VisualAge C++ Professional for AIX

Visual Composition Tasks and Concepts

Version 5.0

Before using this information and the product it supports, be sure to read the general information under .

Second Edition (Feburary 2000)

This edition applies to Version 5.0 of the VisualAge C++ product, 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.

Contents

About this Book

This Portable Document Format (PDF) file is a collection of some of the concepts and tasks selected from the online information for this product. PDF files give you an easier way to review or print a group of related topics.

References to VisualAge C++ in this book should be interpreted as VisualAge C++, Version 5.0.

Chapter 1. Visual Programming Fundamentals

VisualAge for C++ includes a state-of-the-art object-oriented visual composition editor for assembling applications visually from IBM Open Class components.

Object-oriented programming facilitates development of complex software systems by breaking them up into a number of much smaller, simpler program elements called *objects*. Objects work together by sending each other *messages*, that is, by requesting behavior that is implemented by the target object. Taken as a group, these behaviors comprise a *class interface*.

Using an object-oriented approach for complex systems provides the following benefits:

- Individual classes are much easier to create and understand.
- Systems are much easier to maintain and enhance. Object implementations can be modified individually without modifying the rest of the system, as long as the objects continue to respond appropriately to messages sent to them by other objects.

Despite these benefits, implementing large systems can still be expensive. One way to reduce the cost is to reuse object implementations. Many companies would prefer to buy reliable reusable classes, creating classes only for functions specific to their business. This vision of constructing custom software using standard building blocks has been called *construction from parts*. The building blocks themselves have popularly been called *parts* or *components*.

However, reuse is hard to achieve when the class interfaces are too specific to the application for which they were originally developed. To promote wider reuse, class interface conventions called *component models* have been defined, such as ActiveX, OpenDoc, and JavaBeans.

The IBM notification framework of the IBM Open Class library is the component model used by VisualAge. The IBM notification framework includes the following definitions:

> **An event model.** Event models specify how a component sends messages to other objects without knowing the exact methods that the other object implements. This enables a component to be reused with a range of objects that have different interfaces
>
> **Events, attributes, and actions.** The IBM notification framework defines a component interface in terms of the events it can signal, the attribute values that can be read and set, and the actions it implements. This definition provides more structure to the interface of a component compared with a simple class interface, facilitating the use of tools such as the VisualAge Composition Editor.

The Composition Editor enables you to create programs graphically from existing parts. Parts are simply C++ classes that comply with the IBM notification framework. The IBM notification framework is the component model supported and used throughout VisualAge, so this documentation will refer to VisualAge components as *parts*.

VisualAge provides user interface parts based on classes in the IBM Open Class library. The Composition Editor is also extensible. It allows you to work with parts you create yourself, and it allows you to include parts imported into the environment from other sources. You can even create your own parts graphically using the Composition Editor and then reuse these parts again within another program being created with the Composition Editor.

To build a program with the Composition Editor, you draw a picture using a canvas and palette of icons representing reusable parts. This picture specifies the set of parts that implements the function of the larger program (or part) you are creating. For parts like user interface controls, the position of the controls relative to each other in the picture specifies how the controls will appear in the final program. For parts such as database components, the position of the part in the picture generally has no significance.

The Composition Editor provides a very sophisticated connection capability to specify how components of the picture will interact to implement functions of the program. Using connections, much of the behavior of an application can be specified graphically. Connections also allow you to integrate custom code written in the C++ language.

RELATED CONCEPTS

“Chapter 2. How Classes and Parts Are Related” on page 3

“Chapter 5. Visual Composition” on page 9

“Chapter 24. Part Design Guidelines” on page 97

Chapter 2. How Classes and Parts Are Related

VisualAge parts are C++ classes that conform to the IBM notification framework. *Composite* parts are made up of embedded parts. We use the term *part* to refer to both a class and its instances, as follows:

- When we refer to parts on a palette or to parts that you create by writing code, we mean part **classes**.
- When we refer to parts on the free-form surface or to parts that are connected, we mean part **instances**.

During visual composition, you interact with part interfaces. The most useful part interfaces contain the following features:

Access to data, or *attributes*. A complete attribute interface includes member functions to return the value of the attribute, to set the value of the attribute, and to notify other parts when the value of the attribute changes. The interface for a attribute does not have to be complete. For example, a attribute might be read-only, in which case the interface would not support the ability to set the value of an attribute. A attribute can be any of the following:

– An actual data object stored within the part, such as the street in an address part

– A computed data, such as the sum of all numbers in an array or the profit that is computed by subtracting dealer cost from the retail price

Access to the behavior of a part, or *actions*. These represent tasks you can ask a part to perform, such as open a window or add an object to a collection of objects.

Event notification. By signaling events, a part can notify other parts that its state has changed. For example, a push button can signal an event to notify other objects when it is clicked, or a window can signal an event when it is opened, or a bank account can signal an event when the balance becomes negative.

Events can also be signaled when the value of a part attribute changes, such as when money is deposited into or withdrawn from a bank account.

RELATED CONCEPTS

"Chapter 1. Visual Programming Fundamentals" on page 1

"Chapter 28. VisualAge Component Model" on page 119

"Chapter 5. Visual Composition" on page 9

"Chapter 24. Part Design Guidelines" on page 97

Chapter 3. Visual, Nonvisual, and Composite Parts

You can use many kinds of parts to construct program elements. All parts exist as either primitives or composites. Primitive parts are the basic building blocks from which composites are constructed. You must construct new primitive parts using a programming language because there are no similar parts to use in building them. Primitive parts can be either visual or nonvisual.

Visual parts are elements of the program that the user can see at run time. The development-time representations of visual parts in the Composition Editor closely match their runtime visual forms. Users can edit these parts in the Composition Editor in their visual runtime forms. Examples include windows, entry fields, and push buttons. In general, visual parts are subclasses of IWindow.

Nonvisual parts are elements of the program that are not necessarily seen by the user at run time. On the Composition Editor’s free-form surface, users can manipulate these parts only as icons. Examples include business logic, database queries, and communication access protocol parts.

Parts that have a visual representation at run time but do not support visual editing are treated as nonvisual. Examples of this kind of nonvisual part include message boxes and file selection dialogs.

Composite parts can contain both visual and nonvisual components. In general, composite parts are based on one of these classes:

- IFrameWindow, for GUI applications
- One of several ICanvas base classes, for reusable GUI client areas

RELATED CONCEPTS

“Chapter 2. How Classes and Parts Are Related” on page 3

“Chapter 5. Visual Composition” on page 9

“Chapter 4. Primary Part” on page 7

“Chapter 28. VisualAge Component Model” on page 119

RELATED TASKS

“Chapter 21. Incorporating User-Written Code into Visual Composites” on page 83

Chapter 4. Primary Part

On the Composition Editor's free-form surface, this part represents the composite's base class. For example, if your composite part inherits from IFrameWindow, VisualAge adds an instance for you when you create the class. All visual composite parts must have a primary part; otherwise, VisualAge cannot generate the correct code.

You can, however, change the primary part under certain circumstances. If the composite part in question inherits from IFrameWindow and you later add another IFrameWindow* instance, you can designate the new instance as the primary part instead of the one originally assigned by VisualAge.

RELATED CONCEPTS

"Chapter 3. Visual, Nonvisual, and Composite Parts" on page 5

"Chapter 5. Visual Composition" on page 9

"Chapter 28. VisualAge Component Model" on page 119

RELATED TASKS

"Chapter 21. Incorporating User-Written Code into Visual Composites" on page 83

Chapter 5. Visual Composition

Visual composition is the creation of object-oriented programs by manipulating graphical representations of components. VisualAge provides a powerful tool, the Composition Editor, that enables you to construct programs visually.

In the Composition Editor, you select and place parts to create graphical user interfaces (GUIs). These GUIs can include VisualAge parts, imported parts, and parts you create yourself. By following a few guidelines, you can design versatile parts that you can reuse in many compositions. VisualAge also enables you to use nonvisual parts to perform the business logic and data access.

Development using visual composition can include the following steps:

1. Design your program elements. Determine what you can compose visually and what you must write by hand.
2. Create nonvisual parts.
3. Using the Composition Editor, enhance these classes by dropping parts and setting initial values for attributes. Extend the behavior of VisualAge parts by writing member functions.
4. For business logic, add code to the appropriate class stubs.
5. Connect parts to define the program element’s behavior and flow.
6. Save and generate the code for your part within the Composition Editor, and compile your application.

RELATED CONCEPTS

“Chapter 1. Visual Programming Fundamentals” on page 1

“Chapter 2. How Classes and Parts Are Related” on page 3

“Chapter 3. Visual, Nonvisual, and Composite Parts” on page 5

“Free-form Surface”

“Parts Palette” on page 10

“About Adding Parts in the Composition Editor” on page 11

“About Setting Groups and Tab Stops” on page 12

“About Tearing Off Attributes” on page 13

RELATED TASKS

“Chapter 9. Working with Parts in the Composition Editor” on page 25

“Chapter 6. Starting the Visual Builder in AIX” on page 15

Free-form Surface

The *free-form surface* is the large open area in the Composition Editor. It is like a blank sheet of paper or work area where you can add, manipulate, and connect the parts that you work with to create your composite part.

The free-form surface represents the part you are developing. It is separate from its parent part, which is represented by the primary part or base class. Options for this part, such as pop-up menu items, do not refer to the inherited features. Access inhereted features through the primary part.

Some of the functions you can perform on the free-form surface include:

- Add visual parts.
- Add nonvisual parts to build the application logic for a composite part.
- Delete parts.
- Connect parts to define behavior.

RELATED TASKS

"Chapter 9. Working with Parts in the Composition Editor" on page 25

Parts Palette

The *parts palette* provides building blocks you can use to construct a program element. It consists of several categories, each one containing a set of parts. You can add a part to your program element by selecting a category from the palette and then a part and dropping it on the free-form surface. The status area displays the name of the category and the part you select from the palette.

The parts palette can contain the following:

- Parts supplied with VisualAge
- Parts you have constructed yourself
- Parts supplied by a vendor

Typically, a *category* contains parts that are similar in behavior or that you can use to perform similar tasks. For example, the **Frame extensions** category contains parts such as menus, menu items, menu separators, and cascaded menu items. You can use all of these to add menus to an application.

Typically, a *category* contains parts that are similar in behavior or that you can use to perform similar tasks. Often you can use the parts in a category to assemble a particular application. However, parts can be in more than one category.

You can also modify the palette by adding parts. Modifying the palette can help increase your productivity in the following ways:

- Allows you to quickly and easily place parts that you have created and that you use often in the Composition Editor.
- Eliminates the need for manually placing the parts, which requires that you know the exact class name of the part.

When you add a new part to the palette, the entire visual part, class interface part, or nonvisual part is represented with a symbol in the category you select. When you originally created the part, you specified its name in the **Name** field. The name you specified appears on the parts palette. Once you have added parts to the palette, you can place these parts on the free-form surface in the same way you place parts that VisualAge provides.

RELATED TASKS

"Dropping a Part that Appears on the Palette" on page 33

"Chapter 18. Managing the Parts Palette" on page 69

About Adding Parts in the Composition Editor

When you place parts in the Composition Editor:

- Avoid overlaying parts

  It is not good interface design for one part to overlay another part. Completely or partially overlaying a part can result in focus problems, causing users to see but be unable to select the part.

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

  Parts included in the Composers category can contain any other visual parts that are placed on top of them. The parts contained by the Composers part become composites of the Composers part. For example, if you place an entry field, a list box, and two push buttons in a frame window, the frame window contains these parts and they in turn become the frame window’s embedded parts.

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

Table 1. Categories and How You Can Use Their Parts

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

- Embed composite parts into other composites

  For example, you can create a composite part that consists of a panel on which you have placed buttons and check boxes, each connected to a variable. When you embed this part in your main program, such as in a frame window, you place it and work with it as one part—not as a panel and separate buttons and check boxes.

  Although the connections and variables for an embedded composite part are not displayed when that part is embedded, they are there. You do not see them because you do not edit them or their embedded parts directly within the main composite part. Using embedded composite parts reduces the number of connections and simplifies your work space.

  To change the connections, the default text on the buttons and check boxes, or to otherwise alter this part, you must edit the part indirectly, as described in “Editing Parts within a Composite Part” on page 29.

RELATED TASKS

“Chapter 10. Dropping Parts in the Composition Editor” on page 33

“Chapter 6. Starting the Visual Builder in AIX” on page 15

Property Sheets

Use the part property sheet to change the initial appearance and operation of parts.

The part property sheet contains two columns: a list of properties on the left and their values on the right. The value column contains the current values and is where you edit the properties. Typical part properties include:

Name There are two name-type properties. The *label* property is for the text that appears on parts such as buttons. The *partName* property is the name by which this part is identified.

Style The property sheet includes several control settings that enable you to specify style information for the part. These controls might include tab stop and group.

Color You can modify both background color and foreground color.

Font You can specify the font, font size, and font highlight you use for the part.

RELATED TASKS

"Editing Part Properties" on page 27

"Opening the Property Sheet for a Part" on page 27

"Applying Changes to Properties" on page 28

"Using Code Strings in Part Properties" on page 28

About Setting Groups and Tab Stops

The tabbing order is the sequence in which the input focus moves from part to part as the user presses the Tab key. 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. 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.

The tabbing order can only be set for parts that are placed within a Composers part. For example, if you place a row of push buttons in a frame window, you can set the tabbing order for the push buttons.

The following guidelines are for setting groups and tab stops:

- The tabbing order of the parts should be the same as the order in which they are displayed in the window, from left to right and then top to bottom.
- Select both **Group** and **Tab stop** for Parts that are not in groups, such as entry fields and list boxes.
- Put each group of related parts, such as check boxes and radio buttons, within a group or outline box and select **Group** for the first part only. Select **Tab stop** for each part within the **Group** that you want to receive input focus. If there is only one group of related parts, such as push buttons, you do not need to put them within a group or outline box.
- Do not select either **Group** or **Tab stop** for Parts that should not receive input focus, such as static text parts.

All parts within a composite part that have only **Group** selected are treated as a single tabbing unit. When the user tabs within the composite part, input focus moves to the next tab stop whether it is within the group or not. When the user uses the keyboard arrow keys, the input focus moves between all parts within the group, regardless of which parts have **Tab stop** selected.

Grouped radio buttons are special in that 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, and RadioButton3 and RadioButton4 become mutually exclusive in their group.

You can set the group and tab stops from the pop-up menu. You can display the tabbing order by opening a parts list for the Composers part that contains the push buttons. You can change the tabbing order within the parts list, by changing the positions of parts.

If the tabbing order includes each entry field in which a user can type, the user can move the input focus from one entry field to another. 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. Read-only fields do not need to be included in the tabbing order.

As you add or rearrange parts, you will probably need to change the tabbing order. For example, rearrange three push buttons so that PushButton3 is between PushButton1 and PushButton2. The tabbing order is still PushButton1, PushButton2, PushButton3, even though PushButton3 is now between PushButton1 and PushButton2. You must change the order in the parts list to have the focus move from PushButton1, to PushButton3, and then PushButton2.

RELATED TASKS

"Changing the Tabbing Order" on page 42

"Setting Tab Stops and Groups" on page 43

"Changing Depth Order within a Composite Part" on page 42

About Tearing Off Attributes

You tear off an attribute to gain access to the encapsulated features of a part. This can be necessary when an attribute is in itself a part and you want to connect to one of its features. The torn-off attribute is not actually a separate part but a variable that represents the attribute itself or points to it.

For example, in an address book application you might tear off attributes as follows:

- You might have a Person part that contains both *homeAddress* and *workAddress* attributes, both of which, in turn, could contain *street*, *city*, and *state* attributes.
- By tearing off either the *homeAddress* or *workAddress* attribute, you can create a new part that contains *street*, *city*, and *state* attributes.
- Tearing off a *homeAddress* or *workAddress* attribute makes the nested *street*, *city*, and *state* attributes directly accessible. Now that the nested attributes are directly accessible, you can make connections to and from them, as well as to their associated events and actions.

RELATED TASKS

"Tearing Off Attributes" on page 43

Chapter 6. Starting the Visual Builder in AIX

To start a Visual Builder session in AIX, do the following:

1. Change to your data directory
2. At the prompt, enter `ivb` and the Visual Builder window appears.

Tasks you can perform from the Visual Builder window include:

- “Preloading the Visual Builder Window” on page 17
- “Creating a New Part” on page 17
- “Opening Parts in the Visual Builder Window” on page 18
- “Displaying Part Names” on page 17
- “Selecting All Parts” on page 17
- “Deselecting All Parts” on page 17
- “Copying Parts from One Part File to Another” on page 19
- “Moving Parts to a Different Part File” on page 20
- “Deleting Parts from a Part File” on page 20
- “Renaming Parts in Part Files” on page 21

RELATED TASKS

“Chapter 7. Working with Parts in the Visual Builder Window” on page 17

Chapter 7. Working with Parts in the Visual Builder Window

The topics in this section describe how to work with parts from the Visual Builder window.

Preloading the Visual Builder Window

In order to preload your part files into the Visual Builder window, create a `vbload.dat` file. This text file contains the names of the part files you wish to preload. Once you have created the file, place it in the working directory along with the actual part files. When you open the Visual Builder window, VisualAge automatically loads the listed files saving you the task of manually loading each file.

Managing your project is easier if you keep the `vbload.dat` file updated with all the project part file names. For an example of a `vbload.dat` file, refer to any of the samples provided with VisualAge.

Displaying Part Names

To display the names of the parts in a part file, from the **Loaded files** list box in the Visual Builder window, select the file containing the parts you want to see. Visual parts appear in the **Visual parts** list box; nonvisual parts and class interface parts in the **Nonvisual parts** list box.

Once part names are displayed, you can perform actions on them, such as opening or deleting them.

Selecting All Parts

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

If you want to deselect any of the parts, hold the Ctrl key and select the part name with mouse button 1.

Deselecting All Parts

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

Creating a New Part

To create a new part, do the following:

1. Select **Part—New** and then select Visual or Nonvisual part or Class. The Part–New window appears.
2. Enter the name that you want to give to your part in the **Class name** field.
3. Enter a description of the part in the **Description** field. Visual Builder uses the description that you enter here in the following ways:
   - If you add the part to the parts palette, Visual Builder displays its description in the information area at the bottom of the Composition Editor when the part is selected.

   - If you export the information about the part to a .vbe file, the description is included with the other information about the part.
4. In the **File name** field, enter the part file name where you want VisualAge to store the part. If the file does not exist, VisualAge creates it for you.
5. Select the type of part that you want to create in the **Part type** field. You can select one of the following:
   - Visual part
   - Nonvisual part
   - Class interface part
6. Either keep the default class name provided by Visual Builder in the **Base class** field, change it, or delete it.

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

Opening Parts in the Visual Builder Window

Use **Part** and then **Open** to open existing parts. You must load the part file that contains a part before you can open it. .

The Composition Editor uses the question mark icon on the free-form surface to represent unloaded parts. If you open a part that contains unloaded parts, Visual Builder displays this icon.

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

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

If you open a part and see a question mark folder icon, load the part file that contains the unloaded part.

