category, from the parts palette.
2. Select the **Sticky** check box.
3. Select , the CContainerColumn part, from the parts palette.
4. Drop as many container column parts as you need on the container part. In our example, we need three container columns.
5. Deselect the **Sticky** check box.

Setting up a container column

Each container column you use must be set up to display an attribute of the part you specified in “Specifying the container item attributes” on page 189. Settings for a container column are set in the container column's settings notebook. You can also alter the following settings for a container column part in the container column's settings notebook:

- Column heading and width
- Attribute to appear in the column
- Vertical and horizontal separators

To set up a container column, do the following:

1. Open the settings notebook for a container column.
2. In the **Column definition** group, you can specify the layout to use to display the attribute in the column, the width of the column, and the alignment of the text in the column. Select the **Help** button for more information on these fields. In our example, make the `Year` column the narrowest, the `Make` column the second narrowest, and expand the `Model` column to fill the remaining width of the container. Use the **Apply** button to check the column widths without having to close the settings notebook.
3. In the **Column attribute item** group, you can specify the attribute to display in the column. The list of attributes available depends on the part you associated to the container. The name of that part is displayed in the **Item type** field. You can not alter the **Item type** field.

   In our example, we use the following:

   - The first column uses *Year*.
   - The second column uses *Make*.
   - The third column uses *Model*.
4. In the **Heading definition** group, you can specify the text, icon (if applicable), and alignment of the text that appears directly above each column.

   In our example, enter the following in the **Text** field:

   - The first column uses `Year`.
   - The second column uses `Make`.
   - The third column uses `Model`.

Once you have made all the changes necessary, close the settings notebook.

Filling in the container with data

The data filling in a container is provided by a nonvisual part. This nonvisual part is defined to the container in “Setting up the container” on page 188. The way in which you fill in a container depends on the design you choose. Some of the possible scenarios include:

- The container is in the same window as the request to fill it. In this case, you need to make sure your connections follow a precise order where the container is filled when the data becomes available. The Employee Lookup sample follows this paradigm.
- The container appears in it's own window and is filled with data. In this case, you want to connect the *ready* event of the part containing the container to the process that fills the container with data.

Building the part

Once you have added all the parts and completed all the connections, follow these steps to build and run your part:

1. Select **File→Save and generate→Part source**. This generates the part source.
2. Select **File→Save and generate→Build files**. This generates the build files.
3. Select **Project→Build normal**. This builds the part using the part source and build files created in the previous two steps.
4. Once the build is complete, select **Project→Run**. The window comes up similar to Figure 73 on page 185.

Creating list box parts

A list box is a convenient part to use to display related information. For instance, cars have related information: make, model, price range. In this section, you construct a window with three list boxes. The completed window appears Figure 76.

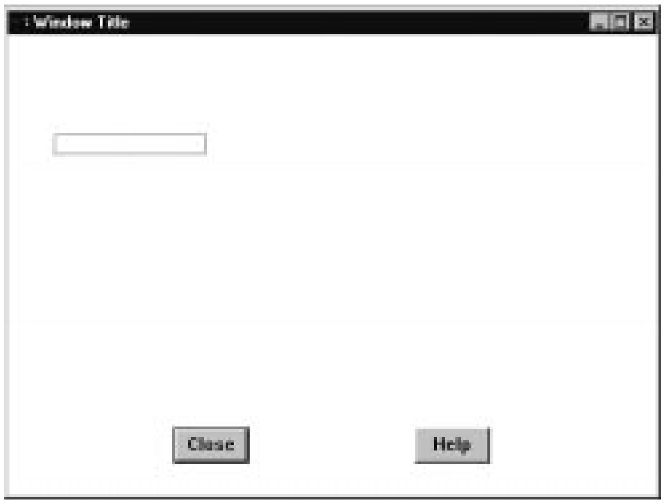


Figure 76. Window with empty list boxes

Creating the project

Begin by constructing a Visual Builder project. Use all the default options, except for the following:

Project title
List box project

Source file directory
LBPROJ

Project file name
LBPROJ

Project target name
LBPROJ

Once you create the project, the project window appears. Double click on LBPROJ.VCB to open the Visual Builder, load the part file (LBPROJ.VCB), and open the Composition Editor.

Adding list boxes

To add the list box parts, do the following:

1. Select the , Lists category, from the parts palette.
2. Select the **Sticky** check box.
3. Select the , ListBox part, from the parts palette. Drop three list box parts on the canvas of the frame window.
4. Unload the mouse pointer by unchecking the **Sticky** check box.
5. Size and position the list boxes as shown in Figure 77 on page 194 using the alignment and distribution tools on the tool bar. Refer to “The tool bar” on page 44 for more information on using the alignment and distribution tools.

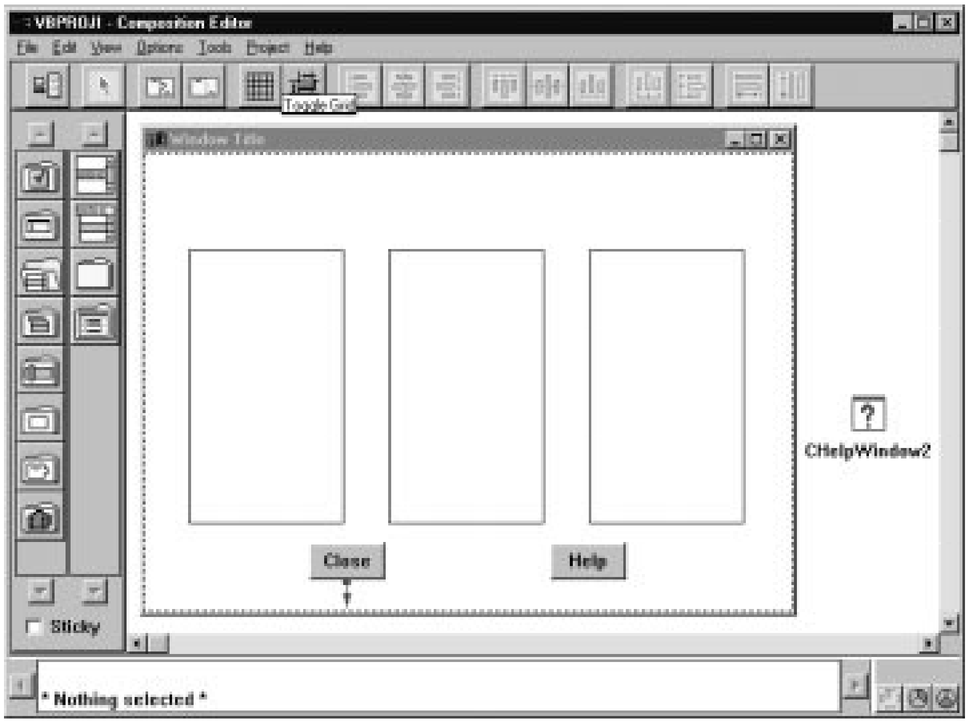


Figure 77. Composition Editor window with list box parts

Filling in the list boxes with data

The data filling in a list box is provided by a nonvisual part. The way in which you fill in a list box depends on the design you choose. Some of the possible scenarios include:

- The list box is in the same window as the request to fill it. In this case, you need to make sure your connections follow a precise order where the list box is filled when the data becomes available. The Employee Lookup sample follows this paradigm, using a container.
- The list box appears in it's own window and is filled with data. In this case, you want to connect the *ready* event of the frame window to the process that fills the list box with data.

Building and running the part

Once you have added all the parts and completed all the connections, follow these steps to build and run your part:

1. Select **File→Save and generate→Part source**. This generates the part source.
2. Select **File→Save and generate→Build files**. This generates the build files.
3. Select **Project→Build normal**. This builds the part using the part source and build files created in the previous two steps.

4. Once the build is complete, select **Project→Run**. The window comes up similar to Figure 76 on page 192.

Chapter 12. Adding notebooks to Visual Builder applications

Notebooks can help you organize detailed information about an object. For instance, you can have a notebook represent a single employee, with several different tabs of information categorizing information about the employee. This section guides you through the construction of a sample notebook. The completed sample looks like Figure 78.

