

License Agreement

This is a legal agreement between you, the end user, and ProtoView Development Corporation. By opening this envelope you are agreeing to accept ownership of this product and to be bound by the terms of this agreement. It also contains proprietary source code which you are entitled to use only under the terms of this license agreement. If, after reading this agreement, you do not agree with its terms, return the software, manuals and diskettes within thirty days of purchase to the party from whom you received it for a refund. In order to receive a refund, the disk and manuals must be in resellable condition. IF YOU PURCHASE THE SOURCE THE SALE IS FINAL.

ProtoView Software License For InterAct

1. **Grant of License.** ProtoView Development Corp. grants a limited, non-exclusive license to use one copy of the application development tool called InterAct for the purpose of developing applications for the Windows environment. This copy of ProtoView must exist on a single terminal connected to a single computer (i.e., with a single CPU). You may not network InterAct or otherwise use it for development on more than one computer at the same time.

2. **Runtime Distribution License.** ProtoView Development Corp. grants you a royalty-free right to distribute copies of the runtime dynamic link libraries for use with applications you have developed using InterAct. InterAct runtime files are listed in exhibit A. These libraries may not be distributed for any other purpose than to accompany software that you have developed using InterAct. You may use InterAct in your specific purpose application programs, in which case ProtoView grants you permission under ProtoView's copyright to use, give away or sell such programs without additional licenses or fees, as long as all copies of these programs bear a valid copyright notice and provided that your program is not merely a set or subset of the or a compilation or development tool or library which includes all or a portion of the , or is otherwise a product that is generally competitive with or a substitute for InterAct. This permission is granted solely for the purpose set forth above, and you are not authorized to use InterAct in any other manner. You may not use InterAct to create a development tool for the AS/400.

3. **Copyright.** InterAct is owned by ProtoView Development Corporation and is protected by United States copyright laws and international treaty provisions. You may make one copy of the software for backup purposes. You may not, under any circumstances, copy the manual and other written materials that accompany this software.

4. **Other Restrictions.** You may not rent or lease InterAct software, but you may transfer the software and accompanying written materials on a permanent basis provided you retain no copies and the recipient agrees to the terms of this Agreement. You may not reverse engineer, decompile, or disassemble this software.

Limited Warranty

ProtoView Development Corp. warrants that the software herein will perform substantially in accordance with the accompanying written materials for a period of thirty days from the date of receipt. Any implied warranties on ProtoView are limited to ninety days. Some states do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

Customer Remedies. ProtoView's entire liability and your exclusive remedy shall be, at ProtoView's option, either (a) return of the price paid or (b) replacement of the software that does not meet ProtoView's Limited Warranty and which is returned to ProtoView with a copy of your receipt. This Limited Warranty is void if failure of the software has resulted from accident, abuse, or misapplication. Any replacement of the software will be warranted for the remainder of the original warranty period or 30 days, whichever is longer.

NO OTHER WARRANTIES. PROTOVIEW DEVELOPMENT DISCLAIMS ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE AND THE ACCOMPANYING WRITTEN MATERIALS. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE TO STATE. NO LIABILITY FOR CONSEQUENTIAL DAMAGES. IN NO EVENT SHALL PROTOVIEW OR ITS SUPPLIERS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THIS PROTOVIEW PRODUCT, EVEN IF PROTOVIEW HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