Opening one part

To open one part, do the following:

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

   Note: If the list boxes are empty or if you cannot find the part, the part file is not loaded.
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, the Composition Editor appears.
   - If you are opening a nonvisual part, the Composition Editor appears.
   - If you are opening a class part, thePart Interface Editor appears.

You can also open an existing part by double-clicking the part name within the **Visual parts** or **Nonvisual parts** list box.

Opening multiple parts

To open multiple parts, do the following:

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

   Note: If the list boxes are empty, you must load the part files.
2. Select the first part you want to open.
3. Do one of the following, depending on how the other parts appear in the list:
   - If the other parts are adjacent in the list to the part previously selected, hold down the Shift key and click on the last part in the group you want to select. All parts between the first and last parts selected are now highlighted.
   - If the parts are not adjacent in the list, hold down the Ctrl key while selecting each part.
4. Select **Part** on the menu bar.
5. Select **Open** in the pull-down menu. Visual Builder displays a separate window for each part that you selected. The window displayed is the Composition Editor for visual/nonvisual parts or the Part Interface Editor for class parts.

Copying Parts from One Part File to Another

To copy a part, do the following:

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

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

Moving Parts to a Different Part File

Here is what happens to the part file where the part or parts are moved:

- If the designated part file does not exist, VisualAge creates and loads it for you.
- If the designated part file exists and is loaded, the part or parts are moved into it.
- If the designated part file exists but is not loaded, VisualAge overwrites the unloaded file with the new part.

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** and then **Move**.
3. Use the following instructions for moving one part or multiple parts:

   Moving one part

   If you selected one part, the Move Part Window appears.

   The **Part name** field displays the name of the selected part. The **File name** field displays the complete path of the part file for 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 where you want to move the part.
   b. Select the **Move** push button.

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

Moving multiple parts

If you selected more than one part, the Move Part Window for moving more than one partappears.

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

1. In the entry field, enter the name of the part file to where you want to move the parts. If the part file is not in your current directory, specify the complete path for the part file.
2. Select **OK** and the parts are moved to the part file specified in the entry field.

You can also move a part by changing the name of the part file specified in the Class Editor.

Deleting Parts from a Part File

To delete a part, do the following:

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

   If you do not select at least one part, the **Delete** function is not available.
2. Select **Part** and then **Delete**.

   Deselect any parts that you do not want to delete. Once you delete a part from a part file, you cannot recover it unless you have another copy stored in another part file.
3. Select the **Delete** push button.

   The selected parts are deleted.

Renaming Parts in Part Files

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

Note: Use care when renaming parts. The name changes only in the part file where the part is stored. The part name does not change in any other part where this part is embedded. Therefore, when you open the part where you embedded the renamed part, VisualAge 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** and then **Rename**. The **Rename** window appears.

   The **Part name** field shows the name of the part that you selected to rename.
3. In the **New part name** field, enter the new name for the part.
4. Select **Rename** and the part is renamed.

   RELATED TASKS

   “Chapter 6. Starting the Visual Builder in AIX” on page 15

Chapter 8. Using Parts from Previous Versions of VisualAge

To continue to use parts from Versions 3.0 and 3.5, perform these steps as required by your application:

- Make sure any member function signatures with input arguments specify both a type and a name for each argument. (Previously, you only had to specify a type.) If you run the migration utility on a part file in which only types are specified, VisualAge defines the name and type to be the same (for example, IDate and IDate).
- Run the migration utility on your part files (.vbb and .vbe). To do this, type the following, substituting the names of your files for the placeholders below:

  ```
  ivb -vbmigrate mypart1.vbb mypart2.vbe @mypart.lst
  ```

  As shown in the example, you can simply list the files separated by a space or create a list file. If you use the list-file method, the file name must be preceded by an at-sign (@). If you do not specify a file name, a file selection window appears.
- Clean up migrated parts that contain connections between the actionResult feature and a connection parameter.
- Any connections from promoted or embedded *ready* events are deleted by the migration utility, because they are no longer permitted. Add alternative connections as necessary.
- Edit handwritten code for updated features and changes in generated code.

RELATED CONCEPTS

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Using Handwritten Code from Previous Versions of VisualAge” on page 83

Chapter 9. Working with Parts in the Composition Editor

Parts make up the user interface and logic of a program. The Composition Editor provides many tools for visually creating these parts. Visually developing a user interface includes:

1. Designing the user interface.
2. Dropping parts on the free-form surface.
3. Changing properties and manipulating the parts.
4. Making connections to determine the parts behavior.
5. Manipulating connections.
6. Generating the code for the parts.
7. Generating source code for the main() function.
8. Compiling and linking the code.
9. Making changes.

When you edit the part that represents the overall structure of your application (usually its main user interface view), you graphically build your application. By making connections between parts, you build your program's business logic.

Building a program with the Composition Editor involves the following tasks:

- "Chapter 10. Dropping Parts in the Composition Editor" on page 33
- "Opening the Property Sheet for a Part" on page 27
- "Chapter 12. Making Connections in the Composition Editor" on page 45
- "Saving a Part" on page 32

RELATED CONCEPTS

"Chapter 5. Visual Composition" on page 9

"About Adding Parts in the Composition Editor" on page 11

"Generated Application Code" on page 112

RELATED TASKS

"Chapter 6. Starting the Visual Builder in AIX" on page 15

"Generating Source Code for Your Application's main( ) Function" on page 61

"Generating Source Code" on page 60

"Displaying Part Pop-Up Menus" on page 26

"Browsing a Part's Features" on page 26

"Editing Part Properties" on page 27

"Undoing and Redoing Changes in the Composition Editor" on page 29

"Editing Parts within a Composite Part" on page 29

"Renaming Parts in the Composition Editor" on page 30

"Listing Parts within a Composite Part" on page 31

"Saving a Part" on page 32

Displaying Part Pop-Up Menus

To see a menu of operations you can perform on a part, click mouse button 2 on the part. The pop-up menu for the part appears. Choices on the pop-up menu allow you to delete the part, rename it, and perform other operations (which vary, depending on the part).

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

To display pop-up menus for multiple parts:

- Select the parts.
- Place the mouse pointer over any of the selected parts.
- Click mouse button 2.

RELATED TASKS

"Selecting Parts" on page 35

Browsing a Part's Features

Sometimes it is useful to browse part features before using them in a connection. For example, you might want to look at an attribute to see if it has a set member function so that it can update itself when it receives new data from another attribute.

By using **Browse part features**, you can see all features of a part in one window and browse, but not change, the information about each feature. To modify a feature, use the Part Interface Editor.

There is an important distinction between browsing part features and displaying features for making a connection. When you browse part 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.) When you display the connection menu for a part, however, you see only those features that are available for connections.

To browse the features of a part, do the following:

1. Move the mouse pointer over the part and click mouse button 2. The Composition Editor displays the pop-up menu for the part.
2. Select **Browse part features**. The Composition Editor displays a browse window that contains three columns: actions, attributes, and events.
3. Select the feature you want to browse.

   The Composition Editor displays information about the feature that you select in the entry fields below the feature columns. The categories of information displayed for a part depend upon whether you select an action, an attribute, or an event.

   The information that the Composition Editor displays when you browse part features is the same as the information in the Part Interface Editor.

Editing Part Properties

The property sheet for a part provides a way to display and set initial attribute values for the part.

If a Boolean value does not behave the way you expect, its property value may be set to default (rather than true or false). This means that the part inherits its value from the parent window.

For properties in numeric entry fields, the Composition Editor reflects only those values entered in decimal notation. You can enter numbers with bases other than 10 by using the number sign (#, as in #0xBF), but the expression is not evaluated until compile time.

The Composition Editor does not evaluate properties that you enter as string expressions (text strings preceded by a number sign). User interface controls where size is determined by the length of the property value, are not displayed in the Composition Editor as they appear in the compiled application. An example would be the size of a push button that is determined by the length of its text string. However, the Composition Editor passes the expression correctly into the generated code so it compiles successfully. For example, suppose you enter `#pbText` as the value of a text attribute of an IPushButton* part. Because `#pbText` is not a literal value, the length of the text string is unknown and the push button appears small in the Composition Editor. If you have defined the value of `#pbText` elsewhere, the push button appears with the correct size and text string in the compiled application.

For help with filling out the property sheet for a part, open the help for that part.

RELATED CONCEPTS

“Property Sheets” on page 11

RELATED TASKS

“Opening the Property Sheet for a Part”

“Using Code Strings in Part Properties” on page 28

“Applying Changes to Properties” on page 28

Opening the Property Sheet for a Part

To change the properties for a single part, follow these steps:

1. Place the mouse pointer over the part and double-click mouse button 1. The property sheet for the part appears.
2. When you have chosen the property you wish to modify, select the value field to the right of the property name.
3. Make the appropriate changes from the provided options. Options for modifying the properties depend upon the part and may include selecting a button, entering information into the field, selecting from a drop-down list, or proceeding to other dialog boxes.

To change the properties for multiple parts, follow these steps:

1. Select the parts with properties 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. A property sheet for each of the selected parts appears.

   RELATED TASKS

   “Editing Part Properties” on page 27

   “Using Code Strings in Part Properties”

   “Applying Changes to Properties”

Using Code Strings in Part Properties

Many property sheets provide fields in which you can specify initial values for part properties. For example, the property sheet for the IEntryField* contains a property name for **Text** and **Limit**. In the value column of the property name for **Text**, you can enter a text string that you want to initially display in the entry field. You can change the default value of **Limit**, which is the maximum number of characters a user can type in the entry field, from `32` to the number you desire.

To facilitate national language support (NLS) translation and code changes in providing property values, such as the ones just described, you can enter a code string to provide those values. You must precede the code string with a number sign (#). If the first character of your code string is a #, enter two #s separated by a space:the first one to signify that a code string follows and the second one to begin your code string.

For example, suppose you want the initial text in an entry field to be `Enter a name here`. Further, suppose that you want the limit for this entry field to be 18 characters. In a user header file (.hpv or .h), you could insert the following #define statements:

```
#define NAME_PROMPT "Enter a name here"
#define NAME_LENGTH 18
```

Be sure to enter the name of the file that contains these #define statements in the **Required include files** field in the Class Editor. Otherwise, this file is not included when you generate the code for this part.

Then, in the property sheet of the IEntryField* you could enter the following in the **Text** and **Limit** value column, respectively:

```
#NAME_PROMPT
#NAME_LENGTH
```

By doing this, the values that you defined for NAME_PROMPT and NAME_LENGTH are used when you generate the source code for the part being edited.

RELATED TASKS

“Editing Part Properties” on page 27

“Opening the Property Sheet for a Part” on page 27

“Applying Changes to Properties”

Applying Changes to Properties

After you make changes to the properties, you can activate them in either of the following ways:

- To immediately activate and save the property changes and to close the property sheet, select the **OK** push button.
- To apply the property changes you have made and keep the property sheet open, select the **Apply** push button. This enables you to see whether you need to modify any of the changes you have made. The changes remain applied until you change them again.

To close the property sheet, select the **Cancel** push button. If you made changes and selected the **Apply** push button, the changes are saved.

For more help with using the property sheet, select the **Help** push button.

RELATED TASKS

"Editing Part Properties" on page 27

"Opening the Property Sheet for a Part" on page 27

"Using Code Strings in Part Properties" on page 28

Editing Part Labels Directly

Some visual parts, such as buttons and menus, contain text strings. To edit the text of a label for a part, follow these steps:

1. Hold down the Alt key. Hold down the Alt + Ctrl keys in AIX.
2. Select the text you want to change with mouse button 1.
3. Edit the text.
4. When you are finished, press Shift+Enter.

Note: Pressing the Esc key cancels direct edit without making changes.

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

Undoing and Redoing Changes in the Composition Editor

If you want to reverse changes you made within the Composition Editor, select **Undo** from the **Edit** pull-down menu. You can undo any or all of the changes you made since you opened the Composition Editor. **Undo** reverses one change at a time, moving backward from the most recently made change.

If you undo an operation that you decide you had right in the first place, select **Redo** from the **Edit** pull-down menu. **Redo** restores the part to the state before the last **Undo**, including any connections that were deleted.

Undo and **Redo** affect operations you perform on the free-form surface and the parts palette in the Composition Editor, the Class Editor, and the Part Interface Editor. They do not affect any of the functions in the **File** pull-down menu.

Editing Parts within a Composite Part

VisualAge provides a way for you to edit a composite part that is embedded within another composite part.

The only exception is that the Composition Editor does not allow you to modify the VisualAge provided base parts. However, if you place one of these base parts on either a Composers part or the free-form surface, you can modify the embedded parts by doing either of the following:

- If you want to add an action to the embedded part, connect to a member function or custom logic that belongs to the composite part. Write a member function or provide custom logic if you need to perform an action of limited use—that is, one that you do not anticipate using very often and that you do not want a derived part to inherit.
- If you want to add a new feature that you plan to use often, create a new part that is derived from the base part. For example, to add a new feature to an IEntryField* part, create a new visual part whose base part is the IEntryField* part and replace the IEntryField* part that you were using with your new part. You can then add as many new features to your new part as you need.

If you want to edit a part that was added to the part you are editing:

1. Move the mouse pointer over the part you want to edit.
2. Click mouse button 2. The pop-up menu appears.
3. Select **Edit part**, and VisualAge displays the appropriate editor for the part:
   - For a visual part, the Composition Editor
   - For a nonvisual or class interface part, the Part Interface Editor.
4. Edit the part. If you want to promote any of the part features used to create the composite part you are editing, do so now to prevent having to edit this part later. See “About Promoting Part Features” on page 103 for more information.
5. Select **File** and then **Save** to save the part.
6. Close the editor and you are returned to the Composition Editor.
7. To apply your changes, select **File** and then **Save** and save the original part.
8. Close the Composition Editor for the original part that you were editing, as described previously.
9. Reopen the original part you were editing to see the changes you made.

RELATED TASKS

“Editing Part Labels Directly” on page 29

“Editing Part Properties” on page 27

“Undoing and Redoing Changes in the Composition Editor” on page 29

“Renaming Parts in the Composition Editor”

“Connecting Features to Member Functions” on page 46

“Connecting Features to Custom Logic” on page 47

Renaming Parts in the Composition Editor

The Composition Editor names parts based on the part palette names 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 displays the message “PushButton1 selected.” The second push button you use is named PushButton2, the third is named PushButton3, and so forth. The Composition Editor assigns these default names to distinguish parts and connections when you generate the code to build your application.

To assign part names that are more descriptive or meaningful to your program element, follow these steps:

1. Move the mouse pointer over the part whose name you want to change.
2. Click mouse button 2 and the pop-up menu for the part appears.
3. Select **Change Name** and a Name Change Request window appears.
4. Type a new name in the entry field.
5. Select **OK**. The Composition Editor changes the name of the part to the name that you typed in the entry field.

You can also change a part name by opening the property sheet for the partand changing the name in the **partName** value column.

RELATED TASKS

"Chapter 9. Working with Parts in the Composition Editor" on page 25

Listing Parts within a Composite Part

The Parts List window provides a way to display an ordered list of the parts dropped on a given part. At first, parts are listed in the order in which they were dropped. You can change the tabbing order of parts that have tabbing set, by rearranging the parts in the list.

Note: If you are using Windows and you have problems with the parts list, set your system palette to 256 colors or fewer.

To view the parts list for a composite part, do the following:

1. Open the composite part in the Composition Editor.
2. Click on the free-form surface with mouse button 2. The pop-up menu appears.
3. Select **View parts list** and the Parts List window appears. At first, the Parts List window displays only the immediate embedded parts of the selected composite part. An expansion icon appears next to each part that contains parts of its own. To see those parts, select the ⊞ expansion icon.

Setting Template Arguments

Collection-class parts rely on class templates to be able to handle a variety of element types. Some IBM Open Class parts contain a collection attribute (actually called *items*), so they, too, use class templates. A list of these parts follows:

- IVBCollectionViewComboBox
- IVBCollectionViewListBox
- IVBContainerControl
- IVSequence

Among these parts, two forms of class template exist. One takes only one argument (element type); the other takes two arguments (element type and collection type).

You are not limited to these template-based classes.

In any case, you must set the appropriate arguments in order for VisualAge to generate code that can be successfully compiled. To do this, select **Change**

Template Arguments from the pop-up menu for the part. Then enter the types required, using template notation for the collection type as necessary.

RELATED TASKS

“Chapter 9. Working with Parts in the Composition Editor” on page 25

Saving a Part

Saving a part that you have constructed includes generating the source code. If the part is an application or if you want to test the part by itself, you need a main part, which is a subclass of IVBMain.

To save the part and generate the source code:

1. Select **File** from the menu bar.
2. Select **Save and generate**.
3. Select **Part source**.

To create an executable application, and generate code for the **main()** function:

1. Create a nonvisual composite part that inherits from IVBMain.
2. Generate source code for the main part as you did for other composites.

RELATED CONCEPTS

“Generated Application Code” on page 112

RELATED TASKS

“Generating Source Code” on page 60

“Generating Source Code for Your Application’s main( ) Function” on page 61

Chapter 10. Dropping Parts in the Composition Editor

Use the Composition Editor to place visual, class interface, and nonvisual parts on the free-form surface. You can add parts that appear on the parts palette, as well as parts that do not appear on the parts palette.

RELATED CONCEPTS

"Chapter 5. Visual Composition" on page 9

RELATED TASKS

"Dropping a Part that Appears on the Palette"

"Dropping a Part that Is Not on the Palette" on page 34

"Unloading the Mouse Pointer" on page 34

"Selecting Parts" on page 35

"Deselecting Parts" on page 35

"Deleting Parts" on page 36

"Chapter 6. Starting the Visual Builder in AIX" on page 15

Dropping a Part that Appears on the Palette

To add a part from the palette to the Composition Editor surface, follow these steps:

1. In the left column of the parts palette, use mouse button 1 to select the category containing the part you want. The right column contains the icons representing the parts in the selected category.
2. From the right column, select the part you want. The mouse pointer becomes a crosshair, indicating that the mouse pointer is loaded with the part you selected.
3. Move the mouse pointer to the location where you want to place the part.
4. Press and hold mouse button 1. An outline of the selected part appears under the mouse pointer. Without releasing the mouse button, move the mouse pointer to position it precisely.
5. Release the mouse button. The part you selected is placed at the location of the mouse pointer, and the mouse pointer returns to normal.

To add multiple instances of the same part, select the **Sticky** check box at the bottom of the parts palette.

If the part you want to add is not on the parts palette, you can add it with the **Add new part** choice from the **Options** pull-down menu.

RELATED CONCEPTS

"About Adding Parts in the Composition Editor" on page 11

"Parts Palette" on page 10

"Free-form Surface" on page 9

RELATED TASKS

"Dropping a Part that Is Not on the Palette" on page 34

"Chapter 10. Dropping Parts in the Composition Editor"

Dropping a Part that Is Not on the Palette

To place a part on the Composition Editor:

1. From the **Options** pull-down menu, select **Add part** and the Add Part window appears.
2. Enter the class name in the **Part class** field. This is the class name that was specified when the part was created. When you begin typing, you replace the highlighted prompt, `class_name*`, with the name of the part you want to add. You can also select the **List...** button and select a class from the **Add Part — Select Class** window.