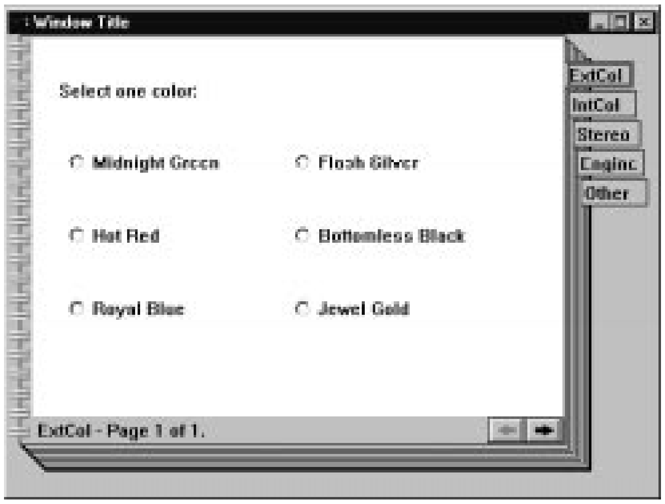


Figure 78. A sample window with notebook part

Creating the project

Begin by constructing a Visual Builder project. Use all the default options, except for the following:

Project title
Notebook example

Source file directory
NBPROJ

Project file name
NBPROJ

Project target name
NBPROJ

Once you create the project, the project window appears. Double click on NBPROJ.VCB to open the Visual Builder, load the part file (NBPROJ.VCB), and open the Composition Editor.

Adding a notebook part

For this example, we replace the default **Close** and **Help** push buttons with the notebook part. Follow these instructions:

1. Select the canvas part on the frame window and hit the DELETE key. A message appears asking you to confirm you want to delete the part and it's connection. Select the **OK** button.
2. Select , the Composers category, from the left side of the parts palette.
3. Select , the CNotebook part, from the right side of the parts palette and drop the part onto the canvas. Resize the notebook so that it covers the canvas completely.

You part should look like Figure 79.

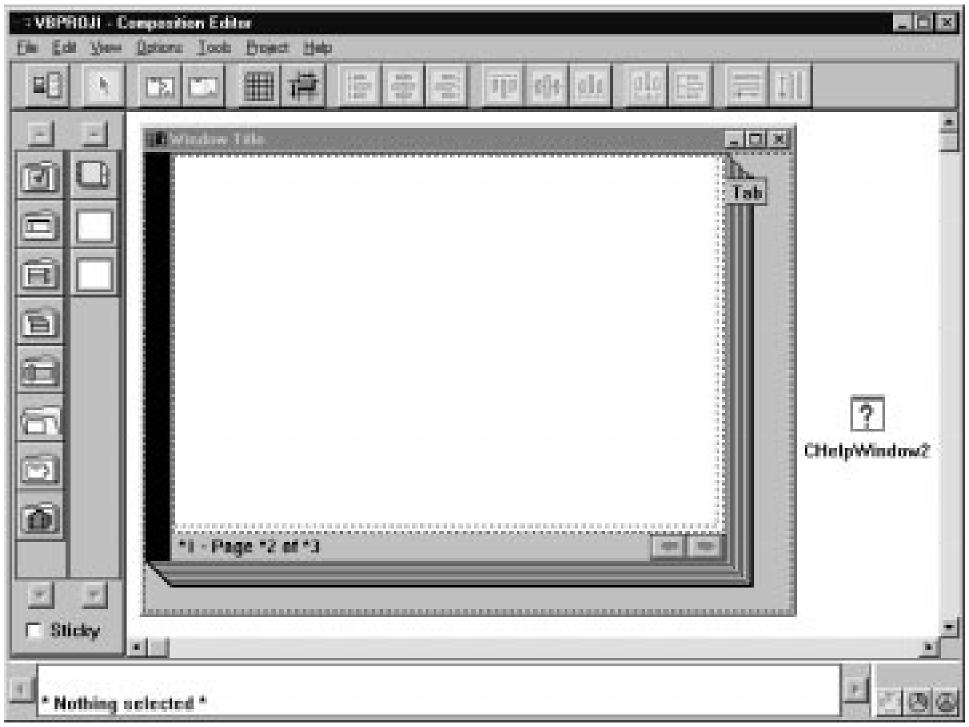


Figure 79. Composition Editor window with a notebook part

When you construct a CNotebook part, it initially comes with one CNotebookPage and a CCanvas part on the CNotebook page. To confirm this, click mouse button 2 on the CNotebook part. Select **View parts list** from the pop-up menu. The parts list for the

CNotebook part appears and the only part on the CNotebook part is a CNotebookPage part. Click on the expansion icon (plus sign) and you see the only part on the CNotebookPage part is a CCanvas. Being able to use the parts list of a CNotebook part becomes important when you start adding more pages and tabs.

Specifying the notebook layout and appearance

You change the notebook’s appearance using its settings notebook. Some of the changes you can make include changing the direction and type of binding, the type of tabs, and the justification of the text in tabs and status text areas. To create the notebook in our sample, do the following:

1. Double-click on the CNotebook part to open the settings notebook. The settings notebook appears and opens to the **General** settings tab, which contains the settings related to the layout of the notebook.
2. The first layout setting you can change is the orientation of the notebook. From the **Layout** group, select the icon representing the orientation you want for the notebook. In our example, we use the last layout, with the binding facing the bottom.
3. The second layout setting you can change is the binding of the notebook. From the **Binding** group, select the type of binding you want for your notebook. In our example, we use the default.
4. The next layout setting you can change is the shape of the notebook tabs. From the **Tab shape** group, select the shape you want for your tabs. In our example, we use the default setting.
5. The next layout setting you can change is the justification of text in the tabs and in the status text area. The status text area is at the bottom of each page which can display one line of text. From the **Justification** group, select the justification you want for the text in the status area and the text in the tab. In our example, we use the default settings.
6. The final layout setting you can change is the format of the text in the status area. Use the **Status text template** field to specify the appearance of text in the status area of a notebook. Select the **Help** button of the settings notebook for a description of how to specify a template. In our example, we use the default settings.
7. Select the **OK** button to accept these options and close the settings notebook.

Your part should look like Figure 80 on page 200.

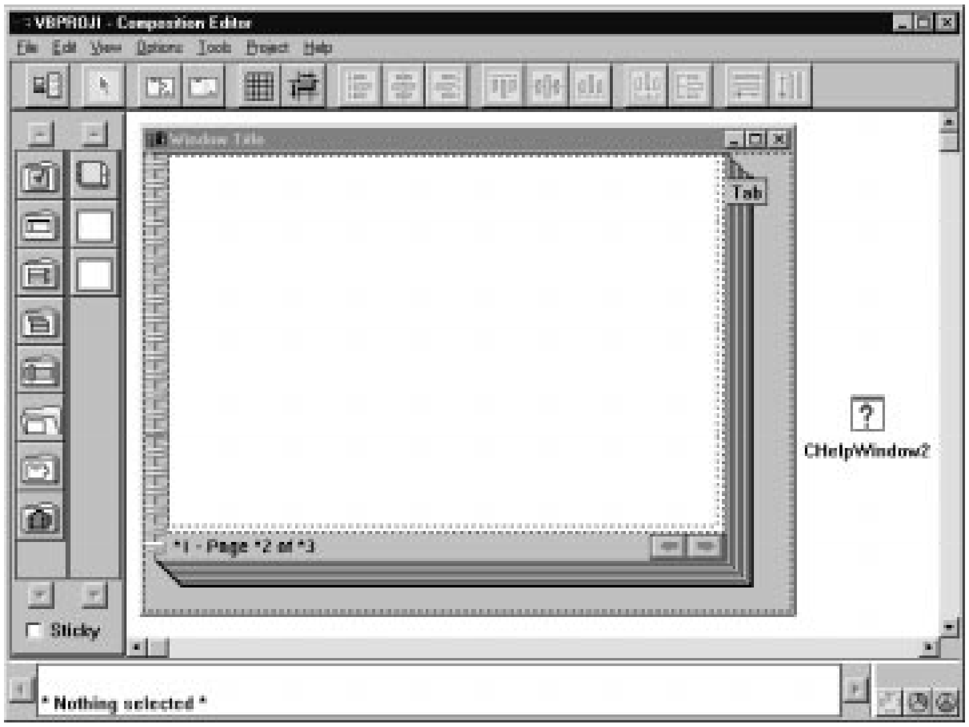


Figure 80. Composition Editor window with notebook part

Adding notebook pages

The first notebook page was added for you when you dropped the CNotebook part on to the CCanvas part. You can add notebook pages from either the CNotebookPage part or the CNotebook part.

To add notebook pages from the CNotebook part, do the following:

1. Open the pop-up menu for the CNotebook part and select **Add page**.
2. From the cascade menu, select either **Before top page** or **After top page**.

To add notebook pages from the CNotebookPage part, do the following:

1. Open the pop-up menu for the CNotebookPage part and select **Add page**. Be careful to select the CNotebookPage part, and not the CCanvas part on top of the CNotebookPage part.

   The easiest way to open settings for a notebook page is through the Parts List window. Select the notebook; select **View parts list** from the notebook's contextual menu. Double-click the icon that represents the notebook page you want to set.

2. From the cascade menu, select either **Before** or **After**.

The rest of this section guides you through setting up the notebook pages and tabs for our example.