Exhibit A

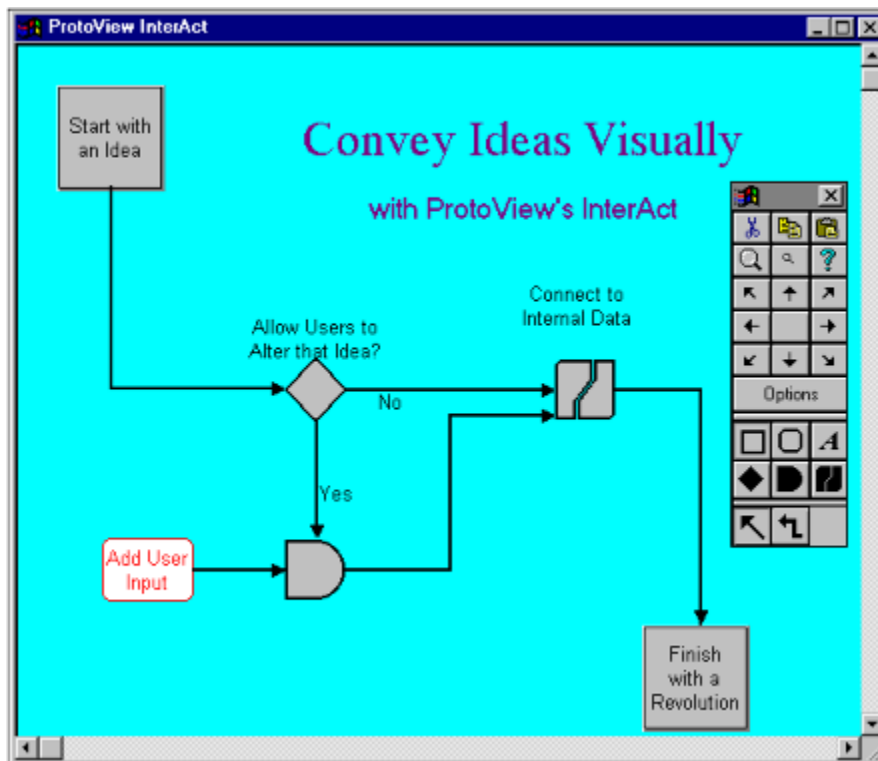
PVIDO.DLL - The 16-bit Control Module
PVIDO32.DLL - The 32-bit Control Module
PVIDO.OCX - The 16-bit OLE Control Module
PVIDO32.OCX - The 32-bit OLE Control Module
PGHT.DLL - The 16-bit Support Module
PGHT32.DLL - The 32-bit Support Module

What is InterAct?

Welcome to ProtoView InterAct, the Interactive Diagramming Object - or IDO for short. InterAct is an advanced custom control available as a DLL and ActiveX (OCX). It can be used in dialog editors, such as Microsoft's AppStudio, Borland's Resource Workshop, ProtoView's ViewPaint or Microsoft's Visual Basic. It can also be created at runtime in C, C++ or Basic programming languages.

InterAct is designed as a tool for displaying data to the user in a graphical format - as a diagram. In addition, it is interactive, meaning that the user can use the mouse and keyboard to modify the diagram. Objects (called entities) can be moved, resized, or have their appearance changed, and lines between objects (called relations) can be added or deleted. The application in which you embed InterAct is alerted to every user action in the diagram, allowing the application to respond to these events. In addition, a full API is exported from InterAct which allows an application to customize it to suit the application's needs.

As additional support, InterAct supplies a complement of built-in dialogs and menus it can display to allow the user to manipulate a diagram. InterAct also provides support for saving or reading diagrams to disk, cutting and pasting information to the Windows clipboard, and printing diagrams. With this support it is possible to plug InterAct into an existing application and with little effort, have a full-fledged diagramming object. The programmer can then concentrate on the details of integrating the diagram with internal data structures.



The Interactive Diagramming Object

Installing InterAct

Before installing InterAct, be sure to read the license agreement included in the product package or displayed during the installation process.

Perform the following steps to install InterAct:

Floppy disk Install

1. Make sure Windows, Windows NT, or Windows 95 is running.
2. Insert the disk labeled “InterAct Disk 1” into drive A or B.
3. For WFWG 3.1 or higher, or Windows NT, select **File | Run** from the Program Manager’s menu. In the **Command Line** edit box of the *Run* dialog, enter “A:\SETUP” or “B:\SETUP” depending on where you inserted the setup disk. Click **OK**.
4. For Windows95, select **Start | Run** from the task window. In the **Open** edit box of the *Run* dialog, enter “A:\SETUP” or “B:\SETUP” depending on where you inserted the setup disk. Click **OK**.
5. Follow the instructions given in the installation program.

CD-ROM Install

Make sure Windows, Windows NT, or Windows 95 is running.

Insert the CD labeled “ProtoView” into drive D or whichever drive is your CD-ROM drive.

For WFWG 3.1 or higher, or Windows NT, select **File | Run** from the Program Manager’s menu. In the **Command Line** edit box of the *Run* dialog, enter “D:\SETUP” or use another drive letter depending on where you inserted the CD-ROM. Click **OK**.

For Windows95, select **Start | Run** from the task window. In the **Open** edit box of the *Run* dialog, enter “D:\SETUP” or use another drive letter depending on where you inserted the CD-ROM. Click **OK**.

Follow the instructions given in the installation program.