   The asterisk at the end of the name is a reminder that you are actually entering a pointer to the part. When you enter a valid class name for the part without deleting the asterisk, the Composition Editor automatically selects the **Part** radio button. You can place a pointer to a variable by selecting the **Variable** radio button.

   If you delete the asterisk, the Composition Editor grays out the **Part** radio button and selects the **Variable** radio button. You will be adding a variable with the class name type you entered in the **Part class** field.

   You cannot add abstract parts or template parts using the Add Part window. For example, IButton* is the abstract part that IPushButton* inherits from. You can add IPushButton*, but not IButton*.

   Note: You cannot add an object factory part using the Add Part window because it is a template. You must add it by selecting it on the parts palette.
3. Type a name for the part in the **Name** field. This name appears in the information area at the bottom of the Composition Editor when you select the placed part; it represents the part in generated part code.

   The **Name** field is optional. If you leave it blank, the Composition Editor uses the class name for the part.
4. Select the **Add** push button to add the part. TheAdd Part window disappears and the mouse pointer turns into the same crosshairs used for placing a part on the free-form surface.

   To enable the **Add** push button, you must enter the name of a part that is known to the Composition Editor in the **Part class** field.
5. To keep the Add Part window active after loading the mouse pointer with the part, select **Apply**.
6. Move the crosshairs to the place where you want to add the part and click mouse button 1.

   RELATED CONCEPTS

   “About Adding Parts in the Composition Editor” on page 11

   RELATED TASKS

   “Chapter 10. Dropping Parts in the Composition Editor” on page 33

   “Dropping a Part that Appears on the Palette” on page 33

Unloading the Mouse Pointer

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

- From the tool bar, select

Selection tool.

- From the menu bar, select **Tool**, then **Selection tool**.

Selecting Parts

You must select a part before you can perform an action on it, such as renaming. The Composition Editor displays the name of the selected part in the information area at the bottom of the window. If more than one part is selected, `*Multiple selection*` is displayed.

You cannot select parts and connections together—they are mutually exclusive. However, if you delete a part that is connected to other parts, the Composition Editor deletes the connections and the part.

When you select a part in the Composition Editor, selection handles ■ appear on the corners. If you select more than one part, the last part selected has solid selection handles, indicating that it is the *anchor* part. The other selected parts have hollow selection handles.

To select a single part, click the part with mouse button 1. If you previously selected other parts, they are deselected automatically.

To select multiple parts, do one of the following:

- Hold down the Ctrl key and click mouse button 1 on each additional part you want to select.
- Hold down mouse button 1 and move the mouse pointer over each part you want to select. After you select the parts, release mouse button 1. (This method does not work in Windows.)

Select your anchor part last, or Ctrl and double-click an already selected part to make it the anchor. For example, if you want to match the widths of two parts, the anchor part is used as the guideline width.

RELATED TASKS

"Deselecting Parts"

Deselecting Parts

To deselect a part after you have selected it, hold down the Ctrl key and click the selected part with mouse button 1.

To deselect multiple parts, follow these steps:

1. Hold down the Ctrl key.
2. Click and release mouse button 1 on all the parts you want to deselect.

You can deselect all selected parts by clicking mouse button 1 in a clear spot on the free-form surface.

RELATED TASKS

"Selecting Parts"

Deleting Parts

To delete one or more parts :

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** and the part or parts are deleted.

You can also delete parts by selecting them and pressing the Delete key.

When you delete a connected part, the Composition Editor displays a message to alert you that the connections between that part and other parts are also deleted. However, when you select **Edit** and then **Undo**, you restore any connections that were removed when you deleted the part.

RELATED TASKS

"Editing Part Properties" on page 27

Chapter 11. Manipulating Parts in the Composition Editor

Once you have placed parts on the free-form surface, you can make the following changes to achieve the look and function you want:

- “Positioning Parts”
- “Moving a Part” on page 39
- “Resizing a Visual Part” on page 39
- “Copying a Part” on page 40
- “Changing the Tabbing Order” on page 42
- “Changing Depth Order within a Composite Part” on page 42
- “Tearing Off Attributes” on page 43
- “Using the Grid for Layout”
- “Aligning Parts” on page 38
- “Spacing Composite Parts within Composers Parts” on page 38
- “Spacing Parts within a Bounding Box” on page 39
- “Listing Parts within a Composite Part” on page 31
- “Editing Parts within a Composite Part” on page 29
- “Deleting Parts” on page 36

Positioning Parts

Positioning a part refers to aligning or spacing. The tool bar and the **Tools** pull-down menu provide options for aligning selected parts.

The pop-up menu provides options for spacing within the bounding box, an unseen box that contains the selected parts. You can also manage the placement of parts by using a multi-cell canvas.

RELATED TASKS

“Chapter 11. Manipulating Parts in the Composition Editor”

Using the Grid for Layout

The Composition Editor provides grids for positioning parts. The free-form surface has a grid and so do container parts such as frame windows. The Composition Editor provides the following grid options:

Snap to Grid
This tool positions the upper left corner of the part(s) to the nearest grid coordinate. **Snap to grid** works on the part(s) you have selected that are already on the surface. The **Snap to grid** tool is on the tool bar and also listed in the **Tools** pull-down menu.

Snap on Drop
When **Snap on drop** is activated, every part you drop on the gridded surface aligns to the nearest grid coordinate. The **Snap on drop** tool is located on the **Options** pull-down menu.

Snap on Size

This tool allows you to align the edges of a part to the grid as you size it. Sizing to the grid helps keep part dimensions in exact and easily countable increments.

Set Grid Spacing

This tool allows you to determine the horizontal and vertical distance, in pixels, between the lines of the grid. Select this tool from the pop-up menu.

Toggle Grid

This tool allows you to show or hide the grid. 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, the **Toggle Grid** tool toggles the grid for the Composers part.

RELATED TASKS

“Selecting Parts” on page 35

“Positioning Parts” on page 37

Aligning Parts

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

   Align Left

   Align Top

   Align Center

   Align Middle

   Align Right

   Align Bottom

RELATED TASKS

“Selecting Parts” on page 35

“Spacing Composite Parts within Composers Parts”

“Spacing Parts within a Bounding Box” on page 39

Spacing Composite Parts within Composers Parts

To evenly space composites within their Composers part:

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

RELATED TASKS

"Selecting Parts" on page 35

"Spacing Parts within a Bounding Box"

"Aligning Parts" on page 38

Spacing Parts within a Bounding Box

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** and then **Distribute**, and then either **Horizontally in bounding box** or **Vertically in bounding box**.

RELATED TASKS

"Aligning Parts" on page 38

"Spacing Composite Parts within Composers Parts" on page 38

Moving a Part

To move a part in the Composition Editor, follow these steps:

1. Place the mouse pointer over the part you want to move.
2. Hold down the appropriate mouse button and move the mouse pointer to the new location.
   - In Windows, hold down mouse button 1.
   - In OS/2, hold down mouse button 2.
   - In AIX, hold down the middle mouse button.

   An outline of the part you are moving appears as you move the mouse.
3. Release the mouse button. The part appears in its new location with a solid border around the window or form that contains the part, indicating target emphasis.

If the part you are dragging is one of several that you selected, all selected parts move together. Pressing the Esc key cancels the move without making changes.

RELATED TASKS

"Chapter 11. Manipulating Parts in the Composition Editor" on page 37

Resizing a Visual Part

You can change the size of a visual part in the Composition Editor using any of the following techniques:

- Dragging the selection handles
- Matching by multiple selection
- Changing Size and position prompter properties
- Resize option on the part pop-up resets to the default size

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

RELATED TASKS

"Resizing a Part by Dragging"

"Matching Part Sizes Using the Tool Bar"

Resizing a Part by Dragging

To change the size of a visual part, follow these steps:

1. Select the part by clicking it with mouse button 1. To size several parts at once, select all the parts you want to size.
2. Place the mouse pointer over one of the handles and hold down mouse button 1.
3. While holding down mouse button 1, drag the handle to a new location. As you move the mouse, the outline of the part dynamically changes size. When it is the size you want, release the mouse button. The part changes to the size that you chose.

Note: Pressing the Esc key cancels resizing without making changes.

To size a part in only one direction, press and hold the Shift key while sizing the part. Holding down the Shift key prevents one dimension of the part from changing when 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.

RELATED TASKS

"Matching Part Sizes Using the Tool Bar"

Matching Part Sizes Using the Tool Bar

1. Select all the parts you want to size, making sure the last part you select, the anchor, is the size you want the others to match. You can change the anchor by holding the Ctrl key and double-clicking the new anchor.
2. Select one of the following from the tool bar or the **Tools** pull-down menu:

   Match Width

   Match Height

   The size of the selected parts changes to match the size of the anchor part.

Copying a Part

You can use the following techniques to copy parts in the Composition Editor:

- "Copying Parts by Dragging" on page 41
- "Copying Parts Using the Clipboard" on page 41

Copying Parts by Dragging

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. Position the mouse pointer over one of the parts you want to copy.
3. Hold down and do not release both the Ctrl key and the appropriate mouse button.
   - For Windows, hold mouse button 1.
   - For OS/2, hold mouse button 2.
   - For AIX, hold mouse button 3.
4. Move the mouse pointer to a new position. To help you with positioning, an outline of the part or parts appears. When you are copying multiple parts, the outlines of each part move together as a group.
5. When the copy is in the desired position, release the mouse button and Ctrl key. The copy appears where you positioned the outline or outlines.

Note: Pressing the Esc key before releasing the mouse button cancels copying without making changes.

RELATED TASKS

"Copying Parts Using the Clipboard"

Copying Parts Using the Clipboard

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. From the **Edit** pull-down menu, select **Paste**. The mouse pointer turns into crosshairs, indicating that it is loaded with the copied parts.
4. Position the mouse pointer where you want the parts to be copied.
5. Click mouse button 1. Copies of the parts are pasted at the position of the mouse pointer.

Note: Only the part or parts are copied to the clipboard, not the connections.

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 ↖ Selection tool on the tool bar or by selecting **Tools** on the menu bar and then **Selection tool** from the pull-down menu.

RELATED TASKS

"Copying Parts by Dragging"

Changing Depth Order within a Composite Part

Depth order is the order in which parts are stacked on the application desktop. Parts lower (or later) in the depth order overlay at least a portion of parts higher up (or first). An example of this is a push button on a canvas. The canvas appears higher in the depth order; the push button, which lies on top of the canvas, appears lower.

The Composition Editor assigns the depth order as parts are dropped. Depth order is hierarchical, depending on the arrangement of Composers parts.

You can change the depth order within a single composite part by dragging items in the parts list. To change the depth order, do the following:

1. Open a parts list in the composite part by selecting **View parts list** from the pop-up menu of the part.
2. To move more than one part:
   - If the parts are adjacent in the list, select the first part in the group you want to move. Hold down the Shift key and select the last part in the group you want to move. VisualAge highlights all the parts between the first and the last part selected.
   - If the parts are not adjacent in the list, select the first part and hold down the Ctrl key while selecting the other parts.

   Note: 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 you set the tabbing for any of the moved parts, the tabbing order is also changed.

RELATED CONCEPTS

"About Setting Groups and Tab Stops" on page 12

RELATED TASKS

"Changing the Tabbing Order"

"Setting Tab Stops and Groups" on page 43

Changing the Tabbing Order

1. Open a parts list for the Composers part.
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.

Note: You cannot move a composite to a new Composers part by changing the tabbing order. You must do this by moving the parts themselves in the Composition Editor.

RELATED CONCEPTS

"About Setting Groups and Tab Stops" on page 12

RELATED TASKS

"Changing Depth Order within a Composite Part" on page 42

"Setting Tab Stops and Groups"

"Opening the Property Sheet for a Part" on page 27

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. Open the pop-up menu for the part.
2. Select **Set tabbing**.
3. Select **Tab stop**, **Group** or both.

A check next to the option, indicates that it is set.

You can also **Set tabbing** through the property sheet, as follows:

1. Select the part.
2. Open the pop-up menu for the part.
3. Select **Open Settings** and the Properties window appears.
4. Select the **Style** value field and a small box ... appears on the right side of the field.
5. Select the box and the **Style** options appear.
6. Deselect the **Set value to nil (None)** checkbox and the Property column is enabled.
7. Select the **Group**, **Tab stop**, or value fields and select **ON** from the drop-down list.
8. Select **OK** twice.

The group includes all parts in the tabbing order below the part that has **Group** selected.

To start another group, enable **Group** for the part with the initial input focus for the group.

Note: If a part has both **Group** and **Tab stop** enabled, 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.

RELATED CONCEPTS

"About Setting Groups and Tab Stops" on page 12

RELATED TASKS

"Changing Depth Order within a Composite Part" on page 42

"Changing the Tabbing Order" on page 42

Tearing Off Attributes

1. Select **Tear off attribute** from the pop-up menu of the part with the attribute you want to access. Another menu appears listing all of the attributes of the part.
2. Select the attribute you want to tear off. The mouse pointer is now loaded with a variable part representing or pointing to the attribute you selected.

3. Place the new part on the free-form surface, as you would any other nonvisual part. The torn-off attribute now appears as a variable part connected to the original part by an attribute-to-attribute connection.

RELATED CONCEPTS

“About Tearing Off Attributes” on page 13

“Attribute-to-Attribute Connections” on page 106

Chapter 12. Making Connections in the Composition Editor

In VisualAge, you draw connections between parts to define their interaction. This involves using the mouse to select part features from each end point in the Composition Editor. The type of feature at either end—attribute, action, event, or member function—determines the type of connection. You may find the **Browse part features** option on the part pop-up menu useful in determining the type of connection you wish to make.

If you decide to change the behavior of the part later, you can edit or reorder the existing connections without redrawing them.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Connecting Features to Other Features”

“Supplying Parameter Values for Incomplete Connections” on page 49

“Opening Connection Settings” on page 51

“Connecting Features to Member Functions” on page 46

“Connecting Features to Custom Logic” on page 47

“Reordering Connections” on page 55

“Selecting and Deselecting Connections” on page 54

“Deleting Connections” on page 53

“Showing and Hiding Connections” on page 53

“Changing the Source and Target of Connections” on page 56

“Connecting from Connection Results” on page 48

Connecting Features to Other Features

To connect two features, follow these steps. In this procedure, the term *source* refers to the first end point. The term *target* refers to the second end point.

1. Select the source part, click mouse button 2, and select **Connect** from the pop-up menu. In OS/2 and Windows NT, you can display the connection pop-up menu more quickly as follows:
   a. Position the mouse pointer over the source.
   b. Hold down the Alt key while clicking mouse button 2.

   In most cases, a cascade menu appears that displays the names of the most commonly used (or *preferred*) features. If additional features exist that are appropriate for the connection type, **All Features** also appears on the menu. Selecting **All Features** opens a connection window with an expanded list of features, sorted alphabetically and by feature type.

   - If a connection window appears instead of the cascade menu, this means that preferred features have not been assigned for the part.
   - If the **All Features** selection does not appear on the menu, this means the menu contains all available features, not just the preferred ones, and there are no more from which to select.

2. Select a feature by doing one of the following:
   - If the feature appears in the preferred list, select it.
   - If the feature does not appear in the list but the **All Features** selection is available, select **All Features** and then select the feature from the expanded list in the connection window.
   - If the feature does not appear in either the preferred or expanded list, you may be able to edit the part to add the feature you need.
3. If, at this point, you decide not to complete the connection, do one of the following:
   - If a pop-up menu appears, move the mouse pointer away from the connection menu and click mouse button 1.
   - If a window showing all the features appears, select the **Cancel** push button at the bottom of the window.

   The menu or window disappears without completing the connection.
4. Place the mouse pointer over the target part. As you move the mouse, a dashed line trails from the mouse pointer back to the source part.
5. Click mouse button 1. As with the source part, either a pop-up menu or connection window appears.
6. Select the target feature as before.

When you complete the connection, a colored connection line appears. The color indicates the connection’s type, based on the features you selected as end points.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Connecting Features to Custom Logic” on page 47

“Supplying Parameter Values for Incomplete Connections” on page 49

“Opening Connection Settings” on page 51

“Reordering Connections” on page 55

“Selecting and Deselecting Connections” on page 54

“Changing the Source and Target of Connections” on page 56

Connecting Features to Member Functions

The source for a member function connection must be either an event or an attribute with an associated event. Connect to a member function as follows:

1. Open the connection menu for the source part.
2. From this menu, select the event. If the event you want is not on the **Preferred Features** list, select the feature from the **All Features** menu.
3. Move the mouse pointer to an empty point on the free-form surface and click mouse button 1. Doing this indicates that you want to connect the event to the composite part rather than to one of its embedded parts.

   The Composition Editor displays the Member Function Connection window.
4. Type the signature of the existing member function in the entry field. Be sure to type the full declaration of the member function, including return type.

5. If the member function takes input parameters and you want to specify them as constants, select the **Set parameters** push button.
6. Select **OK**.

   The connection window disappears and VisualAge draws a green connection arrow from the source part toward the edge of the free-form surface. If the connection arrow is dashed, you must supply values for the input parameters of the member function.
7. Before you generate the code for your part, you must switch to the Class Editor and enter the names of the files containing the member function code so that VisualAge includes them in your part code.
8. You must create an .hpv file that contains a declaration for this member function and a .cpv file that contains the code for this member function. The member function declaration that you enter in this field must be **identical** to the declaration in the .hpv file.

RELATED CONCEPTS

"Member Function Connections" on page 108

RELATED TASKS

"Supplying a Parameter Value Using a Constant" on page 50

"Connecting Features to Custom Logic"

"Connecting from Connection Results" on page 48

"Opening Connection Settings" on page 51

Connecting Features to Custom Logic

The source for a custom logic connection must either be an event or an attribute with an associated event.

1. Open the connection menu for the source part.
2. From this menu, select the event.
3. Move the mouse pointer to the target part and click mouse button 1. The connection menu opens for the target part.
4. Select **Custom logic**. The Custom Logic Connection window appears.
5. Enter information about the connection in the **Description** field.

   The **Description** field serves two purposes. After you complete the connection, if you select the connection line, the text that you entered in this field appears in the information area at the bottom of the Composition Editor.

   When you generate the code for this part, VisualAgeinserts this description in the ITRACE_DEVELOP statement that appears in the dispatchNotificationEvent member function for the connection class. When you run the compiled application with tracing set on, VisualAge write this description to the trace destination whenever the connection code runs.
6. Enter the appropriate value in the **Return type** field.

   This field contains the return type for your custom logic. If you want to pass a return value from your custom logic, change the return type from the default `void` to the type of data that you want to pass.
7. Add code to the **Custom Logic** field.

   The **Custom Logic** field is a multiline entry field, where you enter your custom logic.

You can specify whether your custom code affects the source or the target part by selecting the appropriate push button before entering a line of code.

Select the **Event** push button to insert a reference to the source of the connection (an INotificationEvent). The event reference gives you access to the data that is passed by the event. You might use this to test for specific event data and do processing based on the result of the test.

8. Select the **Add** push button and the connection window disappears. VisualAge draws a connection arrow from the source to the target.