Setting up the notebook pages and tabs

To set up the notebook pages and tabs, do the following:

1. Open the settings notebook for the first notebook page.
2. Type in the text you want to appear on the tab into the **Tab text** field.

   Note: If you use the default font, you are limited to about six characters. In our example, we abbreviate Exterior Color to ExtCol.
3. Type in the text you want to appear in the status area into the **Status text** field. In our example, we use the default settings.
4. Click on the **Styles** tab and select from the following styles:
   - The **autoPageSize** style, to enable automatic sizing of the notebook page. In our example, use the default setting.
   - The **statusTextOn** style, to enable display of the status text. In our example, use the default setting.
   - The **majorTab** style, to give the notebook page a major tab or **minorTab** to give the page a minor tab. These styles are mutually exclusive, so you cannot have both styles chosen.

     Note: The first page in a notebook must be a major tab.
5. Select the **OK** button to save these settings and close the settings notebook.

To continue constructing the notebook as per our example, add pages after the first page and follow the previous steps, with the following settings:

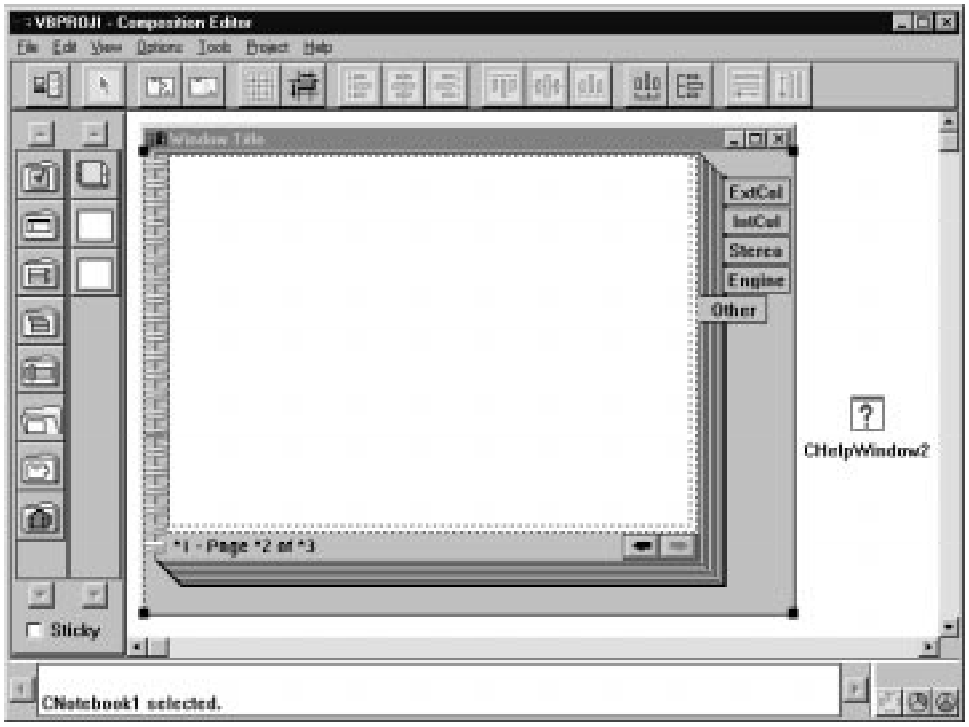
Second page
: **Tab text**: IntCol (abbreviation for Interior Color). Use defaults for all other settings.

Third page
: **Tab text**: Stereo. Use defaults for all other settings.

Fourth page
: **Tab text**: Engine. Use defaults for all other settings.

Fifth page
: **Tab text**: Other. Use defaults for all other settings.

You part should look like Figure 81 on page 202.



CNotebook1 selected.

Figure 81. A sample window with notebook part

Adding parts to a notebook page

Each notebook page initially contains a CCanvas part. You can add Visual Builder-supplied parts like push buttons and entry fields on the CCanvas part. Figure 82 on page 203 contains a notebook page with Visual Builder-supplied parts. To see examples of notebook pages, open any part settings notebook.

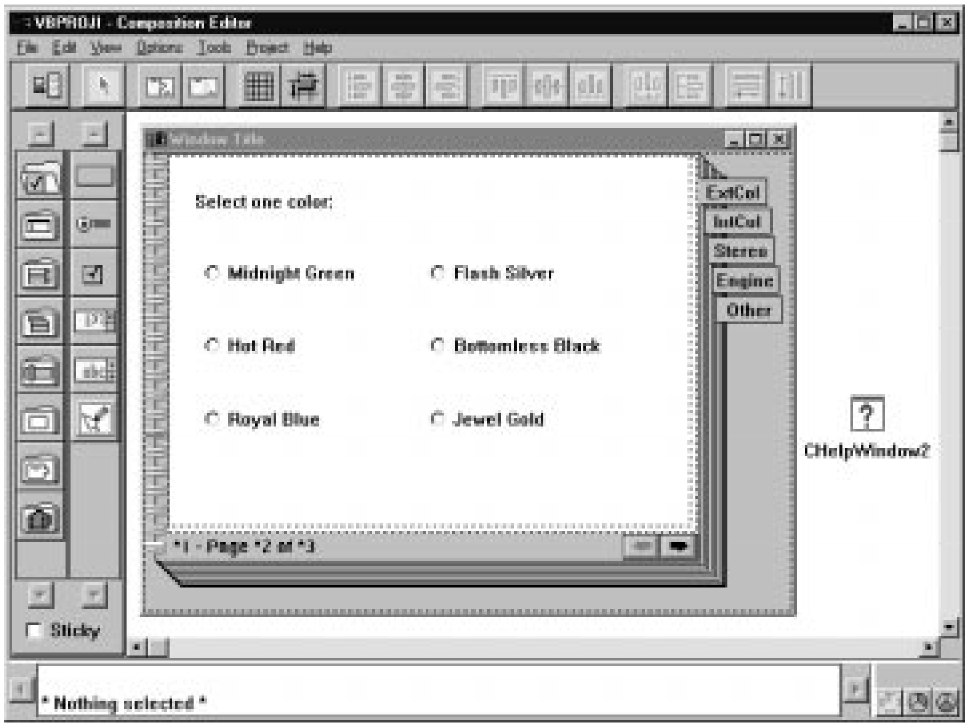


Figure 82. A notebook page with Visual Builder-supplied parts

For our example, we add parts to the **ExtCol** page only, as follows:

ExtCol page

1. Drop one text static text part on the canvas. Change the text to *Select one color:*.
2. Drop six radio button parts on the canvas. Change the text to *Midnight Green*, *Hot Red*, *Royal Blue*, *Flash Silver*, *Bottomless Black*, and *Jewel Gold*.
3. Use the distribution and alignment tools on the tool bar to distribute and align the parts so they appear even within the canvas. For more information on the distribution and alignment tools, refer to “The tool bar” on page 44.

Building and running the part

Once you have added all the parts and completed all the connections, follow these steps to build and run your part:

1. Select **File→Save and generate→Part source**. This generates the part source.
2. Select **File→Save and generate→Build files**. This generates the build files.

3. Select **Project→Build normal**. This builds the part using the part source and build files created in the previous two steps.
4. Once the build is complete, select **Project→Run**. The window comes up similar to Figure 78 on page 197. Click on tabs to verify you can open the pages. Click on items in the page to verify they work.

Chapter 13. Adding help to Visual Builder applications

Once you develop the user interface, provide the necessary help panels to complete the user interface. In this chapter, you learn about adding different types of help panels to your user interface:

Help Type	Description
Context-sensitive help	Help information for the current choice, object, or group of choices or objects. The user can display context-sensitive help by tabbing or cursoring to a choice or other object and doing either of the following: • Pressing the F1 key. You do not have to do anything extra to make this happen. The operating system (through the IBM Open Class Library) handles it for you. • Selecting the **Help** push button if you have provided one. See "Providing a Help button" on page 211 for information on how to add a **Help** push button.
General help	Help for a specific window, explaining the purpose of the window and how it operates.

This chapter guides you through the process of adding the two basic types of help panels to a window. The completed window looks like Figure 83. The window contains a **Help** button, which opens up general help, and several entry fields, each conected to context-sensitive.