For instructions on how to install InterAct into a programming environment such as Microsoft’s Developer Studio or Visual Basic, or ProtoView’s ViewPaint, see *Section III: InterAct Tutorials*. These tutorials will also show some of the ways in which InterAct can be programmed.

Product Support

If you have questions about InterAct, you may contact our Technical Support staff. ProtoView now offers different types of technical support services for you to choose from including 30 days of complimentary telephone support, standard support, and available extended support packages.

When contacting our Technical Support staff, please provide the following information:

- Your product registration number found on the label of each disk that you received with InterAct.
- The version of InterAct you are using and the file dates of any updates you have installed.
- The exact wording of any messages that appeared on your screen.
- The error code returned from the property or method you are having a problem with.
- A brief and concise explanation of the question or problem that you are experiencing.
- How you tried to solve the problem.

Complimentary Technical Support

If you are a registered user, free technical support is available by telephone for 30 days from the date of your first telephone call. (Standard Technical Support is always available at no charge.)

To obtain technical support by telephone, call (609) 655-5000. Technical Support hours are Monday through Friday, 9:00 A.M. to 5:00 P.M. Eastern Standard Time (excluding holidays).

Standard Technical Support

Available at no charge, Standard Technical Support provides all registered users with technical support through the following services:

Bulletin Board Services(BBS)

ProtoView Development maintains a 24 hour Bulletin Board Service. This bulletin board is available for technical questions and problem reports. It also contains updated information about InterAct along with a library of examples and source code that can be downloaded for your own use.

This service is free of charge to all users and can be reached at (609) 655-4411. The communication parameters are "9600/2400/1200/300,8,N,1".

On-line Support Forum

You can join ProtoView on CompuServe where you can ask questions, download files and communicate with other ProtoView users. To access the ProtoView forum on CompuServe, type "GO PROTOVIEW" at the CompuServe command prompt.

Internet Support

You can ask questions and download files with ProtoView users on the Internet. To access the ProtoView site use the following addresses:

<http://www.protoview.com>

[ftp protoview.com](ftp://protoview.com)

tech@protoview.com

ProtoView Technical Fax

You may also ask about a problem or question by sending a facsimile message. The number for faxing is

(609) 655-5353. Be sure to include your name, fax number, telephone number and as much information about the problem as possible. Our technical support personnel will review your questions and fax back a reply.

Extended Support Packages

Several support packages are available at an additional charge. ProtoView offers priority, premier, and pay-as-you-go support options. Call ProtoView sales department at 1-800-231-8588 for additional information about the different support packages available.

Product Training and Consultation Services

Within the United States, ProtoView Development offers the following services for training and consultation.

ProtoView Professional Services

ProtoView Professional Services offers education and training in many areas of Windows programming. Both beginning and advanced courses are available in 3 or 5 day units

ProtoView Professional Services has highly skilled and experienced personnel available for the development of Windows applications and systems. These services can be provided on-site, off-site, full or part time. For more information, PPS can be reached at 770-390-6226 or info@protoview.com.

InterAct File Listing

InterAct will install different components based on the install you purchased and options you select during the install process. Certain components will not be installed if you do not select those options during the installation.

An InterAct DLL setup may install the following components:

1. The InterAct 16-bit dynamic link library (PVIDO.DLL)
2. The InterAct 32-bit dynamic link library (PVIDO32.DLL)
3. Two support DLLs (PGHT.DLL, PGHT32.DLL)
4. The InterAct header file (PVIDO.H)
5. The InterAct DLL import libraries (PVIDO.LIB, PVIDO32.LIB, PVIDO32B.LIB, PVIDO32S.LIB)
6. The InterAct MFC classes (IDOMFC.H)
7. The InterAct OWL classes (IDOOWL.H)
8. The InterAct Diagramming Assistant, a program that allows you to visually design diagrams (INTERACT.EXE)
9. Sample Files
10. InterAct Help Files

An InterAct ActiveX setup may install the following components:

1. The InterAct 16-bit ActiveX library (PVIDO.OCX)
2. The InterAct 32-bit ActiveX library (PVIDO32.OCX)
3. Two support DLLs (PGHT.DLL, PGHT32.DLL)
4. The InterAct Diagramming Assistant, a program that allows you to visually design diagrams (INTERACT.EXE)
5. Sample Files
6. InterAct Help Files

Introduction

The sections in this chapter will give a comprehensive demonstration of InterAct. This tutorial will not cover any of the programming aspects of InterAct - it will simply demonstrate with step-by-step instructions how to manipulate InterAct using the mouse and keyboard. For this tutorial we will use the sample program *InterAct Diagramming Assistant* installed with InterAct.