RELATED CONCEPTS

“Custom Logic Connections” on page 109

RELATED TASKS

“Connecting Features to Other Features” on page 45

“Connecting Features to Member Functions” on page 46

“Opening Connection Settings” on page 51

“Reordering Connections” on page 55

“Selecting and Deselecting Connections” on page 54

“Changing the Source and Target of Connections” on page 56

Connecting from Connection Results

An *exception* is any user, logic, or system error detected by a function that does not deal with the error itself but passes the error on to a handling routine, called an *exception handler*. In VisualAge, you can catch exceptions by connecting exception events to either actions or member functions.

An exception event is a feature of a connection, not a part. It appears as *exceptionOccurred* on the connection’s connection menu. Exception events are typically connected to IMessageBox* parts, which are used to display error messages associated with the exceptions.

The return type of a target action displays as the *actionResult* event of the connection. You can connect the *actionResult* event to a feature of the same part or another part.

To connect an exception event, do the following:

1. Write a member function or action that throws an exception. The easiest way to do this is to include something like the following:

   ```
   throw IException("Error message.");
   ```

   where the text of the error message that you want to display in the message box is the only parameter given for the IException constructor.
2. For an action, update the part interface.
3. Connect an event to the action or member function you just created.
4. Drop an IMessageBox* part on the free-form surface. This message box will display the error message in the completed application.
5. Connect the *exceptionOccurred* event of the connection to the *showException* action of the IMessageBox* part.

This connection causes the application to show a message box that contains the exception error message whenever the exception is thrown.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Connecting Features to Other Features” on page 45

“Connecting Features to Member Functions” on page 46

“Connecting Features to Custom Logic” on page 47

“Opening Connection Settings” on page 51

“Selecting and Deselecting Connections” on page 54

Supplying Parameter Values for Incomplete Connections

Connections sometimes require parameters, or input arguments. If a connection requires parameters that have not been specified explicitly or by default, it appears as a dashed arrow, indicating that it is incomplete. When you have made all the necessary parameter connections, the connection line becomes solid, indicating that the connection is complete.

Be sure that you make the parameter connections before code is generated for your part.

RELATED CONCEPTS

“Parameter Connections” on page 110

RELATED TASKS

“Supplying a Parameter Value Using a Connection”

“Supplying a Parameter Value Using a Constant” on page 50

“Specifying Default Values for Parameters” on page 50

“Chapter 12. Making Connections in the Composition Editor” on page 45

“Opening Connection Settings” on page 51

“Reordering Connections” on page 55

“Changing Connection Settings When a Member Function Is the Target” on page 52

Supplying a Parameter Value Using a Connection

One way to supply parameter values is to make a connection from the incomplete connection line to the parts supplying the parameter values. Most of the time, the values you need are those of attributes from other parts that you are working with in the Composition Editor. Sometimes, however, the value you need is the return value from an action, a member function, or custom logic.

1. Start a new connection using as the source, the dashed connection line that requires the parameter.

   When you make a connection, the Composition Editor 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.

2. For the target, select the feature that is to provide the value.

   RELATED CONCEPTS

   "Parameter Connections" on page 110

   RELATED TASKS

   "Supplying a Parameter Value Using a Constant"

   "Specifying Default Values for Parameters"

   "Chapter 12. Making Connections in the Composition Editor" on page 45

   "Connecting from Connection Results" on page 48

Supplying a Parameter Value Using a Constant

When connections need parameters with constant input values, provide these values through the settings window of the incomplete connection, as follows:

1. Open settings for the incomplete connection by selecting **Open Settings** from the pop-up menu. The settings window of the incomplete connection appears.
2. Select **Set parameters**. The Constant Parameter Value Settings window appears showing the parameters for which you can set constant values.
3. Enter the constant parameter values you want to use.

   Type the parameter values just as you would type them as parameters for the member function if you were coding it by hand. For example, to provide a constant value for a *text* parameter, enter the string that you want the parameter to receive. VisualAge copies these values as strings to the output files when you generate code.
4. Do one of the following:
   - To apply the values and save them, select **OK**. This action closes the window.
   - To close settings without saving any of the parameter values you just entered, select **Cancel**.

   RELATED CONCEPTS

   "Parameter Connections" on page 110

   RELATED TASKS

   "Opening Connection Settings" on page 51

   "Supplying a Parameter Value Using a Connection" on page 49

   "Specifying Default Values for Parameters"

Specifying Default Values for Parameters

You can specify default parameters for actions in the declaration of the member function for the action, as follows:

- In the .vbe file for the part
- On the Action page of the Part Interface Editor

Either of these enables a default value to be passed in an event-to-action connection, thus avoiding the need to supply a parameter value.

For example, suppose you want to connect the *buttonClickEvent* feature of a **Remove** push button to a *removeSelected* action that you created for an

IListBox*-based part. Normally, you would also need to connect the *selection* attribute of the IListBox*-based part to the *index* attribute of the connection between *buttonClickEvent* and *removeSelected*. This connection would be required to get the index of the selected item in the list box.

However, in the .vbe file of the IListBox*-based part, you can specify the following default parameter value for the *removeSelected* action:

```
//VBAction: removeSelected, "removeSelected",,
//VB:             removeSelected(unsigned long index=selection())
```

This means that if no attribute of the IListBox*-based part is connected to the *index* attribute, the selection member function (the get member function of the *selection* attribute) is called by default to provide the index of the selected item.

You could do the same thing by creating a *removeSelected* action on the **Action** page of the Part Interface Editor for the IListBox*-based part. You would specify the default parameter in the declaration of the removeSelected member function in the **Action member function** field as follows:

```
virtual unsigned long removeSelected(unsigned long index=selection())
```

However, you must include this signature and the default parameters in the actual C++ code or you will receive an error.

RELATED CONCEPTS

“Parameter Connections” on page 110

RELATED TASKS

“Supplying a Parameter Value Using a Connection” on page 49

“Supplying a Parameter Value Using a Constant” on page 50

“Chapter 12. Making Connections in the Composition Editor” on page 45

Opening Connection Settings

Connection settings enable you to change a connection without redrawing it. Through the settings window, you can do the following:

- Change the source or target feature, depending on the connection type
- Reverse the direction of an attribute-to-attribute connection
- Specify an input parameter as a constant
- Delete the connection

To open settings 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.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Chapter 12. Making Connections in the Composition Editor” on page 45

“Supplying a Parameter Value Using a Constant” on page 50

"Changing the Source and Target of Connections" on page 56

"Deleting Connections" on page 53

"Changing Connection Settings When a Member Function Is the Target"

Changing Connection Settings When a Member Function Is the Target

The **Settings** window for member function connections looks similar to the window you used to specify the original connection:

- To change the source event, either select a new event fro the drop-down box or type the name of an event in the **Event name** field. If a field labeled **Attribute name** appears instead, the current source event is associated with an attribute. To change it, type the name of an attribute in the field.
- To change the member function, enter a new member function signature in the entry field at the bottom of window.
- To supply a constant parameter value, select the **Set parameters** push button.

RELATED CONCEPTS

"Member Function Connections" on page 108

Chapter 13. Manipulating Connections on the Free-Form Surface

Once you have made connections to and from parts you dropped on the free-form surface, you can modify them as follows:

- Display or hide the connection lines
- Delete the connection
- Reorder the connections
- Change the source and target of the connection without starting over

RELATED TASKS

"Showing and Hiding Connections"

"Deleting Connections"

"Selecting and Deselecting Connections" on page 54

"Reordering Connections" on page 55

"Changing the Source and Target of Connections" on page 56

Showing and Hiding Connections

You can show and hide connections by using the **Show connections** and **Hide connections** tools. They can be found on the tool bar or as selections on the **Tools** pull-down menu. These tools show or hide all connections that have the selected part or parts as their end points. If no parts are selected, these tools show and hide all connections in the composite part.

If you hide connections, the Composition Editor is refreshed faster and is less cluttered, making it easier for you to work.

You can also do this from the pop-up menu by selecting the **Browse Connections** cascade menu. The choices in this menu affect only connections going to and from the part whose pop-up menu you opened.

RELATED CONCEPTS

"Chapter 26. Connections" on page 105

RELATED TASKS

"Selecting Parts" on page 35

"Chapter 12. Making Connections in the Composition Editor" on page 45

Deleting Connections

To delete a connection, do one of the following:

- Select the connection and press the **Delete** key.
- From the connection's pop-up menu, select **Delete**.
- From the connection's **Settings** window, select the **Delete** push button.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Opening Connection Settings” on page 51

“Displaying Part Pop-Up Menus” on page 26

“Chapter 12. Making Connections in the Composition Editor” on page 45

“Selecting and Deselecting Connections”

Selecting and Deselecting Connections

You select connections in the same way that you select parts. When you select a connection, boxes called *selection handles* ■ appear on it to show that it is selected. When first drawn, a connection contains three selection handles: one at each end and one in the middle. You can use selection handles to change either of the following:

- The end points of the connection.
- The shape of the connection line, by dragging the middle box to another location. This helps you distinguish among several connection lines that are close together.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Changing the Shape of a Connection” on page 57

“Selecting a Single Connection”

“Selecting Multiple Connections”

“Deselecting Connections” on page 55

Selecting a Single Connection

1. Move the mouse pointer over the connection you want to select.
2. Click mouse button 1 and the connection is selected.

RELATED TASKS

“Selecting Multiple Connections”

“Deselecting Connections” on page 55

Selecting Multiple Connections

To select multiple connections in OS/2 using just the mouse, follow these steps:

1. Move the mouse pointer over one of the connections you want to select.
2. Hold down mouse button 1 instead of clicking it.
3. Move the mouse pointer over each connection that you want to select. The selection boxes appear on each connection that the mouse pointer passes over to show they are selected.
4. After the connections are selected, release mouse button 1.

To select multiple connections on all platforms using both the mouse and the keyboard, follow these steps:

1. Hold down the Ctrl key.
2. Move the mouse pointer over a connection.
3. Click mouse button 1 while the mouse pointer is over the connection line.
4. Without releasing the Ctrl key, repeat the preceding steps until all connections that you want to select are selected.

   RELATED TASKS

   “Selecting a Single Connection” on page 54

   “Deselecting Connections”

Deselecting Connections

To deselect a connection without selecting another part or connection, follow these steps:

1. Move the mouse pointer over the connection line.
2. Hold down the Ctrl key.
3. Click mouse button 1.

   RELATED TASKS

   “Selecting a Single Connection” on page 54

   “Selecting Multiple Connections” on page 54

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 might need to reorder the connections. If so, do the following:

1. Select the source part.
2. From the source part pop-up menu, select **Reorder Connections From**. The Reorder Connections window appears, showing a list of your connections.
3. With the mouse pointer over the connection you want to reorder, press and hold as follows:
   - In OS/2, mouse button 2
   - In Windows, mouse button 1
   - In AIX, mouse button 3
4. Drag the connection to the place in the list where you want the connection to occur.

   Note: Parameter connections must always follow the connections they supply.
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.

   RELATED CONCEPTS

   “Chapter 26. Connections” on page 105

   RELATED TASKS

“Chapter 12. Making Connections in the Composition Editor” on page 45

“Opening Connection Settings” on page 51

“Selecting and Deselecting Connections” on page 54

Changing the Source and Target of Connections

You can change the end points of a connection without redrawing it, either by dragging the connection or by changing its settings.

You can change the source of any connection. In most cases, you can also change the target. However, depending on the feature that you connect to when you make the change, you might get a different type of connection than the one you started with.

Table 2. What Can Be Changed on a Connection, by Type

Connection type	**Move either end**	**Move source end only**
attribute-to-attribute	x	
attribute-to-member function		x
attribute-to-action	x	
event-to-action	x	
event-to-attribute	x	
event-to-member function		x
custom logic	x	
parameter connection	x	

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Moving Either End of a Connection to a Different Part”

“Moving Either End of a Connection to a Different Feature” on page 57

“Reversing the Direction of a Connection” on page 57

“Chapter 12. Making Connections in the Composition Editor” on page 45

“Opening Connection Settings” on page 51

“Changing the Shape of a Connection” on page 57

Moving Either End of a Connection to a Different Part

1. Select the connection.
2. Move the mouse pointer over the appropriate selection handle at the end of the connection.
3. Press and hold mouse button 1.
4. Move the mouse pointer to the new part or connection.
5. Release the mouse button.

If you change the target 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 target feature.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Moving Either End of a Connection to a Different Feature”

“Chapter 12. Making Connections in the Composition Editor” on page 45

“Selecting and Deselecting Connections” on page 54

Moving Either End of a Connection to a Different Feature

1. Open settings for the connection. The **Settings** window for that connection type appears.
2. Select new end points from the lists shown.
3. Select the **OK** push button.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Opening Connection Settings” on page 51

“Moving Either End of a Connection to a Different Part” on page 56

“Chapter 12. Making Connections in the Composition Editor” on page 45

Reversing the Direction of a Connection

The direction of attribute-to-attribute connections determines which end point is initialized first. The target attribute is initialized first based on the value of the source. Only attribute-to-attribute connections can be reversed. To do this, open settings for the connection and select the **Reverse** push button.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Opening Connection Settings” on page 51

Changing the Shape of a Connection

To help you distinguish among several connection lines that are close together, you can change the shape of connections. To do this, follow these steps:

- Select the connection line you want to change.
- Place the mouse pointer over the middle box of the connection line.
- Click and hold mouse button 1 and drag the connection line to the desired shape.
- Release the mouse button and the new line is set.

You can change the connection back to its original shape by selecting **Restore Shape** from the pop-up window.

Chapter 14. Generating Code

The parts you have created are stored in the VisualAge data model. This includes information about the part interfaces, connections between them, and initial property settings. At several points during development, you will want to generate code for your application, compile it, and test it. The process for a visually constructed application follows.

1. “Preparing for Code Generation”
2. “Generating Source Code” on page 60, including “Generating Resource Code for Translation” on page 61
3. “Generating Source Code for Your Application’s main( ) Function” on page 61
4. “Generating User Interface Source” on page 61 (for custom visual parts only)
5. “Chapter 15. Preparing for Compilation of Generated Files” on page 63
6. “Chapter 16. Debugging Connections” on page 65
7. “Chapter 17. Fixing Problems with Your Visually Constructed Application” on page 67
8. “Chapter 23. Packaging Visually Composed Applications” on page 95

The process of generating feature code is treated separately under “Assembling a Part from Generated and User-Written Code” on page 84.

RELATED CONCEPTS

“Chapter 27. Generated Code” on page 111

Preparing for Code Generation

Before generating code, you must set generation options in the Class Editor. A summary of tasks associated with these settings follows.

Enabling code generation

To enable generation of class code, select the **Generate code (.hpp and .cpp)** check box.

To enable generation of feature code, select the appropriate check boxes in the **User files included in generation** group. This selection also ensures that feature files are properly included in the generated class files.

Specifying a name for generated files

By default, VisualAge sets a name for generated files. However, you can change the default by modifying the contents of the **C++ header file** field. When this value is changed, the value of the **Base name for other files** field is also changed.

Including handwritten code files

Before VisualAge can generate the proper include statements for handwritten code, you must select the appropriate check boxes in the **User files included in generation** group.

Linking in a part that will be compiled separately

If the part will be compiled separately into a runtime library, you can indicate so to VisualAge by entering the name of the link library file in the **.LIB file name** field. If this is set, VisualAge generates a #pragma library statement in the.hpp file for the part.

Using a resource library

If the part uses a resource library, you must indicate so to VisualAge by entering a value in the **Resource DLL name** field.

Enabling custom visual support

To enable behavioral support for a custom visual part on the free-form surface, enter the name of the part’s interface library in the **Composition Editor interface DLL name** field and select the **Use interface DLL** check box.

Selecting the check box indicates to VisualAge that you want the part instantiated ″live″ on the free-form surface. If you do not select the check box, VisualAge uses a gray box to represent the part on the free-form surface.

RELATED CONCEPTS

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Generating Source Code”

“Generating Resource Code for Translation” on page 61

“Generating User Interface Source” on page 61

Generating Source Code

After you finish constructing a part, you must generate source code for it. You can generate source code for the part being edited as follows:

1. From the editor’s menu bar, select **File**.
2. Select **Save and generate**; then select **Part source**. When generation is complete, VisualAge displays the results in a log window.

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

If you have not already done so, the next step is “Generating Source Code for Your Application’s main( ) Function” on page 61.

RELATED CONCEPTS

“Generated Part Code” on page 112

RELATED TASKS

“Chapter 14. Generating Code” on page 59

RELATED REFERENCES

“Code Generation Errors” on page 116

Generating Resource Code for Translation

VisualAge automatically generates separate code for window resources contained within visual parts. If you intend to have your application’s user interface translated into another language, you can opt to have VisualAge generate separate code for string resources as well.

To have VisualAge generate separate code for string resources, set the following generation options in the Class Editor before generating code:

- Select the **Generate code (.hpp and .cpp)** check box.
- Select the **Generate resource IDs for strings** check box
- Enter a value in the **Starting resource ID** field.
- Specify the name you plan to give the compiled resource library by entering a value in the **Resource DLL name** file.

For guidelines on choosing an appropriate starting resource ID, read the conceptual topic listed below.

RELATED CONCEPTS

“Resource Files Generated for Translation” on page 116

RELATED TASKS

“Chapter 14. Generating Code” on page 59

RELATED REFERENCES

“Code Generation Errors” on page 116

Generating Source Code for Your Application’s main( ) Function

To create an executable application, you must generate code for the standard main( ) function. You can do this for parts that you want to test individually or for your entire application. In either case, the process is as follows:

1. Create a nonvisual composite part that inherits from IVBMain.
2. Generate source code for the main part as you would any other composite.

The next step is “Chapter 15. Preparing for Compilation of Generated Files” on page 63 .

RELATED CONCEPTS

“Generated Application Code” on page 112

RELATED TASKS

“Generating Source Code” on page 60

Generating User Interface Source

To have VisualAge generate interface library source code for custom visual parts, follow these steps:

1. In the Class Editor, select the Generation Options tab.
2. Enter the name of the interface library in the **Composition Editor interface DLL name** field.

3. From the menu bar, select **File**.
4. Select **Save and generate**; then select **User Interface Source**. When generation is complete, VisualAge displays the results in a log window.

The generated code file has an extension of .cp. In OS/2 and Windows, you can treat this file as you could a .cpp file. In AIX, you must do a little work by hand before VisualAge can compile the interface library. For more information, read “Chapter 15. Preparing for Compilation of Generated Files” on page 63.

RELATED CONCEPTS

“Generated Interface Code” on page 113

“About Implementing Custom Visual Parts” on page 101

RELATED TASKS

“Chapter 20. Deriving from Visual Classes” on page 81

“Sharing Custom Visual Parts” on page 89

Chapter 15. Preparing for Compilation of Generated Files

Before compiling your application, be sure the following files have been generated:

- Header files for all parts
- .cpp files for all parts, including the main part
- Resource files for any applicable parts

Make sure the following options are enabled in the application’s build setup:

```
gen(enumsize, 4),
gen(rtti, yes),
defaults(xlC_r),
lang(nokeyword,"false"),
lang(nokeyword,"true"),
lang(nokeyword,"bool"),
define(IC_LANG_BOOL,0),
define("IC_TRACE_DEVELOP"),
link(linkwithsharedlib, yes),
link(linkwithmultithreadlib, yes),
link(map, no),
link(padding, no),
incl(searchpath,".")
```

If you intend to compile parts into separate runtime or interface libraries, remember to set the appropriate targets in the application’s build setup. For complete examples of build setups for executable files, runtime libraries, and interface libraries, see the OASearch sample.