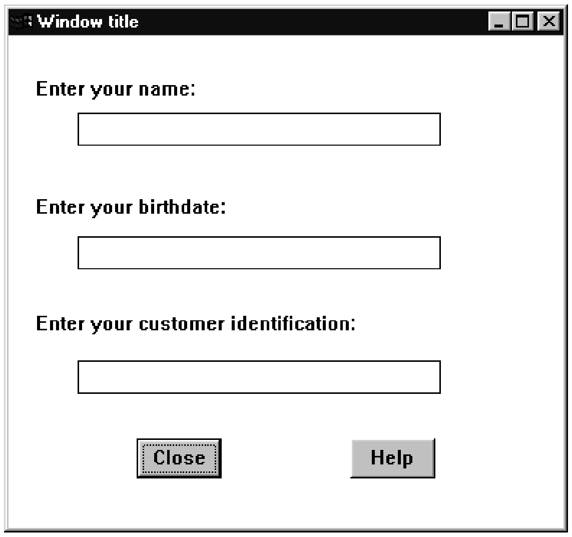


Figure 83. The completed window for help example

The help subproject

When you create a COBOL Visual Builder project, WorkFrame creates a help subproject. The subproject contains a help source file. This help source file contains template code to create general or context-sensitive help. When you build your project, a step in the build process compiles the help source file. In this chapter, you will use this help source file to create context-sensitive and general help for the window on Figure 83 on page 205.

Writing portable help

If you want to use your help file in both OS/2 and Windows, choose the Information Presentation Facility (IPF) format when you construct your COBOL Visual Builder project. For more information about creating help source code in IPF format, refer to the *IPF User's Guide*.

If portability is not a concern for your Windows application, choose Rich Text Format (RTF) when you create your COBOL Visual Buider project. For information about creating help source code in RTF, see your Windows documentation.

Creating the help file

If you are not creating Visual Builder applications in COBOL Visual Builder projects, you must create the help source file. Use your favorite editor to create a new file and type in the help source. Save this file and be sure to use an .ipf extension when naming the file. Compile this file from the command line using the Information Presentation Facility compiler, using the `ipfc` command.

Editing the help source file

Before proceeding with our example, first create a COBOL Visual Builder project with the following settings:

Project title
Help project

Source file directory
HELPPROJ

Project file name
HELPPROJ

Project target name
HELPPROJ

Once the project is created, the **IBM VisualAge COBOL Project - Help project** window appears. Double click on the HELPPROJ.VCB part file to start the Visual Builder, load the part file, and open the Composition Editor. Edit the visual part provided to match Figure 83 on page 205. Do not build the project when you are done. Save the part and exit out of Visual Builder.

To edit the source file provided with the COBOL Visual Builder project, do the following:

1. Double click on the help subproject: **VBHELP.IWP**. The help subproject window appears.
2. Double click on VBHELP.IPF, the source file created when the COBOL Visual Builder project was created. The COBOL Editor appears, with the following default code:

```
.* ------------------------------------------------
.*
.*       VisualAge for COBOL
.*     IPF Help FIle Template
.*
.* ------------------------------------------------
:userdoc.
:title.Help
:docprof toc=12.
.* -----
:h1 res=100.Main Window Help
:p.
This is the Help for the Main Window.
.* -----
:h2 res=110.Close Push Button
:p.
Click on the :hp2.Close:ehp2. push button to close the
window.
.*
:euserdoc.
```

3. Create help panels for the three entry fields using 120, 130, and 140 as the resource IDs. Do this by adding the following text before the `:euserdoc.` line:

```
.* -----
:h1 res=120.First entry field help
:p.
This is the help for the first entry field.
.* -----
:h1 res=130.Second entry field help
:p.
This is the help for the second entry field.
.* -----
:h1 res=140.Third entry field help
:p.
This is the help for the third entry field.
```

4. Select **File→Save** from the Editor's menu to save the file.
5. Select **File→Exit** from the Editor's menu to close the Editor.

In the default code, the :h1 tags are heading tags. These tags cause the IPF compiler to create a new help panel using the text on the tag as the panel's title. The *res* parameter specifies the panel's resource ID. For more information about creating help source code in IPF format, refer to the *IPF User's Guide*.

If you create RTF help, you must define resource IDs for each panel in your .rtf file.

Make a note of all panel resource IDs; you need them to set help support properly in your visual parts. See “Providing context-sensitive help” to learn how these resource IDs are used.

Building the help source file

To build your help from the COBOL Visual Builder project:

1. Select **Project→Build Normal**. The project monitor displays the status of the build process.
2. When the build process is done, type `exit` in the project monitor to close it.

   Note: You can't run a help subproject.

If you are not creating your application from a COBOL Visual Builder project, build your help file from the command line. Ensure that you have the Information Presentation Facility (IPF) compiler installed on your system. The IPF compiler comes with VisualAge COBOL.

To build the help file, simply run the IPF compiler. For example, if you saved your help file with the name iwzhpb.ipf, you enter the following command in the directory where you saved your file:

```
ipfc iwzhpb.ipf
```

This command generates a file called iwzhpb.hlp.

You have now built your help file. The next step is to use the iwzhpb.hlp file to provide context-sensitive and general help in your application.

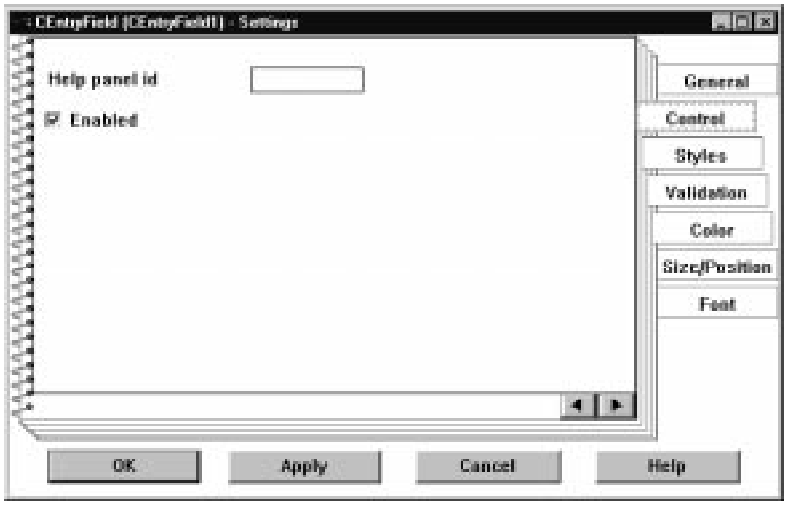
Providing context-sensitive help

This section tells you how to provide context-sensitive help for parts in your application. If you use the default window created by WorkFrame, the initial context-sensitive help for the **Help** and **OK** button are already provided.

Before you begin, make sure you have a list of your help panel titles and their corresponding resource IDs.

To provide context-sensitive help, do the following:

1. Double-click on HELPPROJ.VCB in the **IBM VisualAge COBOL Project - Help project** window. The Visual Builder opens, loads HELPPROJ.VCB, and opens the Composition Editor.
2. Open the settings notebook for the first entry field. See “Changing settings for a part” on page 129 if you need information on how to do this.
3. Select the **Control** page. It is similar to Figure 84 on page 209.



Help panel id

☑ Enabled

Figure 84. The Control page of an entry field part's settings notebook

4. Enter the resource ID for the appropriate help panel in the **Help panel ID** field. In this example, use `120` for the first entry field help.
5. If the **Enable** check box is not checked, select it.
6. Select the **OK** button.

Repeat these steps for the other two entry fields, using the appropriate resource IDs. The next step is to provide general help for your application.

Providing general help

This section describes how to provide general help for your application. If you are using COBOL Visual Builder projects to construct your application, the project and part are set up to provide general help. You can skip this section.

Before you begin, make sure you have a list of your help panel titles and their corresponding resource IDs.

To provide general help, do the following:

1. Open the settings notebook for the CFrameWindow-based part. See "Changing settings for a part" on page 129 if you need information on how to do this.
2. Select the **Control** page. It is similar to Figure 84.
3. Enter the ID for the general information help panel in the **Help panel ID** field. For this example, use `100` for the General help for this application help panel.
4. If the **Enable** check box is not checked, select it.
5. Select the **OK** button.

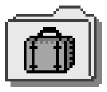
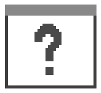
If you are not building your application with COBOL Visual Builder projects, the next step is to provide a help window in which to display the help panels.