Creating Diagrams

To run the InterAct Diagramming Assistant:

When you installed InterAct, an icon to launch the *InterAct Diagramming Assistant* was installed. Double-click this icon to begin the *InterAct Diagramming Assistant*. If you do not have this icon, browse to the directory where you installed InterAct and run the program INTERACT.EXE.

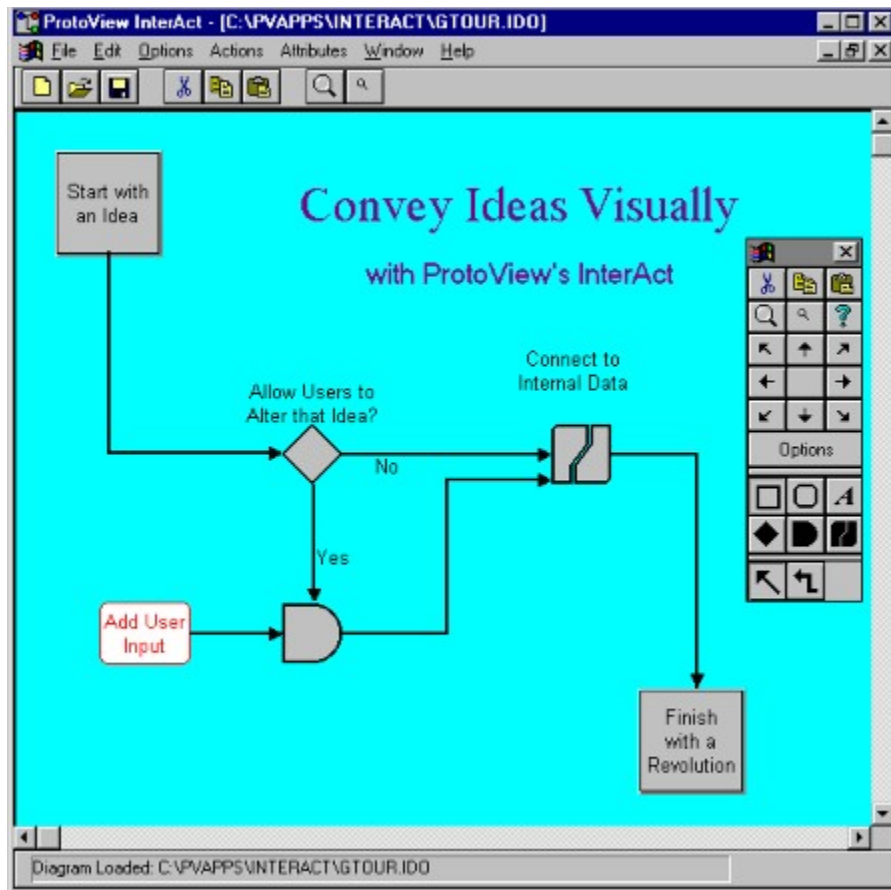


The InterAct Diagramming Assistant

After the *InterAct Diagramming Assistant* splash screen disappears, the *InterAct Diagramming Assistant* will open a new window automatically. An InterAct control will completely occupy the window. InterAct will have a grey background and white grid lines evenly spaced. Also notice that a tools palette is floating over the upper-right corner of InterAct. The tools palette is maintained completely by InterAct and provides an advanced interface for modifying a diagram. We will begin using InterAct by loading a diagram already created.

To load a diagram:

Select the Tools Palette button **Options | Load Diagram**. With the Common Dialog select the file GTOUR.IDO. InterAct will now load the diagram saved in the GTOUR.IDO file.



The diagram 'GTour.IDO.'

InterAct will restore all the entities and relations in the diagram along with the colors, styles, and graphics of each item. An IDO file also stores all the attributes of a diagram, including background and gridline color, grid properties, and zoom mode. One of the uses for InterAct is to create diagrams beforehand and load those diagrams when needed. These diagrams can be modifiable or read-only.

To mark this diagram read-only:

Select the menu item **Options | Edit Mode**. The checkmark will be removed from this menu option and InterAct is made read-only. Notice that the tools palette disappears automatically when InterAct is marked as read-only. Since the InterAct is read-only and the tools palette's main purpose is to modify the diagram, it is unnecessary to display the tools palette.