In OS/2 and Windows, part files (.vbf) are automatically recognized as a valid source format in the IDE; it is from these that VisualAge reads dependency information. In AIX, however, you must use the generated .cpp file names; you must also explicitly include any system runtime libraries that must be linked. Here is a portion of the AIX build setup for the OASearch application, found in oasearch.icc. Note the specification of the runtime library for a custom visual part, OAMorphableSpinButton, which is found in the current build directory.

```
        "skill.cpp",
        "skillb.cpp"
 source "pthread.h"
 source "./oamorph.a"

 source "libvacbase.a"
 source "libvacui.a"
 source "libvacvgui40.a"
 source "libvacvngui40.a"
 source "libvacgraph2d.a"
```

If you intend to build an interface library for custom visual parts in AIX, you must write a small amount of code by hand and add the new code file to your build setup. (You need one of these code files for each interface library you plan to build, regardless of the number of parts to be included.) This is not necessary for OS/2 or Windows; in these cases, build the interface library the same way you would build the runtime library for the part. Here is the AIX interface code for the OAMorphableSpinButton part, found in oamorph.cp:

```
#include <iavlbase.h>
#if defined (IC_MOTIF)
//
// Add one extern statement for each custom visual part.
// Get the name of the primitive table from the interface source file
//   that is generated for the part.
```

```
//
extern EsPrimitiveTable (OAMorphableSpinButtonTable);
//
// You can name this new primitive table anything you want,
//   but the name must be unique to your application.
//
EsDefinePrimitiveTable(OAMorph)
//
// Add one EsSubTable statement for each custom visual part.
// Use the same primitive table name as you did in the extern statement.
//
EsSubTable (OAMorphableSpinButtonTable)
EsEndPrimitiveTable
#endif
```

In AIX, the IDE does not recognize .cp as a valid source type, so you must explicitly declare it in the build setup for the interface library. Note that the name of the AIX build target ends in **.w**; Visual Builder assumes this for all interface libraries in AIX. Also note the correspondence between the primitive table created in the previous code and one link option that follows. Here is the AIX build setup for OAMorphableSpinButton, found in oamorphw.icc:

```
option
       gen(enumsize, 4),
       gen(rtti, yes),
       defaults(xlC_r),
       lang(nokeyword,"false"),
       lang(nokeyword,"true"),
       lang(nokeyword,"bool"),
       define(IC_LANG_BOOL,0),
       define("IC_TRACE_DEVELOP"),
       link(linkwithsharedlib, yes),
       link(linkwithmultithreadlib, yes),
       link(map, yes),
       link(padding, no),
       link(sharedLibPriority,-2147482557),
       link(exportAll),
       link(entry,OAMorph),
       incl(searchpath,".")

{


      target type(shr) "oamorphw.w"
      {
         source type(cpp) "oamorphw.cp"
         source type(cpp) "oamrfbtn.cp"
       source "libvacvifbase.a"
         source "libvacbase.a"
       source "libvacui.a"
         source "./oamorph.a"

      }

}
```

RELATED TASKS

“Chapter 14. Generating Code” on page 59

Changing a Build Setup

“Chapter 16. Debugging Connections” on page 65

Chapter 16. Debugging Connections

To make connections easier to debug, follow these steps:

1. Set environmental variables.

   In OS/2 or Windows, add these lines to your config.sys file and reboot:

   ```
   SET ICLUI TRACE=ON
   SET ICLUI TRACETO=OUT
   ```

   In AIX, add these lines to your .profile file and log back in:

   ```
   export ICLUI_TRACE=ON
   export ICLUI_TRACETO=OUT
   ```

2. Make sure the following options are enabled in the application's build setup:
   - Under Linker Options, **Include debug information in the generated executable module**
   - Under Miscellaneous Options, **Define macros**. In the **Macro name** field, enter `IC_TRACE_DEVELOP`
3. Build your application.
4. When you run your compiled application, redirect the output to a file. For example, run myapp.exe as follows on OS/2 and Windows NT:

   ```
   myapp >myapp.out 2>&1
   ```

   Run myapp as follows on AIX:

   ```
   myapp >myapp.out
   ```

5. Browse the output file (myapp.out) to see what connections were run and in what order. The file also lists any exceptions that were thrown.

   RELATED TASKS

   "Chapter 14. Generating Code" on page 59

   Changing a Build Setup

   "Chapter 17. Fixing Problems with Your Visually Constructed Application" on page 67

Chapter 17. Fixing Problems with Your Visually Constructed Application

If you have problems running your application, check first for the following situations:

- Check the order in which connections occur within the part. Even if they seem correct in the Composition Editor, enable tracing, recompile, and run the application in debug mode.
- If you mix static and dynamic linking on OS/2 or Windows, the resulting application might behave unpredictably. We recommend you use DLLs and link dynamically. Alternatively, compile and link one executable file statically with no DLLs. Do not mix static and dynamic libraries.
- When you disable notification on collection parts, unpredictable results occur. For example, if you use a sequence part to manage objects in a container and then disable notification in the sequence, the container is not notified of the change in the sequence and is therefore not refreshed.

RELATED CONCEPTS

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Chapter 14. Generating Code” on page 59

“Chapter 16. Debugging Connections” on page 65

Chapter 18. Managing the Parts Palette

You can modify the parts palette at any time from the Composition Editor.

RELATED CONCEPTS

"Parts Palette" on page 10

RELATED TASKS

"Preparing Icons for the Parts Palette"

"Adding a Category to the Parts Palette"

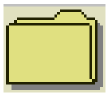
"Adding a Part to the Palette" on page 70

"Saving Parts Palette Changes" on page 71

"Deleting a Part or Category from the Palette" on page 71

Preparing Icons for the Parts Palette

Each category and part on the Composition Editor parts palette is represented by a bitmap so you can recognize it visually. 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.



To prepare bitmaps for use with the Composition Editor:

1. Create your icons using an icon editor. In OS/2, for example, 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 library that contains your icons.
3. Place the resource library in the proper directory: for OS/2 or AIX in a directory in your LIBPATH statement, and for Windows in a directory in your PATH statement.

RELATED CONCEPTS

"Parts Palette" on page 10

RELATED TASKS

"Adding a Category to the Parts Palette"

"Adding a Part to the Palette" on page 70

"Deleting a Part or Category from the Palette" on page 71

"Saving Parts Palette Changes" on page 71

Adding a Category to the Parts Palette

Once you have prepared an icon in a resource library, you can add a new category to the parts palette.

1. From the Composition Editor, select **Options**, **Modify palette**, and then **Add new category**. The Add palette category window appears.
2. In the **Category name** field, enter the name for your category. When you select your new category on the parts palette, the Composition Editor displays this text in the information area.
3. From the Category Graphic group, select the category of graphic.
4. Enter the name of your resource library in the **Module name** field.

   Note: Only enter the resource library file extension in the **Module name** field for AIX platforms.
5. In the **Opened Category Graphic** group, follow the same procedure to specify the graphic you want to represent the opened category.
6. Select **OK**.

If you do not specify a resource library, the Composition Editor uses the default icon. ❓

If you specify a resource library the Composition Editor cannot locate, it uses the question mark icon. If the question mark icon appears, make sure the following conditions are met:

- The resource library exists and is in the LIBPATH for OS/2 or PATH for Windows.
- The resource library file name is correct.
- The resource ID for the icon (in the .rc file) exists in the resource library.

RELATED CONCEPTS

“Parts Palette” on page 10

RELATED TASKS

“Preparing Icons for the Parts Palette” on page 69

“Adding a Part to the Palette”

“Saving Parts Palette Changes” on page 71

“Deleting a Part or Category from the Palette” on page 71

Adding a Part to the Palette

You can add the part that you are currently editing to any category on the parts palette.

1. Open the part in the Composition Editor.
2. Select **Options**, **Modify palette**, and **Add new part**. The Add to Palette window appears.
3. Enter the name of the part in the **Name** field.
4. Select a category for the new part.
5. Select **OK**. The Composition Editor adds the part to the category on the parts palette.

RELATED CONCEPTS

“Parts Palette” on page 10

RELATED TASKS

“Preparing Icons for the Parts Palette” on page 69

“Adding a Category to the Parts Palette” on page 69

“Saving Parts Palette Changes”

“Deleting a Part or Category from the Palette”

Deleting a Part or Category from the Palette

To remove a part from the parts palette:

1. Select the part on the parts palette.
2. Select **Modify palette** and then **Remove part** from the **Options** pull-down menu. The selected part is removed from the parts palette.

To remove a category from the parts palette:

1. Select the category on the parts palette.
2. Select the **Options** pull-down menu.
3. Select **Modify palette** and then **Remove category**. The selected category and all of the parts in it are removed from the parts palette.

RELATED CONCEPTS

“Parts Palette” on page 10

RELATED TASKS

“Preparing Icons for the Parts Palette” on page 69

“Adding a Category to the Parts Palette” on page 69

“Adding a Part to the Palette” on page 70

“Saving Parts Palette Changes”

Saving Parts Palette Changes

The Composition Editor automatically saves all parts palette changes for you. When you create a new category or part, the Composition Editor stores information about that category or part in a file named *vbpalet.dat*, which is stored in your startup directory. This file is written automatically.

Once VisualAge creates the *vbpalet.dat* file, it reads it each time you start VisualAge. Therefore, the new categories or parts that you added are always included, and those that you removed are prevented from appearing on the parts palette.

Note: If you update the icon associated with a part, the parts palette is updated the next time you select the category where the icon appears.

RELATED CONCEPTS

“Parts Palette” on page 10

RELATED TASKS

“Chapter 18. Managing the Parts Palette” on page 69

“Adding a Part to the Palette” on page 70

“Adding a Category to the Parts Palette” on page 69

“Deleting a Part or Category from the Palette” on page 71

Chapter 19. Defining Part Interfaces for Visual Composition

You can define the part interface in the following ways:

- Create a new part based on a class with features you need. The new part inherits the features of the class it extends.
- Add features to a part in the Part Interface Editor. You can add features to extend the inherited feature set, to override inherited features, or both.
- Promote features of embedded parts to the interface of a composite part.
- Describe parts and features with part information statements. This is useful primarily for existing code that you want to use in visual composition. You can define any of the following constructs in this manner:
  - A C++ class that you want to use as a part
  - A group of C functions that you want to use as a part
  - An enumeration or type definition that you want to use with parts

RELATED CONCEPTS

“Chapter 25. About Defining Part Interfaces” on page 103

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Adding Features to the Part Interface”

“Promoting Embedded Part Features in the Composition Editor” on page 76

“Promoting Embedded Part Features in the Part Interface Editor” on page 77

“Defining the Part Interface Using Part Information Statements” on page 79

“Specifying Preferred Features” on page 78

Adding Features to the Part Interface

Use the Part Interface Editor to add features to the part interface. Define features on the following tabs:

- Attribute
- Event
- Action

The defined feature tabs list features that have been defined for a part. These tabs do not list either promoted or inherited features. Features that have been promoted from embedded parts are listed on the Promote tab. Both promoted and inherited features are listed on the defined feature tabs of the parts that define the features.

You can add features to the part connection menu on the Preferred tab.

RELATED TASKS

“Adding Attribute Features” on page 74

“Adding Action Features” on page 75

“Adding Event Features” on page 75

“Specifying Preferred Features” on page 78

Adding Attribute Features

Add each new attribute on the Attribute tab as follows:

1. Enter a name in the **Attribute name** entry field.
2. If you want to add the attribute with default values for all fields, select **Add with defaults**. The attribute is added to the part interface and to the **Attribute name** list.
3. If you do not add the attribute with default values for all fields, do the following:
   - Select **Defaults** to have VisualAge produce default values for the following fields:
     - **Attribute type**
     - **Get member function**
     - **Set member function**
     - **Event identification**
   - If you want to specify a different class or data type for the attribute, either select a choice in the **Attribute type** drop-down list or enter a different type in the entry field.
   - If you changed the attribute type, select **Defaults** again to update the get and set member function signatures.
   - If you do not want the attribute value to be retrievable, clear the **Get member function** field. If you want to change the default signature, edit the value. The get member function provides the attribute value for connections.
   - If you do not want the attribute value to be changeable, clear the **Set member function** field. If you want to change the default signature, edit the value. The set member function changes the attribute value for connections.
   - If you do not want the attribute to notify other parts when the value changes, clear the **Event identification** field. If you want to change the default identifier, edit the value.
   - If you want a text description of the attribute, either enter text in the **Description** field or select **Edit** to specify a resource ID for translatable text. This description appears when you select the attribute in the Feature Implementation Browser window.
   - If you want to hide the attribute from the part property sheet, select *No* for the **Property** field in the **Constraints** group.
   - If you want to hide the attribute from the part connection list, select *No* for the **Connectable** field in the **Constraints** group.
   - If you want to prevent definition of the attribute for some system platforms, select relevant check boxes in the **Platform constraints** group.
   - Select **Add** to add the attribute to the part interface. The attribute is also added to the **Attribute name** list.
4. Select **Clear** to clear the input fields before adding the next attribute.

When you have finished adding features to the part interface, select **Save** from the **File** menu to save the part.

RELATED CONCEPTS

“Naming Guidelines for Parts and Their Features” on page 97

RELATED TASKS

“Specifying Hidden Features” on page 78

“Adding Action Features” on page 75

“Adding Event Features”

“Adding Features to the Part Interface” on page 73

Adding Event Features

Define event features to represent the occurrence of any events in your part that you want other parts to be aware of.

Add each new event on the Event tab as follows:

1. Enter a name in the **Event name** entry field.
2. If you want to add the event with default values for all fields, select **Add with defaults**. The event is added to the part interface and to the **Event name** list.
3. If you do not add the event with default values for all fields, do the following:
   - Select **Defaults** to have VisualAge produce default values for the following fields:
     - **Event identification**
   - If you want to change the default identifier, edit the value in the **Event identification** field. The identifier is used to specify what event occurred for connections.
   - If you want to pass a parameter in event data, specify the parameter data type in the **Event parameter type** field.
   - If you want a text description of the event, either enter text in the **Description** field or select **Edit** to specify a resource ID for translatable text. This description appears when you select the event in the Feature Implementation Browser window.
   - If you want to prevent definition of the event for some system platforms, select check boxes for relevant **Platform constraints** fields.
   - Select **Add** to add the event to the part interface. The event is also added to the **Event name** list.
4. Select **Clear** to clear the input fields before adding the next event.

When you have finished adding features to the part interface, select **Save** from the **File** menu to save the part.

RELATED CONCEPTS

“Naming Guidelines for Parts and Their Features” on page 97

RELATED TASKS

“Specifying Hidden Features” on page 78

“Adding Attribute Features” on page 74

“Adding Action Features”

“Adding Features to the Part Interface” on page 73

Adding Action Features

Define action features to represent part behaviors or functions that you want other parts to have access to.

Add each new action on the Action tab as follows:

1. Enter a name in the **Action name** entry field.

2. If you want to add the action with default values for all fields, select **Add with defaults**. The action is added to the part interface and to the **Action name** list.
3. If you do not add the action with default values for all fields, do the following:
   - Select **Defaults** to have VisualAge produce default values for the following fields:
     – **Action member function**
     – **Return type**
   - If you want to change the default action member function signature, edit the **Action member function** value. The action member function performs the action for connections.
   - If you do not want the action member function return value to be available for connections, clear the **Return type** field. If you change the return type, be sure that it is compatible with the action member function signature. The return value is connectable through the *actionResult* event.
   - If you want a text description of the action, either enter text in the **Description** field or select **Edit** to specify a resource ID for translatable text. This description appears when you select the action in the Feature Implementation Browser window.
   - If you want to hide the action from the part connection list, select *No* for the **Connectable** field in the **Constraints** group.
   - If you want to prevent definition of the action for some system platforms, select relevant check boxes in the **Platform constraints** group.
   - Select **Add** to add the action to the part interface. The action is also added to the **Action name** list.
4. Select **Clear** to clear the input fields before adding the next action.

When you have finished adding features to the part interface, select **Save** from the **File** menu to save the part.

RELATED CONCEPTS

"Naming Guidelines for Parts and Their Features" on page 97

RELATED TASKS

"Specifying Hidden Features" on page 78

"Adding Attribute Features" on page 74

"Adding Event Features" on page 75

"Adding Features to the Part Interface" on page 73

Promoting Embedded Part Features in the Composition Editor

You can promote features from an embedded part in the Composition Editor when you are composing the composite part. You can also promote part features in the Part Interface Editor when you are defining the interface of a composite part.

Promote features in the Composition Editor as follows:

1. From the pop-up menu of the embedded part, select **Promote part feature** to open the Promote Features window.
2. For each feature that you want to promote, do the following:

   a. In the **Action**, **Attribute**, or **Event** list box, select the feature you are promoting. A default feature name is placed in the **Promote feature name** field.
   b. If you do not want to use the default name, change the feature name in the **Promote feature name** field.
   c. Select **Promote**. The feature is added to the **Previously promoted** list.
3. Select **OK** to close the Promote Features window.
4. When you have finished promoting features to the part interface, select **Save** from the **File** menu to save the part.

RELATED CONCEPTS

"About Promoting Part Features" on page 103

"Default Promoted Feature Names" on page 104

RELATED TASKS

"Editing Parts within a Composite Part" on page 29

"Promoting Embedded Part Features in the Part Interface Editor"

"Chapter 19. Defining Part Interfaces for Visual Composition" on page 73

Promoting Embedded Part Features in the Part Interface Editor

You can promote features from an embedded part in the Part Interface Editor when you are defining the interface of a composite part. You can also promote part features in the Composition Editor when you are composing the composite part.

Promote features of an embedded part in the Part Interface Editor as follows:

1. Select the Promote tab.
2. For each feature that you want to promote, do the following:
   a. In the **Subpart name** list box, select the embedded part.
   b. In the **Feature type** list box, select the feature type.
   c. In the **Promotable feature** list box, select the feature you are promoting.
   d. To add the feature with a default name, select **Add with defaults**. Otherwise, enter a feature name in the **Promote feature name** field and select **Add**. The feature name is added to the **Promoted features** list.
3. When you have finished promoting features to the part interface, select **Save** from the **File** menu to save the part.

RELATED CONCEPTS

"About Promoting Part Features" on page 103

"Default Promoted Feature Names" on page 104

RELATED TASKS

"Promoting Embedded Part Features in the Composition Editor" on page 76

"Chapter 19. Defining Part Interfaces for Visual Composition" on page 73

Specifying Preferred Features

You can designate *preferred features* that you expect to connect to most often. You can select a preferred feature from the **Connect** menu for a part. To connect to a nonpreferred feature, you must select **All Features** from the **Connect** menu, then select the feature from all connectable features for the part.

To designate a feature as preferred, do the following:

1. Open the Part Interface Editor for the part.
2. Select the Preferred tab.
3. Select the attribute, action, or event that you want to add to the preferred list.
4. Select **Add** to add the selected feature to the preferred list.
5. Select **Save** from the **File** menu to save the part.