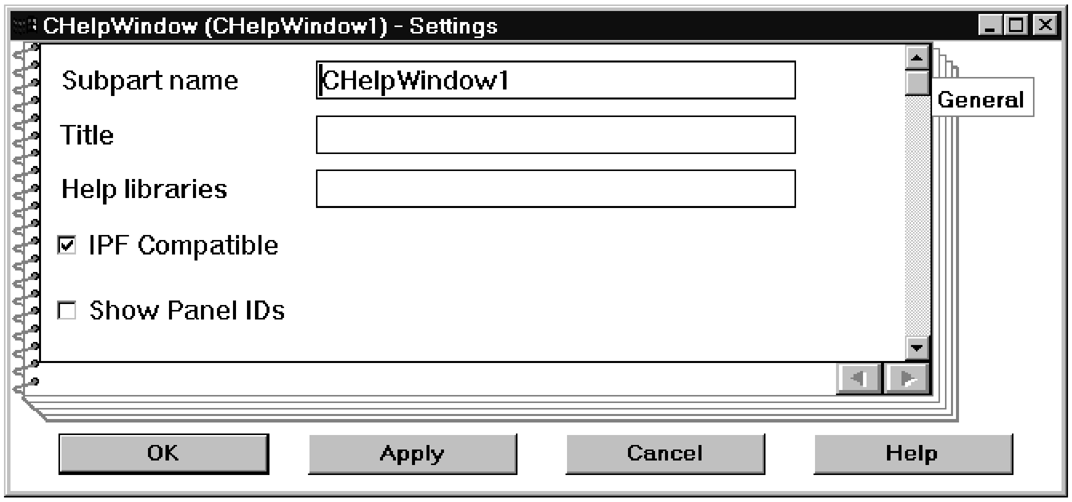
Providing the application Help window

If you do not use COBOL Visual Builder projects to create your application, you must place a CHelpWindow part on the free-form surface to give Visual Builder a window in which to display the help information.

The owner of the CHelpWindow part is the primary part for your application.

To add a help window to your application, do the following:

1. Select , the Other category, on the parts palette.
2. Select , the CHelpWindow part, and place it on the free-form surface.
3. Open the settings notebook for the CHelpWindow part. See "Changing settings for a part" on page 129 if you need information on how to do this. It is similar to Figure 85.



Subpart name: CHelpWindow1
Title:
Help libraries:
☑ IPF Compatible
☐ Show Panel IDs

Figure 85. The help window's settings notebook

4. In the **Title** field, enter the title of the help window.

   This must be the same title you entered on the :title tag in your help source file.
5. In the **Help libraries** field, enter the name of the help file you compiled.

   If you had created multiple help files for your application, you would enter all of their names in this field.

6. To use IPF help in Windows, select the IPF Compatible check box.
7. Select the **OK** button.

Note: You must have a CHelpWindow part in the main part of your application, even though you only use it in your subparts.

Providing a Help button

Many applications provide a help button to give users quick and easy access to the help information the application provides. If you are using COBOL Visual Builder projects to create your application, a **Help** button is already provided. You can skip this section.

To provide a **Help** button in your application, do the following:

1. Select , the Buttons category, in the left-hand column on the parts palette.
2. Select , the CPushButton part, and place it where you want it to be.
3. Change the text on the push button to `Help`.
4. Open the settings notebook for the push button.
5. In the **General** page of the settings notebook, select **Display Help** in the **Action on the press event** group. This setting turns a regular push button into a help push button.
6. Select the **OK** button to close the settings notebook.

You now have a **Help** button. If you have followed the steps in the preceding sections, clicking this button causes the contextual help panel for the part with input focus to be displayed. If no part has the focus, the main help panel for the window is displayed. The behavior of the **Help** button is identical to the F1 key.

To provide a help panel for the **Help** button itself, follow the instructions in "Providing context-sensitive help" on page 208.

Building and running the part

Once you have added all the parts and completed all the connections, follow these steps to build and run your part:

1. Select **File→Save and generate→Part source**. This generates the part source.
2. Select **File→Save and generate→Build files**. This generates the build files.
3. Select **Project→Build normal**. This builds the part using the part source and build files created in the previous two steps.

4. Once the build is complete, select **Project→Run**. The window comes up similar to Figure 76 on page 192. Click on the **Help** push button to verify the general help panel appears. Then click on an entry field and hit the F1 key. Verify the proper help panel appears.

Chapter 14. Integrating visual parts into a single application

In this chapter, you create two visual parts that demonstrate how to use factories and variables to create dynamic applications. This chapter includes the following tasks:

- Creating the dynamic visual parts
- Creating the static visual parts
- Adding visual parts as dynamic instances
  - Adding and setting factory parts
  - Adding and setting variable parts
  - Connecting to the factory parts
  - Connecting the factory parts to their corresponding variable parts

This chapter constructs a simple application with two windows. The main window contains two push buttons and one static text field. The main window is a static part, which means once it is destroyed, it can not be reinstantiated. The second window is dynamic, which means it is created and destroyed, as needed, while the application runs.

Creating the dynamic visual parts

Begin by creating the dynamic visual parts. Dynamic visual parts must be based on a frame window part. You can create subwindows which are dynamic visual parts, such as the one in our example.

For our sample application, create a window as a visual part and name the part **GenericGreeting**. This subwindow has a push button and static text. Label the push button part, static text part, and frame window part as shown:

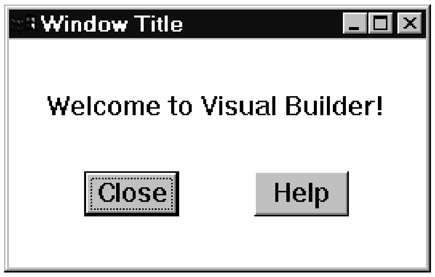


Figure 86. Dynamic window

Creating the static visual parts

One visual part, the main window, exists statically in this example. When you use frame window-based static visual parts note that once the user closes the window, the part is destroyed and cannot be instantiated again until the parent window is recreated.

Create the main window part and call it **MainWindow**. Add two push buttons and a static text field as shown:

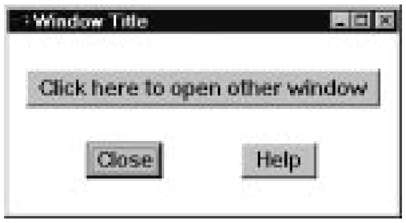


Figure 87. The main window

You can add other static window parts by dropping a frame window part on the free-form surface of this part.

Adding visual parts as dynamic instances

All the visual parts in this example, except for the main window, are created at run time as the user requests them from a button selection. Factory parts *create* the dynamic visual parts, variable parts *represent* the dynamic visual parts created by the factory. Like static visual parts, dynamic visual parts are destroyed when the user closes them. However, the existence of the factory enables the creation of a new instance of that same part the next time the user requests it.

Factory parts create other parts. Each factory part has a type which you set to the part it represents. The factory part works in tandem with a variable part that represents the dynamic part instance. You can use factory parts to create both visual and nonvisual parts. The visual parts must have a frame window part as a parent.

Factory parts have a *new* action to create a new instance. The *new* action triggers a *newObject* event that can be used to connect the new instance of a part or an attribute of the new instance of a part to other parts containing the part. Variable parts stand in for instances created elsewhere.

Implementing dynamic parts involves the following tasks:

- Adding and setting factory parts
- Adding and setting variable parts
- Connecting to the factory parts
- Connecting the factory parts to the corresponding variable parts

Adding and setting factory parts

Before you can use factory parts, you must have created the part classes to be represented. Before you start, be sure to load the part files (.VCB) that contain those parts into Visual Builder.

Adding the factory parts

1. Open the main window part.
2. Select , the **Models** category, from the left side of the parts palette.
3. Select , the CFactory part, from the right side of the parts palette.
4. Drop a CFactory part on the free-form surface.

Setting a factory part

1. Change the factory part name using the part's pop-up menu.

   In this example, the CFactory part is named as follows:

GenericGreetFactory	Factory creating generic greeting subwindow

2. Change the part type from the default (CStandardNotifier) to the type of the window it creates using the part's pop-up menu and selecting **Change Type**.

   In this example, the CFactory part has the following types:

GenericGreeting	Type of the GenericGreeting part

Adding and setting variable parts

When used with factory parts, variable parts represent the newly created part instance. Add and set variable part types by opening the variable part's pop-up menu and selecting **Change type...**. For this example, add and set variable parts as follows:

1. Select , the **Models** category, from the left side of the parts palette.
2. Select [ ] , the variable part, from the right side of the parts palette.
3. Drop a variable part on the free-form surface.
4. Change the name of the variable using the variable part's pop-up menu to the following:

GenericGreetVariable	Name of the variable representing GenericGreeting part

5. Change the type to the type of the window created using the variable part's pop-up menu to the following:

Name	**Type**
GenericGreeting	Type of the GenericGreeting part

Connecting to the factory parts

Once you have added and set both the factory and variable parts, connect the push button parts on the main window to the factory parts representing the dynamic windows, as follows:

From part, feature	**To part, feature**
CPushButton1, *press*	GenericGreetFactory, *new*

Connecting the factory parts to their corresponding variable parts

Once you have set both factory and variable parts, you must connect them to the variable part that represents them.

Connections for modeless windows

You must make connections to the modeless windows.

From part, feature	**To part, feature**
GenericGreetFactory,*newEvent*	GenericGreetVariable,*this*

Now you can connect subparts of the main window to features of the variable part, features of the part the variable represents. These connections behave in the same way static subpart connections behave, with one difference: the behavior applies to the *instance* the variable currently represents. By changing the instance, you can change the effect of the connection.