Now if you click on any objects or attempt to drag objects to a new location your efforts will be ignored. However, notice that the selection rectangle changes to the last relation, entity or entities clicked. The container application is also notified of these clicks and selection changes so it can provide a response to these events even though the diagram is read-only.

We will restore InterAct to be editable and make some changes to the diagram.

To make InterAct editable:

Select the menu item **Options | Edit Mode**. The checkmark will be added to this menu option and InterAct made editable. Notice that the tools palette reappears automatically.

For more information about creating and altering diagrams, see *Chapter 2: The InterAct Environment*.

Modifying Objects in a Diagram

InterAct provides many built-in dialogs for modifying diagrams. Many of these dialogs take the form of property pages - that is, a collection of dialogs which allow customization of common features. It is entirely up to the programmer to decide which of these dialogs to use and which to ignore.

To modify InterAct's attributes:

1. Right click on InterAct's background, not on an entity or relation. InterAct's floating menu will appear.
2. Select the menu item **Diagram Colors** from the floating menu. This takes us to the **Colors** property page. This property page allows you to select any of the predefined colors for the InterAct control background or grid lines.
3. Select the color "Light Red" for the **Background Color**.
4. Click the **Apply** button to have this change accepted immediately. InterAct is immediately updated with this new color.

Note: Once you apply changes you cannot cancel these changes.

5. We will now change the colors of InterAct to something less bright. Change the **Background Color** back to "Light Cyan." Change **Grid Line Color** to "Red."
6. Now display the **Grid** property page. The main attributes of the **Grid** property page is the spacing between grid lines.
7. Set the **Horizontal Spacing** to "25."

Note: To change the value of these fields you can either enter the new value, use the scrollbar on the right edge of the field, or use the up/down arrow keys when focus is in the field.

8. Set the **Vertical Spacing** to "35."
9. Click **OK** to have these changes applied to InterAct and close the property window.

Our diagram now has red grid lines on a light blue background. Grid lines can be an important aid in helping users design their diagram. InterAct can automatically "snap" entities in the diagram to the nearest grid line to make it easier to align entities, even if grid lines are not visible. We will hide grid lines in our Guided Tour diagram.

To hide grid lines:

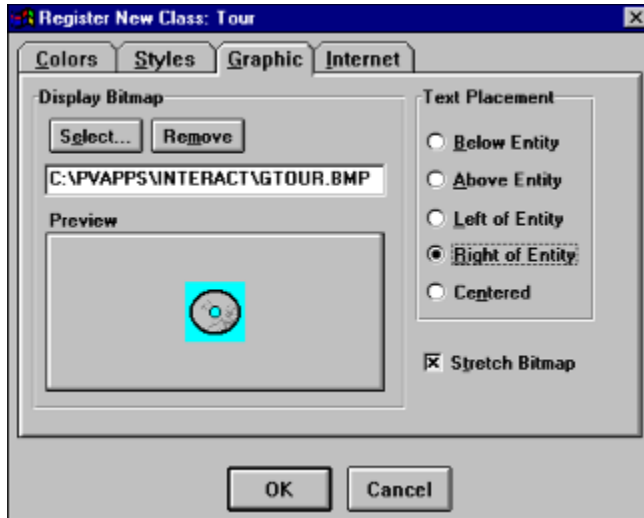
Select the menu item **Options | Grid Lines**. The checkmark will be removed from this menu option and the grid lines will disappear.

We will now modify some of the attributes on an entity in the diagram.

To modify an entity's attributes:

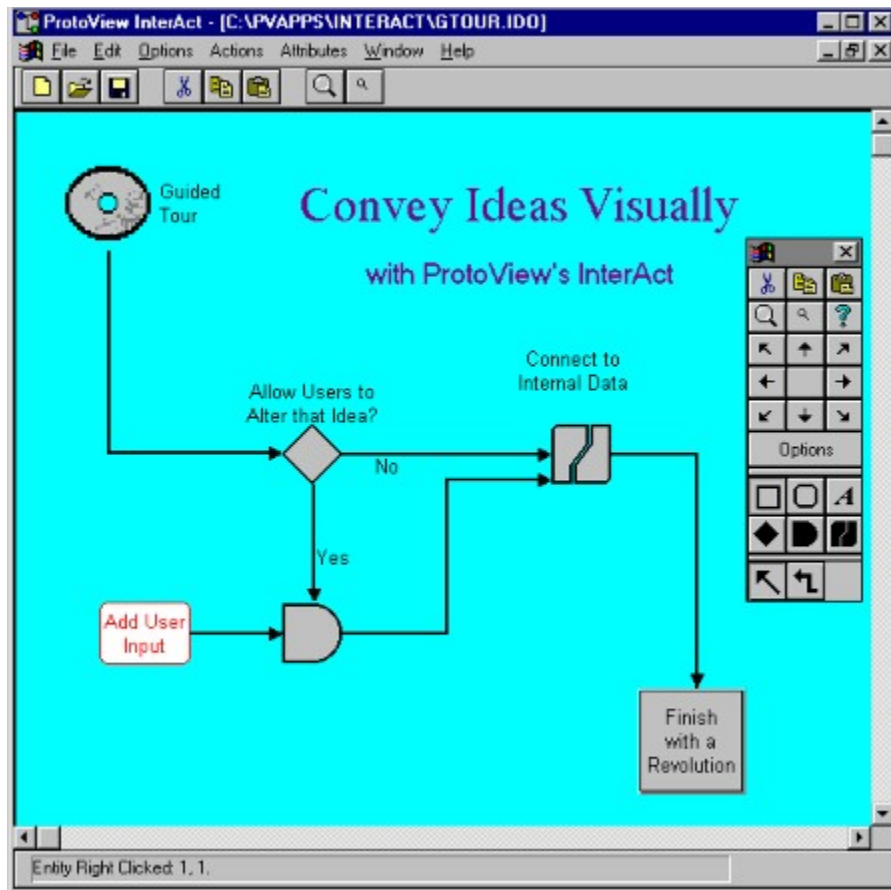
1. Right click in the interior of the entity with the text "Start with an Idea." The entity's floating menu will appear.
2. Select the menu item **Graphics**. This takes you directly to the **Graphic** property page.
3. Click the button marked **Select** to select a bitmap file. A common dialog *Select BMP File* will appear.

4. Navigate to the InterAct directory and select the bitmap GTOUR.BMP. Click **OK** to accept this file to return to the **Graphic** property page. Notice that the property page displays a preview of this bitmap.
5. Check on the option **Stretch Bitmap**.
6. In the **Text Placement** group, select the radio button **Right of Entity**. Any text associated with the entity will be placed to the right of the graphic. We will now add that text.



The Graphics Property page.

7. Display the **Text** property page.
8. Replace the existing text with the text "Guided Tour" in the field.
9. Click the radio button **Left**. This will left justify the text.
10. Display the **Colors** property page.
11. Select "Light Cyan" as the **Background Color**.
12. Display the **Styles** property page.
13. Turn off **Border**, and set **3D Effect** to "None."
14. Click the **Apply** button. These changes will be applied to the entity.
15. Click the **Close** button.



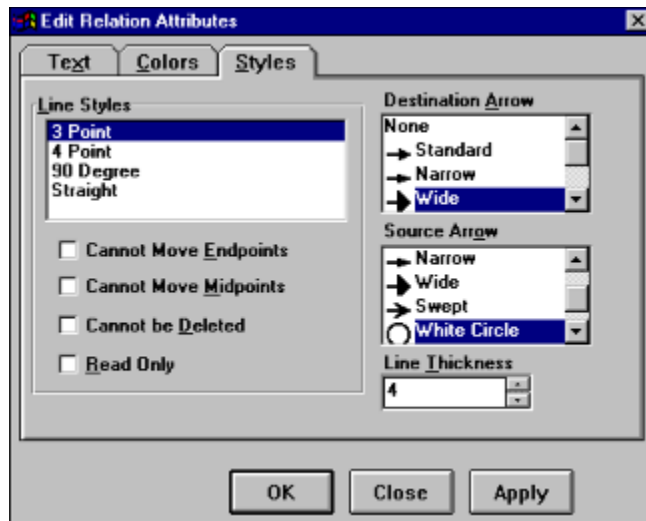
An entity after we have applied some changes to it.

To modify a relation's attributes:

1. Right click on a portion of the line representing the relation originating directly below the entity "Guided Tour." The relation's floating menu will appear.