RELATED TASKS

“Connecting Features to Other Features” on page 45

“Adding Features to the Part Interface” on page 73

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

Specifying Hidden Features

You can designate *hidden* attributes or actions that you do not want to be available for property settings and connections. These are features that you use within the part for implementation that you do not want exposed.

To designate a feature as hidden, do the following:

1. Open the Part Interface Editor for the part.
2. For an attribute, do the following:
   - Select the Attribute tab.
   - Select the attribute in the **Attribute name** field.
   - Select *No* for the **Property** field in the **Constraints** group.
   - Select *No* for the **Connectable** field in the **Constraints** group.
3. For an action, do the following:
   - Select the Action tab.
   - Select the action in the **Action name** field.
   - Select *No* for the **Connectable** field in the **Constraints** group.
4. Select **Save** from the **File** menu to save the part.

RELATED TASKS

“Adding Features to the Part Interface” on page 73

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

Defining the Part Interface Using Part Information Statements

If C++ code already exists for your application, you can more efficiently define the part interface using part information statements. This involves the following steps:

1. Determine the part features.
2. Create a part information file using a text editor.
3. Import the part information file.

To write part information statements, you need to be aware of rules for statement composition. Some rules apply to all part information. Others rules are specific to statements for particular kinds of part information.

The following rules apply to all part information:

- All part information code lines begin with //VB in columns 1-4. Between statements, lines that do not start with //VB are ignored.
- You can arrange a statement on a single line or continue it on multiple lines by using the VB continuation statement.
- All part, class, function group, enumeration, and type definition names must be unique.
- A single file can contain information about multiple constructs. The VisualAge IDE for OS/2 and Windows works with only one part per file, but you can include multiple enumerations and type definitions used by the part.

  You can package multiple parts in a part interface (.vbe) information file, but the VisualAge IDE for OS/2 and Windows recognizes only the first part in the file. To indicate that there are multiple parts in the file, you can define an informational part as the first part in the file. Specify the noAdd and noBaseClass constraints for the informational part to prevent its use in the Composition Editor. The informational part can be opened in the IDE, providing information that it represents a file with multiple parts. All other parts in the file are available in the Add Part window. For AIX, all other parts are also available in the Visual Builder window. See the vbcc.vbe and vbmm.vbe files in the ivb directory for examples of informational part definition.
- The part information about a specific part, class, function group, enumeration, or type definition must be contained in a single file.
- You cannot begin one type of information until you have ended another. For example, you cannot put a VBBeginEnumInfo statement between VBBeginPartInfo and VBEndPartInfo statements. If you do, you will not be able to import the part information.
- Part information statement parameters are separated by commas. When you omit optional parameters between specified parameters, you must include intervening commas. You do not have to include commas after the last specified parameter. For example, you can use the VBComposerInfo statement to specify different types of support information, only one of which is required. The following statements are acceptable:

  ```
  //VBComposerInfo: visual
  //VBComposerInfo: visual,803,cppres,primitive
  //VBComposerInfo: visual,,,primitive
  ```

  The following statement is not acceptable:

  ```
  //VBComposerInfo: visual,primitive
  ```

RELATED TASKS

"Adding Part Information for Visual Composition" on page 90

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

Chapter 20. Deriving from Visual Classes

Use this procedure to set up behavioral support on the free-form surface for parts not contained in the IBM Open Class runtime libraries shipped with VisualAge. For best results, read the conceptual topics listed below.

1. If the part is a composite, generate part source (.cpp) and interface source (.cp).
2. Otherwise, define the part interface in VisualAge and generate interface source.
3. Export the part's interface (.vbe).
4. Compile the part source (.cpp and .rc, if applicable) into runtime and string resource libraries.
5. Compile the interface source (.cp) into an interface library. You must have this in order to work with the part on the free-form surface.

RELATED CONCEPTS

"About Implementing Custom Visual Parts" on page 101

"Chapter 27. Generated Code" on page 111

RELATED TASKS

"Chapter 19. Defining Part Interfaces for Visual Composition" on page 73

"Generating User Interface Source" on page 61

"Sharing Custom Visual Parts" on page 89

Chapter 21. Incorporating User-Written Code into Visual Composites

Although VisualAge enables you to compose and generate user interface parts, you will probably want to write other parts yourself at some point. These are typically nonvisual parts that provide business logic. You can either create a new part and write the code to support its features (“Assembling a Part from Generated and User-Written Code” on page 84), or you can define part interface features for code you have already written (“Adapting User-Written Classes for Use as Parts” on page 86). If you intend to export your part for use by others, read “Chapter 22. Sharing Parts with Other Programmers” on page 89.

If you just need to extend the function of the composite, you can probably accomplish this by using member function or custom logic connections. As a last resort, you can modify generated code for the visual composite.

RELATED CONCEPTS

“Chapter 3. Visual, Nonvisual, and Composite Parts” on page 5

“Member Function Connections” on page 108

“Custom Logic Connections” on page 109

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Using Handwritten Code from Previous Versions of VisualAge”

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

“Specifying Code Files” on page 85

“Generating Feature Code” on page 85

“Modifying Generated Feature Code” on page 86

“Adding Part Information for Visual Composition” on page 90

Using Handwritten Code from Previous Versions of VisualAge

If your application contains user-written code from Versions 3.0 or 3.5, you might have to update it so that it meshes properly with generated code. In addition, several key member functions are being marked obsolete, so although you are not required to update your code in this release, you might want to do it anyway. A general list of changes that affect user-written code follows.

- Several IBM Open Class part features have been changed, removed, or renamed. Consider reviewing part documentation for changes that affect your application. In particular, template support has been added, affecting the interfaces of the ICollectionViewComboBox, ICollectionViewListBox, and IVBContainerControl parts.
- Most embedded parts are now instantiated in an implementation class separate from the composite class. As a result, any user-written code referring to these embedded parts must be updated.

  For example, suppose member function or custom logic code sets the label of an IPushButton* part, named MyButton, that is embedded into a part called MyPart. The current code looks something like this:

```
iMyButton -> setText("Update");
```

Generated code for the push button now appears in an implementation class associated with MyPart called fImpl, so your code needs to look like this:

```
fImpl -> iMyButton() -> setText("Update");
```

- If your handwritten code calls initializePart( ), remove the call. This function is no longer used to enable notification and signal the *ready* event.
- If the name of a composite is the same as that of its primary part, VisualAge changes the name of the primary part to force uniqueness. It does this by appending the word *Part* to the end of the name. For example, a primary part called MyWindow is renamed automatically to MyWindowPart. However, any user-written code referring to this part must be updated manually.
- Use of the INotificationEvent class for passing event data in the notification signal is being phased out. Its replacement, INotificationEventFor, is a template, enabling support for event data other than pointers. Instances of INotificationEvent commonly appear in generated feature code for attributes; if you have modified the code to pass event data, consider updating the call to use the new class.

  The current event signal looks something like this:

  ```
  notifyObservers(INotificationEvent(SomeClass::someId, *this, true, (void*)&someEventData));
  ```

  The preferred form for an event signal now looks something like this:

  ```
  notifyObservers(INotificationEventFor<someEventDataType>(IInterest(*this, someId), someEventData)
  ```

RELATED CONCEPTS

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Chapter 8. Using Parts from Previous Versions of VisualAge” on page 23

Assembling a Part from Generated and User-Written Code

1. Design the part interface.
2. Define the part interface in the Part Interface Editor.
3. Specify code files for the part interface features.
4. Generate feature code for the part interface.
5. Modify the feature code to provide the behavior you want.

RELATED CONCEPTS

“Chapter 24. Part Design Guidelines” on page 97

“Chapter 3. Visual, Nonvisual, and Composite Parts” on page 5

RELATED TASKS

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

“Specifying Code Files” on page 85

“Generating Feature Code” on page 85

“Modifying Generated Feature Code” on page 86

“Adding Part Information for Visual Composition” on page 90

Specifying Code Files

Before you can generate feature code for a part, you must specify your code files as follows:

1. Open the Class Editor for the part.
2. Specify the .hpv and .cpv files for the feature code by filling in the User .hpv file and User .cpv file fields.

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

1. Change the file extension of .cpp files to .cpv.
2. Change the file extension of .hpp files to .hpv.
3. Change the file extension of any .rc files to .rcv.
4. Add these files in the Class Editor.

RELATED TASKS

“Chapter 21. Incorporating User-Written Code into Visual Composites” on page 83

“Generating Feature Code”

“Modifying Generated Feature Code” on page 86

Generating Feature Code

You can have VisualAge generate part source code and feature code for your part. If you generate part source code, you should generate feature code separately. Each time you generate part source code, VisualAge replaces existing code because there is no need for you to modify it. Each time you generate feature code, however, VisualAge appends the newly generated code to the end of the existing feature code. This is done so that you will not lose any code that you have written.

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

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

To generate feature code, do the following:

1. Select **File**, then select **Save and generate**, then select **Feature source**. The Generate Feature Source Code window appears.
2. Generate the feature code using one of the following methods:
   - Select the **Generate all** push button to generate feature code for member functions and data members.
   - Select the appropriate member functions and data members from the **Member functions**, **Attribute data members**, or **Event data members** list boxes. Then, select the **Generate selected** push button.

RELATED CONCEPTS

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Chapter 21. Incorporating User-Written Code into Visual Composites” on page 83

“Specifying Code Files” on page 85

“Modifying Generated Feature Code”

Modifying Generated Feature Code

For attributes, generated feature code is usually sufficient without modification. For action features, you must modify the feature code to add the behavior you want your part to provide.

You can modify the feature code using your favorite editor. The code is in the files that you specified in the Class Editor.

If you need to modify the signature for a member function that supports a feature, follow these steps:

1. Remove the feature in the the page.
2. Modify or replace the member function in the page.
3. Add the feature again in the the page.

RELATED TASKS

“Using Handwritten Code from Previous Versions of VisualAge” on page 83

“Chapter 21. Incorporating User-Written Code into Visual Composites” on page 83

“Specifying Code Files” on page 85

“Generating Feature Code” on page 85

Adapting User-Written Classes for Use as Parts

This topic is geared toward classes that would become nonvisual parts. To adapt GUI classes for use as visual parts, start at “Chapter 20. Deriving from Visual Classes” on page 81 and read the topic that suits your particular circumstance.

1. Design the part interface.
2. Modify your code to support the part interface.
3. Define the part interface using part information statements.
4. Import the part information.
5. Extend the part interface with new features as needed.

If you add a new action feature with the same name as a member function you have already written, VisualAge replaces the existing member function.

RELATED CONCEPTS

“Chapter 24. Part Design Guidelines” on page 97

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

“Specifying Code Files” on page 85

“Adding Part Information for Visual Composition” on page 90

“Assembling a Part from Generated and User-Written Code” on page 84

Chapter 22. Sharing Parts with Other Programmers

Well-designed parts can be reused. To use a part provided by another programmer, add its part information file to the list of part source files you are working with. To share a part with other programmers, package its part information with dependent source and library files.

To share a finished part for reuse, do the following:

1. Export the part interface
2. Package part files for reuse
3. Distribute the package

To share a part for further development, do the following:

1. Package part files for development
2. Distribute the package

RELATED CONCEPTS

"Chapter 2. How Classes and Parts Are Related" on page 3

RELATED TASKS

"Chapter 19. Defining Part Interfaces for Visual Composition" on page 73

"Sharing Custom Visual Parts"

"Sharing Nonextendable Parts" on page 90

"Adding Part Information for Visual Composition" on page 90

"Exporting a Part Interface for Reuse" on page 91

"Packaging Part Files for Distribution" on page 92

Sharing Custom Visual Parts

To provide a custom visual part for other programmers to embed in visual composite parts, do the following:

1. Export the part interface
2. Package as *visual* part files for reuse
3. Distribute the package

Provide an interface library so the part consumer can do the following:

- Make property settings changes to the part that are reflected in the Composition Editor as well as in generated code.
- Add and visually compose embedded parts in the custom visual part if it is a composer.
- Derive parts from the part.

RELATED CONCEPTS

"About Implementing Custom Visual Parts" on page 101

"Generated Interface Code" on page 113

RELATED TASKS

"Chapter 20. Deriving from Visual Classes" on page 81

"Exporting a Part Interface for Reuse" on page 91

"Packaging Part Files for Distribution" on page 92

"Chapter 22. Sharing Parts with Other Programmers" on page 89

Sharing Nonextendable Parts

To provide a part that is not visually modifiable or extendable, do the following:

1. Export the part interface
2. Package as *nonvisual* part files for reuse
3. Distribute the package

The part consumer will *not* be able to do the following:

- Make property settings changes to the part that are reflected in the Composition Editor. However, property settings are applied to generated code.
- Add and visually compose embedded parts in the part.
- Derive parts from the part.

RELATED TASKS

"Exporting a Part Interface for Reuse" on page 91

"Packaging Part Files for Distribution" on page 92

"Chapter 22. Sharing Parts with Other Programmers" on page 89

Adding Part Information for Visual Composition

To reuse a part, you need to add its part information file to your list of part source files. You can add either of the following types of part information files:

- A part interface (.vbe) file, for a part that you want to use in visual composition.
- A part implementation (.vbf) file, for a part that is under development.

On Windows or OS/2, add a new target based on existing part information files as follows:

1. From the **Project Workbook** menu in the IDE, select **Add Target**. Then, select **Executable(EXE)** or **Dynamic Link Library(DLL)** to open a Target SmartGuide.
2. Select **Next** in each window until you reach the Source Files window.
3. In the Source Files window, do the following for each part information file that you want to add.
   - Select the part information file
   - Select **Add**
4. When you have added the part information files you want in the Source Files window, select **Finish** to add the target.

On Windows or OS/2, add part information files to an existing target as follows:

1. Open the target pop-up menu from any of the following views in the Configuration section of the IDE:
   - The **Targets** view of the Targets page
   - The **Source and Targets** view of the Options page

   - The **Details** view of the Advanced page
2. From the target pop-up menu, select **Add source into** to open the Add Source into window.
3. In the Add Source into window, do the following for each part information file that you want to add.
   a. Select **Find** to open a file dialog
   b. Select the part information file
   c. Select **Add**
4. When you have added the part information files you want in the Add Source into window, select **Apply** to add the files to the target.

On AIX, add a part information file as follows:

1. From the **File** menu of the Visual Builder window, select **Load** to open a file dialog.
2. Select the part information file that you want to load.
3. Select the **OK** push button to load the part information. The part information file is added to the list of part source files in the **Loaded Part Files** list box. Any visual parts in the part information file are displayed in the **Visual Parts** list box, and any nonvisual parts and class interface parts are displayed in the **Nonvisual Parts** list box.

If you add part information from a part interface (.vbe) file, you must have the following files:

- The .hpp header file for the part. If you have an .hpv header file for the part, include that as well.
- Resource and runtime libraries for the part.
- For custom visual primitives, an interface library for working with the part on the free-form surface.

RELATED TASKS

Add or Change Targets

“Exporting a Part Interface for Reuse”

“Packaging Part Files for Distribution” on page 92

“Chapter 22. Sharing Parts with Other Programmers” on page 89

Exporting a Part Interface for Reuse

On Windows or OS/2, export a part interface as follows:

1. Open the part pop-up menu in the **Parts** view of the Parts page in the Project section of the IDE.
2. From the part pop-up menu, select **Export Part**. The part interface is saved in a part interface (.vbe) file with a name based on the part name.

On AIX, export a part interface as follows:

1. In the Visual Builder window, select the part or parts whose information you want to export. You can select parts in either the **Visual Parts** list box, the **Nonvisual Parts** list box, or both.
2. Select **Part**, then select **Export interface** to open a file dialog.

3. In the **Save as filename** field, either use the default name or specify the name of the part interface (.vbe) file in which you want to save the part information.
   - If you select a single part, the default name is based on the part name.
   - If you select multiple parts, the default name is Exported.vbe.
4. Select the **OK** push button. The part interface is saved in the file you specify.

The part interface file does not contain any information about the implementation of the part. The part can be used in visual composition, but cannot be edited.

RELATED TASKS

"Adding Part Information for Visual Composition" on page 90

"Packaging Part Files for Distribution"

"Chapter 22. Sharing Parts with Other Programmers" on page 89

Packaging Part Files for Distribution

Before you distribute a finished part for reuse, you must export its interface to a part interface (.vbe) information file. If you are distributing an unfinished part for further development, use its part implementation (.vbf) information file. Package the part information file with dependent files used by the part. The following factors determine what file types you need to package:

- Are you distributing the part for reuse or for further development?
- Is the part visual or nonvisual?
- Are you providing translatable text?

The following table summarizes the code file types you need. You should also provide documentation, including installation or build instructions and any information about how to add the part to the parts palette.

File type	**Package for reuse**		**Package for further development**
	Nonvisual	**Visual**	
Part interface (.vbe) or implementation (.vbf) file	Yes (.vbe)	Yes (.vbe)	Yes (.vbf)
C++ header (.hpp, .hpv) files	Yes	Yes	Yes (.hpv)
C++ user source (.cpv) files	No	No	Yes
Resource header (.h, .hv) and source (.rc, .rcv) files	No	Yes, for text that the part consumer can translate	Yes (.hv, .rcv)
Resource libraries	Yes, for part palette and composition surface icons	Yes, for icons, bitmaps, and text (including any translated text that you provide)	No
Interface library	No	Yes	No
Runtime library	Yes	Yes	No

File type	Package for reuse		Package for further development
	Nonvisual	Visual	
Import library (.lib) file	Yes, for dynamic linking on OS/2 and Windows	Yes, for dynamic linking on OS/2 and Windows	No

RELATED TASKS

“Exporting a Part Interface for Reuse” on page 91

“Adding Part Information for Visual Composition” on page 90

“Chapter 22. Sharing Parts with Other Programmers” on page 89

Chapter 23. Packaging Visually Composed Applications

Before you distribute an application, you must package your work with any files that you use. Your application uses resources such as programming language services, help files, translatable text, bitmaps, and menus. In addition to resources that you create, you should package the following VisualAge resources with visually composed applications:

- For Windows and OS/2, package C runtime libraries:
  - cppzmi40.dll
  - cpprmi40.dll
  - cpprbi40.dll
- If you use Open Class resources, package a renamed copy of the Open Class resource library. This library contains resources such as default bitmaps for common tool bar buttons.
  - For Windows, copy cpporr40.dll
  - For OS/2, copy cpporr40.dll
  - For AIX, copy libvacocres.o

  Specify the renamed library for your application in the *userInterfaceResourceLibrary* property of its IVBMain* part.
- Package Open Class runtime libraries. For Windows and OS/2, package the following libraries:
  - cppobi40.dll—for all programs
  - cppoui40.dll—for programs with user interface classes
  - cppogi40.dll—for programs with 2-dimension graphics classes

  For AIX, package the following libraries:
  - libvacbase.a—for all programs
  - libvacui.a—for programs user interface classes
  - libvacgraph2d.a—for programs with 2-dimension graphics classes
- Package runtime libraries for visually composed parts. For Windows and OS/2, package the following libraries:
  - cppvpi40.dll—for all programs
  - cppvoi40.dll—for programs with visual parts

  For AIX, package the following libraries:
  - libvacvngui.a—for all programs
  - libvacvgui.a—for programs with visual parts

Your license type determines which libraries you are authorized to redistribute. If you are not sure, check the license card that came with your copy of the product.