Part 4. Extending Visual Builder applications

This part provides the information you need to extend your applications beyond the basic functions that Visual Builder provides.

Extending Applications

Chapter 15. Hints and tips for using Visual Builder

This chapter contains hints and tips that you might find useful when using Visual Builder.

Porting your application
In general, the source of GUI applications developed either on OS/2 or on Windows NT can be copied to the other platform, rebuilt, and executed. However, some settings and methods of parts in VAccess.vcb are ignored. To determine if a certain setting or method is ignored, do the following:

1. Open the Visual Builder and load a part file (.VCB).
2. Select a part and press mouse button 2.
3. select **Browse part features** from the pop-up menu.
4. Press the **Help** push button and follow the directions on the help panel to find out more about features.

Using visual parts with nonvisual parts
You can not have a visual part be a subpart of a nonvisual part.

Using part supplied in VAccess.vcb as objects
The parts supplied in VAccess.vcb are pointers and not COBOL objects. To create a COBOL object with behavior similar to a part supplied in VAccess.vcb, place the part on a canvas and promote its features.

Words to avoid
The part source Visual Builder generates contains code which can contain unique variable or parameter names. In order to avoid compilation errors, avoid using the following terms when naming your features and parts:

- anObject
- iActionResult
- windowIdIn
- inlineVariable1
- aSELF
- actionResult
- windowIdOut
- inlineVariable2

Extending Applications

Part 5. Appendixes

Appendix A. Creating part information files

This appendix describes the syntax of part information files.

Describing part interfaces in part information files

You can 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 statements used to describe the interface information.

You include the statements describing the interface in a *part information file*, which has a .VCE file extension. All interface information code lines begin with `//VB` in column 1. Between statements, lines that do not start with `//VB` are ignored and can be use for comments. You can arrange statements on a single line or continue them on multiple lines by using the `//VB:` continuation statement.

General rules for entering part information

The following rules apply to all interface information statements:

1. The interface information statements must be entered exactly as shown in this chapter. You should pay particular attention to using the correct case, because the Visual Builder's interpretation of the statements is case-sensitive.
2. All part, enumeration, and type definition names must be unique.
3. A single file can contain multiple part, enumeration, and type definitions.
4. You must completely specify one type of information before you begin to specify another. For example, you should not put a `//VBBeginEnumeration:` statement between `//VBBeginPartInfo:` and `//VBEndPartInfo:` statements. If you do, you will not be able to successfully import the part information.
5. The interface information for a specific part, enumeration, or type definition must be contained in a single file. The file extension required for a VisualAge COBOL part information file is .VCE.
6. If you omit a field from a statement, and wish to code a subsequent field, then you must account for the missing field 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: VisualPart

   //VBComposerInfo: VisualPart,803,cobov33r,

   //VBComposerInfo: VisualPart,,,
   ```

   The following statement is not acceptable:

```
//VBComposerInfo: VisualPart,cobov33r
```

The individual statement descriptions will tell you when missing keywords must be accounted for by commas.

Part and class information syntax

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

1. To specify a part, use a block delimited by `//VBBeginPartInfo:` and `//VBEndPartInfo:` statements.
2. The name on the `//VBBeginPartInfo:` statement and the name on the `//VBEndPartInfo:` statement must match.
3. 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.
4. All part names must be unique and must be valid COBOL class names.
5. Attribute, event and action information statements can appear more than once for a specific part, but all other information statements must appear only once.
6. The valid part information statements for a part specification are:

   ```
   //VBParent:
   //VBCopy:
   //VBPartDataFile:
   //VBLibFile:
   //VBComposerInfo:
   //VBEvent:
   //VBAttribute:
   //VBAction:
   //VBPreferredFeatures:
   //VB:
   ```

For information about how to read these syntax diagrams, see “How to read the 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 COBOL class name that implements the part interface.

description
Optional part description.

VBParent statement for a part

The //VBParents: statement describes the part's parent part. The attributes, actions, and events defined by the parent part are inherited. The part class hierarchy must include the CStandardNotifier class for nonvisual parts.

```
►►─┬───────────────────────────┬──────────────────────────────────────►◄
   └─//VBParent:──parent_name──┘
```

parent_name
Valid and unique COBOL class name.

VBCopy statement for a part

The //VBCopy: statement specifies the `COPY` file to be used in the `REPOSITORY` paragraph to specify external class references.

```
                ┌─,──────────────┐
►►──//VBCopy:───▼─copy_file_name─┴──────────────────────────────────►◄
```

copy_file_name
Name of copy file.

VBPartDataFile statement for a part

The //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 .VCB extension.

```
►►─┬──────────────────────────────┬───────────────────────────────────►◄
   └─//VBPartDataFile:──file_name─┘
```

file_name
File name in which to save the part interface information.

VBLibFile statement for a part

The //VBLibFile: statement specifies either the .OBJ file for the part, or the import library for the .DLL file that contains the part. Visual Builder uses this information to link to other parts. It does this by adding the file name to the dependency list and the ilink command in the make file generated for applications using the part.

```
►►─┬─────────────────────────┬────────────────────────────────────────►◄
   └─//VBLibFile:──file_name─┘
```

file_name
Name of the .OBJ file that will contain the compiled nonvisual part, or of the import library (.IMP or .LIB) for the .DLL file that contains the compiled nonvisual part.

VBComposerInfo statement for a part

The part //VBComposerInfo: statement specifies the information needed to support visual and nonvisual parts.

```
►►──//VBComposerInfo:──┬─VisualPart────┬──────────────────────────────────►
                       ├─NonvisualPart─┤
                       ├─Class─────────┤
                       └─Programs──────┘

►──┬────────────────────────────────────┬─────────────────────────────────►◄
   └─,──resource_id──,──resource_dll──┘
```

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.

VBEvent statement for a part

The //VBEvent: statement specifies the event information. This information includes the event name, event description, method to get the event ID, method to get the event data, and the get method return parameter name. 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──,──┬─────────────────┬──,──method_to_get_id──────►
                               └─"description"─┘

►──┬─────────────────────────────────────────────────┬───────────────────────►◄
   └─,─method_to_get_event_data──,──parm_name──┘
```

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

description
: Optional event description.

method_to_get_id
: COBOL method to get the event ID.

method_to_get_event_data
: COBOL method to get the event data.

parm_name
: get method's return parameter. A parameter has the form:

```
01 aParm aType.
```

where `aType` is either a type provided by Visual Builder, or a type loaded into Visual Builder. The type definition information to load the type into Visual Builder may be contained in the same file, or it may be loaded separately. Multiple parameters must be separated by a blank.

VBAttribute statement for a part

The //VBAttribute: statement specifies the attribute information. This information includes the attribute name, attribute description, get method name, get method return parameter name, set method name, set method return parameter name, and event ID. Use this statement to describe attributes implemented by the part that are useful in making connections.

```
►►──//VBAttribute:──attribute_name──,──┬───────────────┬──────────────────────────►
                                       └─"description"─┘

►──,──get_method_name──,──get_method_parm─────────────────────────────────────────►

►──┬─────────────────────────────────────────────────────────────┬────────────────►◄
   └─,──set_method_name──,──set_method_parm──┬───────────────┬───┘
                                             └─,──event_id───┘
```

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

description
: Optional attribute description.

get_method_name
: COBOL method to get the attribute data.

get_method_parm
: Get method's return parameter. A parameter has the form:

```
01 aParm aType.
```

`aType` is either a type provided by Visual Builder, or a type loaded into Visual Builder. The type definition information to load the type into Visual Builder may be contained in the same file, or it may be loaded separately. Multiple parameters must be separated by a blank.

set_method_name
: Optional COBOL method to set the attribute data.

set_method_parm
: Set method's set parameter. A parameter has the form:

```
01 aParm aType.
```

`aType` is either a type provided by Visual Builder, or a type loaded into Visual Builder. The type definition information to load the type into Visual Builder may be contained in the same file, or it may be loaded separately. Multiple parameters must be separated by a blank.

event_id
: Optional event ID associated with the attribute.

VBAction statement for a part

The //VBAction: statement specifies the action information. This information includes the action name, action description, action parameters and return parameters. Use this statement to describe actions implemented by the part that are useful in making connections.

```
►►──//VBAction:──action_name───────────────────────────────────────────────►

►───┬──────────────────────────────────────────────────────────────┬────────►◄
    └─,──┬───────────────┬──┬──────────────────────────────────┬───┘
         └─"description"─┘  └─,──┬────────────┬──┬─────────────┬┘
                                 └─parameters─┘  └─,return_parm─┘
```

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

description
Optional action description.

parameters
Parameters for the action. A parameter has the form:

```
01 aParm aType.
```

`aType` is either a type provided by Visual Builder, or a type loaded into Visual Builder. The type definition information to load the type into Visual Builder may be contained in the same file, or it may be loaded separately. Multiple parameters must be separated by a blank.

return_parm
Return parameter name.