RELATED CONCEPTS

"Generated Interface Code" on page 113

Chapter 24. Part Design Guidelines

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?

To design a new part, do the following:

1. Determine the attributes (properties) of the part.
2. Determine the change-in-state notifications (events) that the part will signal.
3. Determine the behaviors (actions) for the part. To minimize the number of connections that the users of your part will have to make, require as few input parameters as possible.
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.

RELATED CONCEPTS

“Chapter 3. Visual, Nonvisual, and Composite Parts” on page 5

“Naming Guidelines for Parts and Their Features”

Naming Guidelines for Parts and Their Features

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

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

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

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

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

The main place that users see your action, attribute, and event names is on the **Connect** 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 often 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.

RELATED CONCEPTS

"About Implementing Nonvisual Parts" on page 99

"About Implementing Custom Visual Parts" on page 101

Part Implementation

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

Make the following interface changes to support parts:

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

Make the following code changes to support parts:

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

RELATED CONCEPTS

"Chapter 24. Part Design Guidelines" on page 97

"Naming Guidelines for Parts and Their Features" on page 97

About Implementing Nonvisual Parts

The first step associated with implementing a part is positioning the part class in the C++ class hierarchy. Place your nonvisual part under the IStandardNotifier class hierarchy. Inserted in this location, your part inherits certain default behavior from IStandardNotifier and the INotifier protocol.

In some cases, you might want your nonvisual part to be abstract. Be aware that your users cannot drop abstract nonvisual parts onto the Composition Editor’s free-form surface. If you do not provide concrete parts derived from this abstract part, your users must derive their own concrete parts.

Deriving from a class that uses or contains template-based classes (such as collection classes) is possible with certain restrictions. To preserve the template implementation, you must define the part interface using a .vbe file; you cannot specify it properly through the Part Interface Editor. (If you do use the Part Interface Editor, VisualAge will issue an error message when it generates code for the part.) The alternative to using an interface file is deriving the part from an instance of the template-based class.

If the part is itself a class template, you must define any notification IDs in a separate IDs class. When used, these notification IDs must be scoped to the class in which they are implemented. (For example, to specify the *itemChanged* event in a notifyObservers call, use IVBContainerControlIds::itemChangedId. To specify the event associated with the selectedCnrElement attribute, use IContainerControl::selectId.) For examples of an IDs class, see the ivbids.hpp header file.

Otherwise, creating a class for a part is not much different from creating any other class. There are just a few additional guidelines to keep in mind for those member functions that support your part’s interface. Before you get started, please read “Constructors in Nonvisual Parts”, “Destructors in Nonvisual Parts” on page 100 , and “Assignment Operators in Nonvisual Parts” on page 100.

RELATED CONCEPTS

“Constructors in Nonvisual Parts”

“Destructors in Nonvisual Parts” on page 100

“Assignment Operators in Nonvisual Parts” on page 100

“Chapter 24. Part Design Guidelines” on page 97

“Naming Guidelines for Parts and Their Features” on page 97

“About Implementing Custom Visual Parts” on page 101

Constructors in Nonvisual Parts

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

```
IAddress();
```

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

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

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

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

The implementation of the copy constructor for `IAddress` follows:

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

RELATED CONCEPTS

“Destructors in Nonvisual Parts”

“Assignment Operators in Nonvisual Parts”

Destructors in Nonvisual Parts

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

```
virtual
   IAddress ();
```

The implementation of the `IAddress` destructor follows:

```
IAddress:: IAddress()
  {
  }
```

RELATED CONCEPTS

“Constructors in Nonvisual Parts” on page 99

“Assignment Operators in Nonvisual Parts”

Assignment Operators in Nonvisual Parts

To ensure that attribute changes are signaled, specify an assignment operator for nonvisual parts with attributes, as follows:

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

The implementation of the `IAddress` class assignment operator follows:

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

Note the following in the previous example:

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

RELATED CONCEPTS

“Constructors in Nonvisual Parts” on page 99

“Destructors in Nonvisual Parts” on page 100

About Implementing Custom Visual Parts

Implementing your own visual parts presents a problem that implementing nonvisual parts does not. Nonvisual parts are represented on the free-form surface by icons, so only a part interface specification is needed to enable the part to be used in theComposition Editor. However, most visual parts on the free-form surface are real-time class instances, so VisualAge must have access to an interface library from which to instantiate each visual part. These *custom visual* parts include the following:

- Composites to be embedded in other composites, if you intend to update them from the free-form surface. Adding this support enables you to drop parts into the embedded composite and directly set properties of the embedded composite. You do not need this support to embed composites without modification capability.
- Composites to be used as base classes for new parts. The issues are similar to what is described in the previous bullet.
- Primitives derived from IBM Open Class controls. Adding this support enables you to interact with dropped instances on the free-form surface as if they had been shipped by IBM.
- Composers derived from IBM Open Class windows. Adding this support enables you to drop parts into a Composer instance as if it were an ICanvas-based part shipped by IBM. (Some restrictions apply. Be sure to read the topics listed below.)

Deriving from a class that uses or contains template-based classes is possible with certain restrictions. To preserve the template implementation, you must define the part interface using a .vbe file; you cannot specify it properly through the Part Interface Editor. (If you do use the Part Interface Editor, VisualAge will issue an error message when it generates code for the part.) The alternative to using an interface file is deriving the part from an instance of the template-based class. These template restrictions apply to the following classes:

- IVBCollectionViewComboBox
- IVBCollectionViewListBox
- IVBContainerControl

If the part is itself a class template, you must define any notification IDs in a separate IDs class. When used, these notification IDs must be scoped to the class in which they are implemented. (For example, to specify the *itemChanged* event in a notifyObservers call, use IVBContainerControlIds::itemChangedId. To specify the event associated with the selectedCnrElement attribute, use IContainerControl::selectId.) For examples of an IDs class, see the ivbids.hpp header file.

Deriving from certain other classes (mostly Composer parts) is possible only with restrictions. For more information, see reference information for the parts you want to use as base classes.

RELATED CONCEPTS

“Chapter 24. Part Design Guidelines” on page 97

“Naming Guidelines for Parts and Their Features” on page 97

“About Implementing Nonvisual Parts” on page 99

RELATED TASKS

“Generating User Interface Source” on page 61

“Chapter 20. Deriving from Visual Classes” on page 81

Chapter 25. About Defining Part Interfaces

Through the part interface, you determine which features of your part can be connected to other parts in visual composition. You can also determine which attributes can be given initial values.

You can define the part interface in the following ways:

- Create a new part based on a class with features you need. The new part inherits the features of the class it extends.
- Add new features to a part. You can add features to extend the inherited feature set, to override inherited features, or both.
- Promote features of embedded parts to the interface of a composite part.
- Describe parts and features with part information statements. This is useful primarily for existing code that you want to use in visual composition.

RELATED CONCEPTS

“Chapter 28. VisualAge Component Model” on page 119

“Chapter 2. How Classes and Parts Are Related” on page 3

“About Promoting Part Features”

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

About Promoting Part Features

When you create a composite part, you might want some features of parts embedded within it to appear in the interface of the composite part. For example, suppose you create a composite part named ButtonSet containing a set of push buttons that you want to reuse. When you add the ButtonSet composite part to another composite, you want to be able to connect to each of the push buttons.

When you first create a composite part, its interface reflects its inheritance, not the features of the parts embedded within it. To expose features of the embedded parts, you must promote them to the composite’s interface.

Alternatively, you can promote the *this* attribute of an embedded part to add the entire part as an attribute of the composite part. Then, when you use the composite part in another composite part, you can tear off the attribute as a Variable part and connect to features of the embedded part represented by the Variable.

RELATED CONCEPTS

“Chapter 25. About Defining Part Interfaces”

“Default Promoted Feature Names” on page 104

“About Tearing Off Attributes” on page 13

RELATED TASKS

“Chapter 19. Defining Part Interfaces for Visual Composition” on page 73

Default Promoted Feature Names

When you promote a part feature, you can use a default composite part feature name produced by VisualAge. The default feature name is a combination of the name of the part you are promoting the feature from and the name of the feature you are promoting. This identifies the part that implements the feature, which is helpful if the composite part contains more than one part with the same feature. Then, when you connect to the feature, you can tell which embedded part it belongs to.

For example, if you promote the *buttonClickEvent* feature for a part named PushButton2, the default composite part feature name is *pushButton2ButtonClickEvent*.

RELATED CONCEPTS

"About Promoting Part Features" on page 103

Chapter 26. Connections

When you make a connection in the Composition Editor, you define the interaction between components. For example, if you want a data value to change when an event occurs, you would make an event-to-attribute connection. The following table summarizes the types of connections that the Composition Editor provides. The return value is supplied by the connection's *actionResult* event.

Table 3. Connection Type Summary

If you want to...	Use this connection type	Color	Does connection have a return value?
Cause one data value to change another	attribute-to-attribute	Dark blue	No
Change a data value whenever an event occurs	event-to-attribute	Dark green	Yes
Call a public behavior whenever an event occurs	event-to-action	Dark green	Yes
Call a private behavior whenever an event occurs	event-to-member function	Dark green	Yes
Call a public behavior whenever a data value changes	attribute-to-action	Dark green	Yes
Call a private behavior whenever a data value changes	attribute-to-member function	Dark green	Yes
Call customized code whenever a data value changes or an event occurs	custom logic	Light blue	Yes
Supply a value to a parameter	parameter	Violet	No

RELATED CONCEPTS

"Attribute-to-Attribute Connections" on page 106

"Attribute-to-Action Connections" on page 109

"Event-to-Attribute Connections" on page 108

"Event-to-Action Connections" on page 108

"Member Function Connections" on page 108

"Custom Logic Connections" on page 109

"Parameter Connections" on page 110

"The Source and Target of a Connection" on page 106

RELATED TASKS

"Chapter 12. Making Connections in the Composition Editor" on page 45

The Source and Target of a Connection

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*; the part on which it ends is the *target*.

Often, it does not matter which part you choose as the source or target, but there are connections where direction is important.

- In an event connection, the event is always the source. In the case of attribute-to-action connections, the source event is signaled when the attribute changes value.If you try to make an event the target, VisualAge automatically reverses it for you.
- For attribute-to-attribute connections, if only one of the attributes has a public set member function, VisualAge makes that attribute the target. This is done so that the attribute that has the public set member function can be initialized at run time.
- When you make attribute-to-attribute connections, the order in which you choose the source and target is important. The source and target attribute values may be different when the part is first initialized. If they are, VisualAge resolves the difference by changing the value of the target to match that of the source. Thereafter, if both attributes have public set member functions, the connection updates either attribute if the other changes.

The target of a connection can have a return value. If it does, you can treat the return value as a feature of the connection and use it as the source of another connection. This return value appears in the connection menu for the connection as *actionResult*.

RELATED CONCEPTS

"Chapter 26. Connections" on page 105

RELATED TASKS

"Changing the Source and Target of Connections" on page 56

Attribute-to-Attribute Connections

An *attribute-to-attribute connection* links two attribute values together. This causes the value of one attribute to change when the value of the other changes, except as noted below.

A connection of this type appears as a bidirectional dark blue line with dots at either end. The solid dot indicates the target, and the hollow dot indicates the source.

For most attribute-to-attribute connections, the target attribute is set to the value of the source attribute when your part is constructed at run time. However, deferred creation affects the timing of initialization in the following ways:

- Attribute-to-attribute connections are initialized when the target is created.
- Creation of the target forces creation of the source.
- Creation of the source does NOT force creation of the target.

Note: These connections never take parameters.

If the source of an attribute-to-attribute connection is a variable, its initialization type determines the initializaiton order.

Table 4. Initialization Order with a Variable Source

If the initialization type is...	Initialization is from...
apply	target to source
targetInit	variable source to target
broadcast	variable source to target

In order to achieve the behavior that you anticipate, you must know something about the attributes you are connecting. The following table shows the results of connecting attributes of different types.

Table 5. Behavioral Considerations for Connections

	And the target has both set member function and event...	**And the target has an event but no set member function...**	**And the target has a set member function but no event...**
If the source has both set member function and event...	Source and target values are fully synchronized.	VisualAge automatically reverses the connection.	The source initializes the target. The target is updated whenever the source's value changes.
If the source has an event but no set member function...	The source initializes the target. The target is updated whenever the source's value changes.	This connection is not valid.	The source initializes the target. The target is updated whenever the source's value changes.
If the source has a set member function but no event...	The source initializes the target only. The source is updated whenever the target's value changes.	VisualAge automatically reverses the connection.	The source initializes the target. No further updates occur.

Usage Notes:

- You can use a class interface part as the source of a connection only when making an attribute-to-attribute connection. You can use an attribute of a class interface part to initialize an attribute of another part without using notification.
- Do not create attribute-to-attribute connections in which the source is not initialized before the *ready* event is signaled. You may get a system error in the resulting application.

RELATED CONCEPTS

"The Source and Target of a Connection" on page 106

"Chapter 26. Connections" on page 105

RELATED TASKS

"Connecting Features to Other Features" on page 45

Event-to-Attribute Connections

An *event-to-attribute* connection updates the target attribute whenever the source event occurs. An event-to-attribute connection appears as a unidirectional dark green arrow ⇢ with the arrowhead pointing to the target.

The attribute must have a public set member function known to VisualAge; otherwise, you cannot make the connection. If you open settings on a connection of this type, the target of the connection appears as an action with the same name as that of the target attribute.

RELATED CONCEPTS

"Chapter 26. Connections" on page 105

RELATED TASKS

"Connecting Features to Other Features" on page 45

Event-to-Action Connections

An *event-to-action* connection calls the specified action of the target object whenever the source event occurs.

An event-to-action connection appears as a unidirectional dark green arrow ⇢ with the arrowhead pointing to the target.

To access behavior that is not part of the part interface, use member function or custom logicconnections.

RELATED CONCEPTS

"Member Function Connections"

"Custom Logic Connections" on page 109

"Chapter 26. Connections" on page 105

RELATED TASKS

"Connecting Features to Other Features" on page 45

"Supplying Parameter Values for Incomplete Connections" on page 49

Member Function Connections

A *member function* connection calls a member function of the composite part whenever the source event occurs. The event can be either a user interface event or one associated with an attribute.

This type of connection appears as a unidirectional dark green arrow ⇢ with the arrowhead pointing to the free-form surface.

A member function is a member function of a visual composite, but unlike a part action feature, a member function does not have to be accessible outside the class.Examples of potential member functions follow:

- Local member functions declared as private or protected

- Inherited member functions declared as protected

Connect to member functions instead of to actions when you want to keep the operation internal to the class. For example, suppose that the composite part needs to perform a calculation whenever a value changes. If users of the composite do not need to be aware that this is happening, use a member function.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Connecting Features to Member Functions” on page 46

Attribute-to-Action Connections

An *attribute-to-action* connection calls an action whenever the source attribute changes value. This connection is similar to an event-to-action connection, because the connection calls the action when the attribute’s event is signaled.

This type of connection appears as a unidirectional dark green arrow ⇢ with the arrowhead pointing to the target. The attribute’s value is passed as the first parameter of the action if no parameter is explicitly specified. If the action requires more than one input parameter, the connection line initially appears dashed to show that it is incomplete. To make the connection complete you must provide the input parameter. You can do this either through a parameter connection or by setting a constant parameter value.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Connecting Features to Other Features” on page 45

“Supplying Parameter Values for Incomplete Connections” on page 49

Custom Logic Connections

A custom logic connection calls your customized logic whenever the source event occurs. You can use either an event or an attribute as the source for this connection; if you use an attribute, the custom logic is called when the attribute’s event is signaled. A custom logic connection appears as a unidirectional light blue arrow with the arrowhead pointing to the edge of the free-form surface.

Use custom logic whenever your connection needs special handling. Like member function connections, custom logic connections cannot be altered by programmers using your part. Unlike member function code, which becomes a member of your composite part, your custom logic code becomes part of the connection. This means that you cannot reuse custom logic by call; you must reenter the code every time.

For example, suppose you want to test an attribute for a specific value before calling an action. Instead of connecting to the action directly, you could write custom logic on the connection: if the attribute is equal to a certain value, call the action. Otherwise, just return.

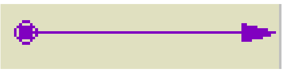
RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Connecting Features to Custom Logic” on page 47

Parameter Connections

A parameter connection supplies an input value to the target of a connection by passing either an attribute’s value or the return value from an action, member function, or custom logic. This connection appears as a unidirectional violet arrow  with the arrowhead pointing from the parameter of the original connection to the attribute or action providing the value.

If you are in doubt about a connection that you want to make, you can browse a part’s features to see the parameter names.

The original connection is always the source of a parameter connection; the source feature is the parameter itself. If you select the parameter as the target, VisualAge reverses the direction of the parameter connection automatically.

If the target of the original connection takes parameters and the same event provides parameters by default, the connection line might appear solid. This is true even if the target takes one input parameter and you have not otherwise provided one. VisualAge can use any of the following means to supply parameters with values:

- If the parameter is connected to an attribute, the connection calls the attribute’s get member function to get the attribute’s value and return it to the parameter.
- If the parameter is connected to an action, the connection code calls the action and passes the action’s return value to the parameter. The same is true when a parameter is connected to a member function or to custom logic.
- If the source of the original connection passes event data in the connection code, VisualAge applies it to the parameter. If several values are required, event data is applied to the first parameter only.
- If you specify a constant parameter value in the original connection, VisualAge passes it in the connection code.

RELATED CONCEPTS

“Chapter 26. Connections” on page 105

RELATED TASKS

“Supplying Parameter Values for Incomplete Connections” on page 49

Chapter 27. Generated Code

VisualAge can generate code for several different uses, as follows:

- C++ code for shared libraries. This includes part source code and (optionally) feature source code.
- C++ code for applications. This includes part source code, main( ) code, and (optionally) feature source code.
- C++ code for C interface libraries. You need this to use custom visual parts in the Composition Editor.
- String resource files for national language support.

RELATED CONCEPTS

“Generated Feature Code”

“Generated Part Code” on page 112

“Generated Application Code” on page 112

“Generated Interface Code” on page 113

“Resource Files Generated for Translation” on page 116

RELATED TASKS

“Chapter 14. Generating Code” on page 59

Generated Feature Code

After you have defined the interface for a new part, VisualAge can generate C++ code for the interface’s features.

Before generation, you must set generation properties in the Class Editor. In assigning file specifications, VisualAge uses the same file name root set for the part source code, but the file extensions are slightly different, as follows:

.cpv A C++ code file.

.hpv The header file for the .cpv file.

In most cases, code generated for attributes is sufficient for use without modification. Code generated for actions is skeletal but compilable—sufficient for visual construction, but you must eventually add the appropriate logic to the generated action code. Event code is automatically generated for all attributes, but if you wish to signal events from actions, you must add the notifyObservers call by hand.

VisualAge does not automatically include generated feature code in part source unless the appropriate generation options have been set in the Class Editor. You select which file types to be included. In addition to the file types listed above, VisualAge also supports inclusion of the following types:

.hv A resource header file for the .cpv file. This file contains the resource IDs you are adding by hand.

.rcv A resource file that contains any strings you are adding by hand.

RELATED CONCEPTS

“Generated Part Code” on page 112

“Generated Application Code”

“Chapter 29. Notification Framework” on page 123

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Generating Feature Code” on page 85

“Preparing for Code Generation” on page 59

Generated Part Code

For each part processed, VisualAge generates several source code files and puts them in the current working directory:

.cpp A C++ code file.

.hpp The header file for the .cpp file.

.h A resource header file for the .cpp file. This file contains the resource IDs for your part. It is generated under any of the following circumstances:

- The part is visual or contains visual parts.
- You specified a starting resource ID for the part.
- You associated a handwritten resource header file (.hv) with this part.

.rc A resource file that contains any strings used in the part for entry field labels, push buttons, menus, and so forth. This file is only generated if you have opted to have VisualAge assign resource IDs.

You must specify a starting resource ID in the Class Editor for all parts for which you want VisualAge to export translatable strings into resource files.

RELATED CONCEPTS

“Resource Files Generated for Translation” on page 116

“Generated Feature Code” on page 111

“Generated Application Code”

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Generating Source Code” on page 60

Generated Application Code

Each composite part you develop represents a piece of the finished application. At some point, you must integrate the various pieces into a runnable whole. This integration point is called the *main part*, a nonvisual composite based on the IVBMain class.

At a minimum, the main part contains the primary part (an icon representing the IVBMain class) and a composite representing the first window that appears in the finished application. To configure certain aspects of main( ), edit properties of the primary part. (An example of this is specifying an IBM Open Class resource library other than the default.)

For each main part processed, VisualAge creates several files and puts them in the current working directory:

.cpp The code file containing the main( ) function definition and part implementation.

.hpp The header file for the .cpp file.

.h A resource header file for the .cpp file. This file contains resource IDs for your part. It is generated under any of the following circumstances:

- The part is visual or contains visual parts.
- You specified a starting resource ID for the part.
- You associated a handwritten resource header file (.hv) with this part.

.rc A resource file that contains any strings used in the part. This file is only generated if you have opted to have VisualAge assign resource IDs.

RELATED CONCEPTS

“Generated Part Code” on page 112

“Chapter 4. Primary Part” on page 7

“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Generating Source Code for Your Application’s main( ) Function” on page 61

Generated Interface Code

Interface code must be generated for each custom visual part used. For each part processed, VisualAge creates a C++ code file (.cp). You then have to compile this file into the interface library needed for behavioral support in the Composition Editor.

The generated code uses types defined in the iavlbase.h header file. The most significant of these is AvlObject, a void pointer used to return an instance of your custom visual part to the Composition Editor. The part is instantiated in a constructor-within-a constructor, a function called *ClassNameClassName*.

RELATED CONCEPTS

“About Implementing Custom Visual Parts” on page 101

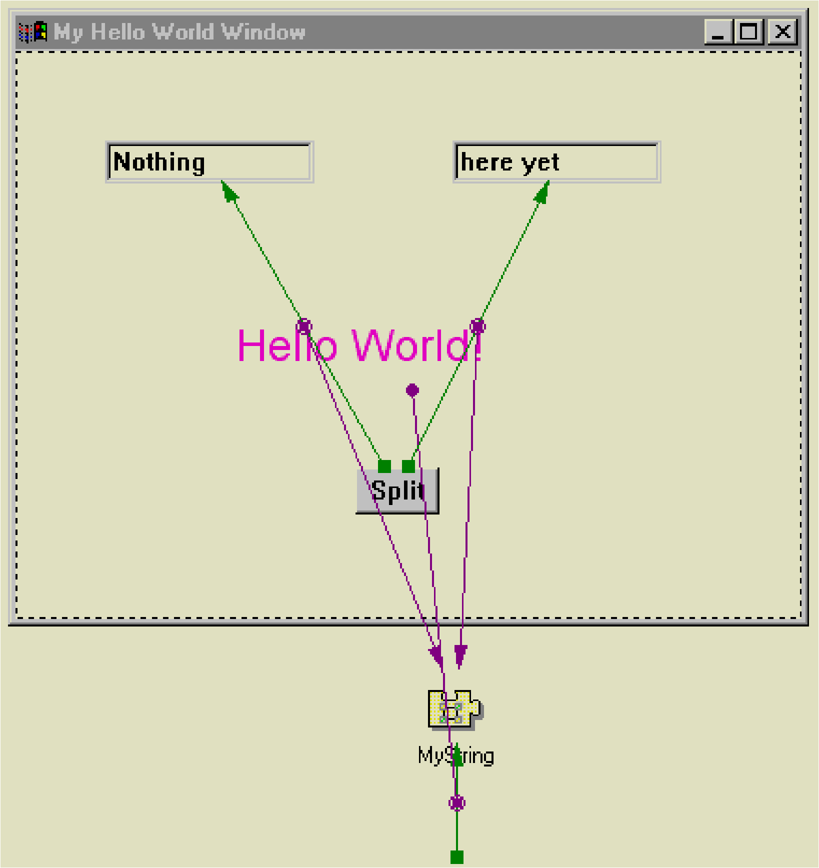
“Chapter 27. Generated Code” on page 111

RELATED TASKS

“Generating User Interface Source” on page 61

“Chapter 20. Deriving from Visual Classes” on page 81

Example of Code Generated for a Visual Composite



This simple Hello World! program puts each word of the pink text into its own entry field. This program consists of two composites, HelloWorld (the main part) and MyHelloWorldWindow (shown above).

HelloWorld contains only two parts:

- An IVBMain* part, the primary part
- A MyHelloWorldWindow* part, named MyHelloWindow

Code is generated for HelloWorld into hllworld.cpp and hllworld.hpp. The header defines only one class, HelloWorld, which is based on IVBMain.

The .cpp file for HelloWorld defines the main( ) function and the following classes:

- HelloWorldImpl, the implementation class associated with HelloWorld.
- HelloWorldConn, the connection class associated with HelloWorld. The only connection in this part is an instance of this class.

Program flow within these classes is summarized below. Obvious steps, such as instantiation of a class by its constructor, are omitted.

1. The main( ) function creates a HelloWorld instance named fiPart.
   - The constructor for HelloWorld creates an instance of HelloWorldImpl named fImpl, initializing it with a call to the constructImpl( ) function of fImpl. This instance is the only data member associated with HelloWorld.

- The constructImpl( ) function assigns a window resource to fImpl and calls the createiPart( ) function of fImpl.
- The createiPart( ) function creates an instance of HelloWorldConn named fConn0Perform and registers it as an event observer by calling the handleNotificationsFor( ) function inherited from the IObserver class.
- The constructor also signals the *ready* event. The *ready* event triggers a call to the conn0Perform( ) function, which creates an instance of MyHelloWorldWindow named fiMyHelloWorld and makes it visible.

  This happens because the HelloWorld part contains a connection between the composite's *ready* event and the *show* action of the MyHelloWorldWindow instance.

MyHelloWorldWindow contains the following parts:

- An IFrameWindow* part, the primary part
- An ICanvas* part as client
- Two IEntryField* parts, named FirstEF and SecondEF
- An IStaticText* part, named MyHelloText
- An IPushButton* part, named SplitButton
- An IString* part, named MyString

Code is generated for MyHelloWorldWindow into myhllwdw.cpp and myhllwdw.hpp. The header defines only one class, MyHelloWorldWindow, which is based on IFrameWindow.

Of all classes represented on the free-form surface, only IFrameWindow is instantiated in the MyHelloWorldWindow constructor. All others are instantiated in the scope of the implementation class described below.

The .cpp file for MyHelloWorldWindow defines the following classes:

- MyHelloWorldWindowImpl, the implementation class associated with MyHelloWorldWindow. All parts other than the primary are members of MyHelloWorldWindowImpl.
- MyHelloWorldWindowConn, the connection class associated with MyHelloWorldWindow. All connections in this composite are instances of MyHelloWorldWindowConn.

In general, these classes are related to each other in a similar way to those associated with HelloWorld. Additional notes follow.

- Connection functions exist for primary connections only. Data represented by a parameter connection is incorporated into the function for its primary connection.
- Promoted features (not shown in this sample) are implemented in code as connection class instances.

RELATED CONCEPTS

"Generated Part Code" on page 112

"Chapter 4. Primary Part" on page 7

"Chapter 27. Generated Code" on page 111

RELATED TASKS

"Chapter 14. Generating Code" on page 59

"About Promoting Part Features" on page 103

Code Generation Errors

One of the most common causes of code generation errors is changing the names of features that are connected to other features. For example, suppose feature A is connected to feature B. If you change the name of feature A and then regenerate the source code for your part, VisualAge 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.

If this message appears, the setting that corresponds to the specified attribute is not currently valid:

```
CPP.GEN.14.e: The X attribute of the Y part cannot be set.
```

If this message appears, a feature that is not currently valid is either the source or target of a connection:

```
CPP.GEN.24.e: (connection_number_and_description): The X feature was not found in the part interfac
```

If this message appears, a feature that is not currently valid is used to supply a parameter value in a connection.:

```
CPP.GEN.3.e: The X feature of the Y part has changed since this part was developed.
```

RELATED CONCEPTS

"Chapter 27. Generated Code" on page 111

RELATED TASKS

"Chapter 14. Generating Code" on page 59

Resource Files Generated for Translation

VisualAge generates the following resource files for you:

- A resource file (.rc), which contains the text strings and other program resources used in your part
- A resource header file (.h), which contains the resource ID definitions for your application

In order for VisualAge to assign most resource definitions to the .h and .rc files, you must specify a starting resource ID in the Class Editor for every part. You can either use the default value or specify an alternative, but consider the following:

- The resource ID must be a number.
- The number specified must be either high enough or low enough that the resource IDs produced do not conflict with the resource IDs that VisualAge generates for other parts that comprise your application.

For example, suppose you have a reusable Address part (a canvas with entry fields and static text) that you want to embed in your application's main window. You might give the main window part a starting resource ID of 5000 and the Address part a starting resource ID of 6000. Doing this would prevent conflicts between the resource IDs that VisualAge generates for the main window part and those it generates for the Address part.

Consider using starting resource IDs between 100 and 14500 for most applications, for the following reasons:

- The operating system has reserved many resource IDs below 100 for its own use.
- When determining resource IDs for window resources, VisualAge begins with 15000 and increments the resource ID of each successive primitive part by 5. Starting resource IDs between 100 and 14500 are low enough to prevent you from experiencing any resource ID conflicts in most cases.

RELATED CONCEPTS

"The Resource File (.rc)"

"The Resource Header File (.h)" on page 118

RELATED TASKS

"Chapter 14. Generating Code" on page 59

The Resource File (.rc)

The .rc file defines nonwindow resources. Nonwindow resources are the strings that are displayed in your composite part. Examples are window titles, static text used to label entry fields and list boxes, and the text on push buttons and menu items. These strings are delimited by quotation marks (″ ″) and can be translated into another language.

Each string that VisualAge inserts in the .rc file is preceded by a resource hame that begins with STRRC, as follows:

```
STRRC_ToDoList_FrameWindow_title,  "ToDo List"
```

VisualAge defines numeric resource IDs for these resource names in the .h file, starting with the resource ID you specified for your part in the Class Editor.

You may have strings in your part (such as the application name) that you do not want translated. If that is the case, you can prevent those text strings from being inserted in the resource file by inserting a number sign (#) at the beginning of the text and enclosing the text in quotation marks (″ ″). This change must be made in the property sheet for the part, not by direct edit in the Composition Editor.

Suppose you do not want a window title to be translated. To prevent VisualAge from inserting the title string in the resource file, edit the window's *title* property. In the case of the ToDoList part, the modified title appears as follows:

```
#"ToDo List"
```

After code generation, no resource definition for the modified title appears in the .rc file. Instead, the hardcoded string appears in the generated .cpp file as part of the defaultTitle( ) function.

RELATED CONCEPTS

"The Resource Header File (.h)" on page 118

"Resource Files Generated for Translation" on page 116

RELATED TASKS

"Chapter 14. Generating Code" on page 59

The Resource Header File (.h)

In the resource header file, VisualAge uses #define statements to assign unique resource IDs to each of the string resources listed in the .rc file. VisualAge also assigns unique window IDs to all primitive visual parts. The only resource ID that you must specify is the starting resource ID for the part; you do that in the Class Editor. During code generation, VisualAge uses this number as the resource ID of the first text string and increments the resource ID of each successive text string by 1.

For an example, look at the todolist.h file that VisualAge generates for the ToDo List application. The first #define statement in the todolist.h file appears as follows:

```
#define RC_ToDoList  10000
```

The number in this #define statement, `10000`, is the starting resource ID.

RELATED CONCEPTS

"The Resource File (.rc)" on page 117

"Resource Files Generated for Translation" on page 116

RELATED TASKS

"Chapter 14. Generating Code" on page 59

Chapter 28. VisualAge Component Model

This component model supports both new and pre-existing classes. You can apply the interface protocol to existing classes without making extensive code modifications. The part interface architecture specifies the general format of the programming interfaces, not the particular implementation behind the interface.

RELATED CONCEPTS

"Access to a Part's Properties"

"Access to a Part's Behavior" on page 120

"Notification of Changes to Parts" on page 121

Access to a Part's Properties

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

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

You can use an attribute 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 make public a complete attribute interface for a property. For example, a property might be read-only, in which case the part's attribute interface would not support the ability to set the property's value.

The attribute interface is represented as follows:

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

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

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

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

`notifyObservers` is the member function that signals the event; `anEventId` is the notification ID for the property change event; `*this` is the notifier object; `true` indicates that the value of the attribute has changed; `aValue` is the event data. (For more information about events, see “Notification of Changes to Parts” on page 121.)

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

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

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

While a property is often represented as a data member of a part, it need not be; the property could be a computed value. What is important is that whenever the value of the property changes, the change takes place using the set member function for the property. Changes made in any other way might not cause the event to be signaled.

RELATED CONCEPTS

“Access to a Part’s Behavior”

“Notification of Changes to Parts” on page 121

Access to a Part’s Behavior

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

The action interface is represented as follows:

```
aType aMemberFunction();
```

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

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

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

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

RELATED CONCEPTS

“Access to a Part’s Properties” on page 119

“Notification of Changes to Parts” on page 121

Notification of Changes to Parts

By signaling events, a part can notify other parts that a state or value in its interface has changed. Events can be signaled 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 signaled when the value of a part's property changes, such as when money is deposited into or withdrawn from a bank account.

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

```
static INotificationId const anEventId;
```

`anEventId` is the notification ID for the event.

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

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

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

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

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

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

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

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

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

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

RELATED CONCEPTS

"Access to a Part's Properties" on page 119

Chapter 29. Notification Framework

You use the IBM notification framework to implement event and attribute notification for visual and nonvisual parts. You can also use it in code you are writing by hand if the code is based on the IBM Open Class Library.

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

The notification framework contains the following entities:

- Notifiers, which broadcast changes in their internal state. Notifier instances support a protocol defined by the INotifier class.
- Observers, which register themselves as dependent upon the occurrence of a specific change in state. Observer instances support a protocol defined by the IObserver class.
- Events, which identify what state has changed in which notifier instance. Event instances support a protocol defined by the INotificationEvent class. An event instance can also contain the new value of the state that was changed (*event data*).
- Notification IDs, static strings in a notifier's interface that specify which events are supported by the notifier class. Each notification ID is in the form of the notifier-class name followed by the event name, such as IStaticText::backgroundColorId.
- Interests, which act as common currency among notifier, observer, and event instances. An instance of the IInterest class, each interest associates a notifier instance with a notification ID. This pairing uniquely identifies an event so that only interested observers respond to the event signaled.

RELATED CONCEPTS

"Notifiers and Observers"

"About Implementing Notification" on page 124

Notifiers and Observers

To register dependence upon a notifier, an observer calls the handleNotificationsFor function:

```
// *fiAccountNumEF is an IEntryField instance, a notifier
// The notification ID for the text attribute is inherited from ITextControl

handleNotificationsFor(IInterest(*fiAccountNumEF, ITextControl::textId));
```

To cancel registration, an observer calls the stopHandlingNotificationsFor function:

```
stopHandlingNotificationsFor(IInterest(*fiAccountNumEF, ITextControl::textId));
```

To signal an event, notifiers call the notifyObservers function. If event data is not passed with the notification, the call looks like this:

```
// Called from the setAccountNum function of OAContract

notifyObservers(INotificationEvent(IInterest(*this, OAContract::accountNumId)));
```

Otherwise, the call looks like this:

```
// Called from the enableIDvalid function of OAContractor
// The type of event data being passed is IString, represented by iContractorID

notifyObservers(INotificationEventFor<IString> (IInterest(*this, OAContractor::IdvalidId), iContrac
```

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.

RELATED CONCEPTS

"About Implementing Notification"

About Implementing Notification

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

- Enabling, disabling, and querying the ability to signal events through the following members:

  ```
  virtual INotifier& enableNotification(bool enable = true) = 0;
  virtual INotifier& disableNotification() = 0;
  virtual bool isEnabledForNotification() const = 0;
  ```

  In general, notifiers are created in a disabled state, so they must be explicitly enabled before they can signal events. This allows time for notifier and connection instances to initialize themselves and related instances.
- Managing the collection of observers through following protected members in INotifier:

  ```
  virtual INotifier& addObserver(IObserver& observer, const IInterest& interest);
  virtual INotifier& removeObserver(IObserver& observer) = 0;
  virtual INotifier& removeObserver(IObserver& observer, const IInterest& interest);
  virtual INotifier& removeAllObservers() = 0;
  virtual IObserverList& observerList(const IInterest* anInterest = 0) const = 0;
  ```
- Signaling an event through the notifyObservers member:

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

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

The IStandardNotifier class implements the notifier protocol and provides the base support for nonvisual parts. For visual parts, the notifier protocol is implemented in subclasses of IWindow. The notification under the IWindow classes occurs primarily using the existing handler classes.

To implement notification, your new nonvisual part can either inherit from IStandardNotifier or contain an IStandardNotifier instance. To use inheritance, specify IStandardNotifier (or a derived notifier class) as the base class when you create the part in VisualAge. If you do not specify a standard notifier as the base class, VisualAge assumes you want the part to notify using containment and generates the appropriate source code by default.

You can override the notification behavior that is provided by default through containment; this requires you write some code to be included in the generated class code. Follow these steps:

- In a .cpv file, derive a class from the implementation class generated for the part. (Make sure to change generation options in the Class Editor so that the .cpv file will be included in generated code.)
- In the derived implementation class, override the notifier( ) function, returning a reference to some type of notifier. It is through this function that you implement unique notification behavior for the part. Define a data member to hold the notifier instance returned from the notifier( ) function.
- To instantiate your derived implementation class instead of the base class at run time, add a line of code to the **beginning** of the constructor for the part in the Class Editor. The code must look something like this:

  ```
  fImpl = new MyImplementationSubclass();
  ```

RELATED CONCEPTS

“Notifiers and Observers” on page 123

RELATED TASKS

“Preparing for Code Generation” on page 59