VBGeneratorValues statement for a part

The //VBGeneratorValues: statement specifies the files to include during code generation and the build options to use during the build process.

```
                          ┌─,───────────────────────────────────────┐
►►──//VBGeneratorValues:──▼─┬─genCBL──("──source_name──")──────────┬─┴───────►◄
                            ├─userCPY──("──repository_copy──")─────┤
                            ├─userCBV──("──feature_source──")──────┤
                            ├─userCPV──("──feature_declaration──")─┤
                            ├─startingResId──(──start_ID──)────────┤
                            └─genMake──┬─"exe"─┬───────────────────┘
                                       └─"dll"─┘
```

source_name
Name of the part source file.

repository_copy
Name of copy file to include in the repository section of the class.

feature_source
Name of the file containing methods to include in the part source.

feature_declaration
Name of the copy file to include in the working storage section of the class.

start_ID
Starting resource ID.

VBPreferredFeatures statement for a part

The //VBPreferredFeatures: statement specifies the preferred part features. If you do not use this statement, the parent part's preferred list is used. If you do use this statement, the parent 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 //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.

```
►►──┬────────┬────────────────────────────────────────────────────────────►◄
    └─//VB:──┘
```

VBEndPartInfo statement for a part

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

```
►►──//VBEndPartInfo:──part_name────────────────────────────────────────────►◄
```

part_name
COBOL class name matching the previous //VBBeginPartInfo: statement.

Part information file example

The following example shows a part information file for a sample nonvisual part:

```
//VBBeginPartInfo: sampnonvis,"A sample non-visual part"
//VBParents: CStandardNotifier
//VBPartDataFile: sampnv.vcb
//VBComposerInfo: nonvisual
//VBEvent: ready,"ready"
//VB:       ,getReadyId
//VBEvent: destroy,"destroy"
//VB:       ,getDestroyId
//VBEvent: ChangeData,
//VB:    "This is a sample event"
//VB:       ,getChangeDataId
//VBAction: getCurrentDate,
//VB:    "This action returns the current date as a string"
//VB:    ,,01 Tday VarLengthString.
//VBAction: getData,
//VB:    "This action is passed an integer and returns current date"
//VB:    ,01 DIndex Integer.
//VB:     01 Counter Integer.
//VB:    ,01 Results VarLengthString.
//VBAttribute: ThisTime,
//VB:    "This holds the current time as an integer"
//VB:       ,getThisTime,01 ThisTime Integer.
//VB:       ,setThisTime,01 ThisTime Integer.
//VB:       ,ThisTimeId
//VBAttribute: LastName,
//VB:    "This holds the last name of the person as a string"
//VB:       ,getLastName,01 LastName VarLengthString.
//VB:       ,setLastName,01 LastName VarLengthString.
//VB:       ,LastNameId
//VBAttribute: Age,
//VB:    "This holds the age of the person as an integer"
//VB:       ,getAge,01 Age Integer.
//VB:       ,setAge,01 Age Integer.
//VB:       ,AgeId
//VBPreferredFeatures: Age, LastName
//VBEndPartInfo: sampnonvis
```

Enumeration information syntax

This section describes the interface information for enumerations used by Visual Builder. Syntax descriptions appear in the recommended order of occurrence in a file. The following rules apply to the interface information for enumerations:

1. To specify an enumeration, use a block delimited by `//VBBeginEnumeration:` and `//VBEndEnumeration:` statements.
2. The name on the `//VBBeginEnumeration:` statement and the name on the `//VBEndEnumeration:` statement must match.
3. All enumeration names must be unique and must be valid COBOL class names.
4. Each information statement can appear only once.
5. The valid enumeration information statements for an enumeration specification are:

```
//VBCopy:
//VBPartDataFile:
//VBComposerInfo:
//VB:
```

For information about how to read these syntax diagrams, see “How to read the syntax diagrams” on page xii

VBBeginEnumeration statement

The first statement describing the interface is the //VBBeginEnumeration: statement. This statement specifies the enumeration name and description.

```
►►──//VBBeginEnumeration:──enum_name──┬─────────────────┬──┬──────────────┬──►◄
                                      └─,"description"─┘  └─,declaration─┘
```

enum_name
Fully qualified enumeration name.

description
Optional enumeration description.

declaration
Optional enumeration declaration. If the declaration is omitted then the enumeration specification must include a `//VBCopy:` statement.

VBCopy statement for an enumeration

The //VBCopy: statement describes the copybook that contains the definition of the enumeration.

```
               ┌─,─────────────┐
►►──//VBCopy:──▼─copy_file_name─┴──────────────────────────────────────►◄
```

copy_file_name
The copybook containing the enumeration definition 88-level record items.

VBPartDataFile statement for an enumeration

The //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 .VCB extension.

```
►►──┬─────────────────────────────┬─────────────────────────────────────►◄
    └─//VBPartDataFile:──file_name─┘
```

file_name
File name in which to save the enumeration interface information.

VB statement for an enumeration

The //VB: statement allows interface information to be continued on the next statement line. This can be useful when specifying 88-level items. You can put each 88-level item on a separate //VB: statement.

```
►►──┬───────┬──────────────────────────────────────────────────────────►◄
    └─//VB:─┘
```

VBEndEnumeration statement

The //VBEndEnumeration: statement specifies the end of the enumeration interface information.

```
►►──//VBEndEnumeration:──enum_name─────────────────────────────────────►◄
```

enum_name
Enumeration name matching the previous //VBBeginEnumeration statement:.

Enumeration information example

The following example shows a basic enumeration definition:

```
//VBBeginEnumeration: DataStatus,"Codes for date retrieval",
//VB:    01  DataStatus PIC 9(9) COMP-5.
//VB:        88 DataStatus-NOTFOUND VALUE 0.
//VB:        88 DataStatus-FOUND    VALUE 1.
//VB:        88 DataStatus-ERROR    VALUE 9.
//VBEndEnumeration: DataStatus
```

If the associated part definition for the program specifies the following action information:

```
//VBAction: getRecord,"Get a record",
//VB:       01 aRec Name. 01 aFlag DataStatus.
```

then the Visual Builder generates the following code:

```
          LOCAL-STORAGE SECTION.
          01  aRec.
              COPY "record.cpy"
          01  aFlag PIC 9(9) COMP-5.
              88 aFlag-NOTFOUND VALUE 0.
              88 aFlag-FOUND    VALUE 1.
              88 aFLAG-ERROR    VALUE 9.
  ⋮
          CALL "getRecord" USING aRec, aFlag.
```

Type definition information syntax

This section describes the interface information for type definitions used by Visual Builder. Syntax descriptions appear in the recommended order of occurrence in a file. The following rules apply to the interface information for types:

1. To specify a type, use a block delimited by `//VBBeginTypeInfo:` and `//VBEndTypeInfo:` statements.
2. The name on the `//VBBeginTypeInfo:` statement and the name on the `//VBEndTypeInfo:` statement must match.
3. Each information statement can appear only once.
4. The valid type information statements for a type specification are:

   ```
   //VBCopy:
   //VBPartDataFile:
   //VB:
   ```

For information about how to read these syntax diagrams, see "How to read the syntax diagrams" on page xii

VBBeginTypeInfo statement

The first statement describing the interface is the //VBBeginTypeInfo: statement. This statement specifies the type definition name and description.

```
►►──//VBBeginTypeInfo:──typedef_name──┬─────────────────┬──┬──────────────┬──►◄
                                      └─,"description"──┘  └─,declaration─┘
```

typedef_name
: Type definition name used by Visual Builder.

description
: Optional type definition description.

declaration
: Optional type declaration. This is a normal COBOL declaration, without column rule restrictions. If the declaration is omitted then the type specification must include a `//VBCopy:` statement.

VBCopy statement for a type definition

The //VBCopy: statement describes the copybook that contains the higher level items of the type definition.

```
►►──//VBCopy:──copy_file_name──────────────────────────────────────────►◄
```

copy_file_name
: The copybook containing the type definition.

VBPartDataFile statement for a type

The //VBPartDataFile: statement specifies the file in which to save the type information. If you do not use this statement the imported file name is used with a .VCB extension.

```
►►──┬─────────────────────────────┬────────────────────────────────────►◄
    └─//VBPartDataFile:──file_name─┘
```

file_name
File name in which to save the type information.

VB statement for a type definition

The //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:─┘
```

VBEndTypeInfo statement

The //VBEndTypeInfo: statement specifies the end of the type definition interface information.

```
►►──//VBEndTypeInfo:──typedef_name──────────────────────────────────►◄
```

typedef_name
Type definition name matching the previous //VBBeginType: statement.

Type information example

The following example shows a basic type definition:

```
//VBBeginTypeInfo: Name,"Name type example",
//VB:    01  Name.
//VB:        03 Name-FIRST-NAME     PIC X(15).
//VB:        03 Name-MIDDLE-INITIAL PIC X(1).
//VB:        03 Name-LAST-NAME      PIC X(15).
//VBEndTypeInfo: Name
```

The type information provided is used by Visual Builder to generate code for actions. With the following action information specified in the associated part definition for the program:

```
//VBAction: getName,"Return the name",,01 NT Name.
```

the Visual Builder generates the following code:

```
          LOCAL-STORAGE SECTION.
          01  NT.
              03 NT-FIRST-NAME     PIC X(15).
              03 NT-MIDDLE-INITIAL PIC X(1).
              03 NT-LAST-NAME      PIC X(15).
⋮
          CALL "getName" RETURNING NT.
```

You can also specify type definitions using COPY files. Using a COPY file the previous type definition becomes:

```
   //VBBeginTypeInfo: Name,"Name type example"
   //VBCopy: NameType.cpy
   //VBEndTypeInfo: Name
```

where NameType.cpy contains:

```
              03 NT-FIRST-NAME  PIC X(15).
              03 NT-MIDDLE-NAME PIC X(1).
              03 NT-LAST-NAME   PIC X(15).
```

In this case the generated code is:

```
          LOCAL-STORAGE SECTION.
          01  NT.
              COPY "NameType.cpy".
⋮
```

Glossary

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.

action. A specification of a function that a part can perform. The Visual Builder uses action specifications to connect parts. Actions are resolved to method invocations in the generated code.

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 connect parts. Attributes are resolved to get and set methods in generated code.

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.

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.

B

behavior. The set of external characteristics that an object exhibits.

bounding box. An imaginary area encompassing three or more parts selected on the free-form surface.

C

caller. An object that invokes a method in another object or calls a program.

category. In the Composition Editor, a selectable grouping of parts represented by an icon in the left-most column of the parts palette. Selecting a category displays the parts belonging to that category in the next column.

class. An aggregate that can contain methods and data. Classes can be defined hierarchically, allowing one class to be a parent of another.

class hierarchy. A tree-like structure showing relationships among object classes. It places one abstract class at the top and one or more layers of less abstract classes below it.

class library. A collection of classes.

client area object. An intermediate window between a frame window (CFrameWindow) and its controls and other child windows.

client object. An object that requests services from other objects.

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.

Composition Editor. A view that is used to build a graphical user interface and to make connections between parts.

concrete class. A subclass of an abstract class that is a specialization of the abstract class.

connection. A relationship between parts that associates attributes, events, and actions of one part to those of another part. The Visual Builder generates the code that then implements these connections.

construction from parts. A software development technology in which applications are assembled from existing and reusable software components, known as parts.

cursored emphasis. When the selection cursor is on a choice, that choice has cursored emphasis.

D

data abstraction. A data type with a private representation and a public set of operations. The COBOL language uses the concept of classes to implement data abstraction.

declaration. A description that makes an external object or function available to a function or a block.

derivation. The creation of a new class from a parent class.

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.

event-to-action connection. A connection that causes an action to be performed when an event occurs.

event-to-attribute connection. A connection that results in the change of the value of an attribute when a certain event occurs.

F

feature. A major component of a software product that can be installed separately. 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 method, a set method, 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

get method. The method used to obtain the data in an attribute.

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.

H

handles. Small squares that appear on the corners of a selected visual part in the visual builder. Handles are used to resize parts.

I

inheritance. A mechanism by which a class can use the attributes and methods defined in more abstract classes related to it (its parent classes). An object-oriented programming technique that allows you to use existing classes as parents for creating other classes.

instance data. 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 methods of the defining class and its subclasses.

L

legacy code. Existing code that a user might have. Legacy applications often have character-based, nongraphical user interfaces.

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.

M

main part. The part that users see when they start an application. This is the part from which the .app file for the application is generated.

The main part is a special kind of composite part.

method. A procedure that is included in a class. A method has access to instance data of its class.

method invoke. A communication from one object to another that requests the receiving object to execute a method.

A method invocation consists of a method name that indicates the requested method, the parameters to be used in executing the method, and, if required, a return variable.

method name. The component of a method call that specifies the requested operation.

model. A nonvisual part that represents the state and behavior of a object, such as a customer or an account.

module definition file. A file that describes the code segments within a load module.

N

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.

no-event attribute. An attribute that does not have an event identifier.

no-set attribute. An attribute that does not have a set method.

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. A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. A collection of data and methods 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 methods. Each object in the class is said to be an instance of the class. An instance of a class. It can represent a person, place, thing, event, or concept. Each instance has the same properties, attributes, and methods as other instances of the class, though it has unique values assigned to its attributes.

object class. A template for defining the attributes and methods of an object. An object class can contain other object classes. An individual representation of an object class is called an object.

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 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 method or service that can be requested of an object.

P

parent class. A class from which other classes or parts are derived. A parent class may itself be derived from another parent class.

parameter connection. A connection that satisfies a parameter of an action or method by supplying either an attribute's value or the return value of an action or method. The parameter is always the source of the connection.

parent class. The class from which another part or class inherits data, methods, or both.

parameter. A data element in the USING clause of a method call. Arguments provide additional information that the invoked method can use to perform the requested operation.

part. A self-contained software object with a standardized public interface, consisting of a set of external features (actions, attributes, and events) that allow the part to interact with other parts. A part is implemented as a class that supports the CNotifier protocol and has a part interface defined.

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 data item 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.

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.

program. That part of a file containing a set of instructions conforming to COBOL language syntax. A self-contained, executable module. Multiple copies of the same program can be run in different processes.

promote. Features of a subpart are made accessible through features of the containing part.

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.

R

receiver. The object that receives a method invocation.

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.

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.

set method. The method used to update the data of an attribute.

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.

subpart. A part instance contained in a part.

System Interface Editor. The editor you use to specify the names of files that Visual Builder writes to when you generate code. You can also use this editor to do the following:

- Enter a description of the part
- Specify a different .VCB file in which to store the part
- See the name of the part's parent class

- 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

T

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

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). The hardware, software, or both that enable a user to interact with a computer. The term *user interface* normally refers to the visual presentation and its underlying software with which a user interacts.

V

view. A visual part, such as a window, push button, or entry field. 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.

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.

visual programming tool. A tool that provides a means for specifying programs graphically. Application programmers write applications by manipulating graphical representations of components.

W

white space. Space characters, tab characters, form-feed characters, and new-line characters.

window. 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. In the Composition Editor, a window is a part that can be used as a container for other visual parts, such as push buttons.

Related information

The following publications can help you find more information on OO software design and Common User Access (CUA) user interface design.

- Booch, Grady. *Object Oriented Design with Applications*.

  This book explores OO concepts and design. It shows you, through various techniques and examples, how to develop applications using OO.

  It is available through Benjamin/Cummings, Redwood City, California. Its ISBN number is 0-8053-0091-0.

- Cox, Brad J. *Object-Oriented Programming: An Evolutionary Approach*.

  This book explores OO programming. It shows you how to build reusable objects through examples and provides a comparison between OO programming and traditional programming.

  It is available through the Addison-Wesley Publishing Company, Reading, Massachusetts. Its ISBN number is 0-201-10393-1.

User Interface

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

  This book describes the guidelines that define the Common User Access user interface. The Common User Access user interface is an OO graphical user interface that provides a consistent look and feel for products that adopt the CUA interface as their standard.

  It is available through QUE Corporation, Carmel, Indiana. Its ISBN number is 1-56529-170-0.

- Rubenstein, R. and Hersch, H. *The Human Factor: Designing Computer Systems For People*.

  This book discusses user-centered design in its full scope: task analysis, prototyping, empirical evaluating, interface techniques, and guidelines.

  It is available through Digital Press, Maynard, Massachusetts.

Index

We'd Like to Hear from You

IBM VisualAge COBOL
Visual Builder User's Guide

Publication No. SC26-9053-02

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB2VVG at IBMMAIL
  - IBMLink: COBPUBS at STLVM27
  - Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

IBM VisualAge COBOL
Visual Builder User’s Guide

Publication No. SC26-9053-02

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name	Address
Company or Organization	
Phone No.	

Readers' Comments
SC26-9053-02

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

SC26-9053-02

Cut or Fold
Along Line

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-9053-02

Spine information:

IBM VisualAge COBOL

Visual Builder User's Guide