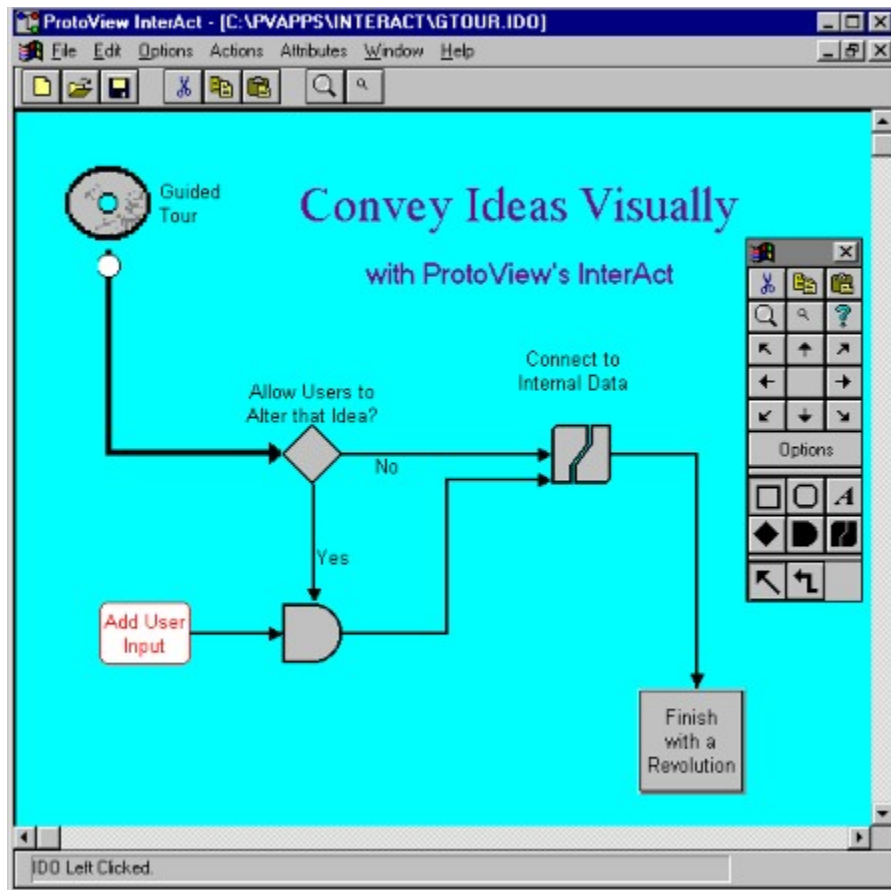
Note: Make sure you click on the line itself and not the text associated with the line.

2. Select the menu item **Styles**. This takes you directly to the **Styles** property page.



*The **Styles** page of the relation property page.*

3. Change **Line Thickness** to “4.”
4. Select “White Circle” as the **Source Arrow**.
5. Select “Wide” as the **Destination Arrow**.
6. Click the **Apply** button. These changes will be applied to the relation.
7. Click the **Close** button.



The relation after we have set some properties on it.

For more information about the available properties, see *Chapter 3: Property Pages*.

Working with the Tools Palette

In addition to being able to restore entire diagrams at once, another use is the ability to save and restore only a specific palette. A palette contains a collection of entity and relation classes, each describing how an object will look or behave. A palette can also contain other useful information, such as rules which govern how entities can be joined by relations, initial colors of InterAct, and spacing of gridlines. To see this, we will load a palette into InterAct. First, close this window by selecting **File | Close**. Next, open a new InterAct window by selecting **File | New**. A fresh InterAct window will be displayed with a tools palette containing a “default” selection of classes.

To load a new tools palette:

Remove the current palette by selecting **File | Reset Palette**. All the classes are removed from the palette.

Select the menu **File | Load Palette**.

Select the file GTOUR.PLT.

InterAct will now load the palette file. Notice the new assortment of buttons on the palette and the new color and grid line spacing of InterAct.

To add an entity from the tools palette:

1. Click on one of the entity symbols on the tools palette.
2. Move the mouse over the InterAct. The mouse cursor will change shape to a hand with a pointer.
3. Click and hold down the left mouse button.
4. Holding the left mouse button down, drag the mouse. A rectangle will be drawn as you drag the mouse to indicate the area which will be occupied by the new entity.
5. Release the left mouse button. A new entity will be added to the diagram.
6. Repeat this procedure to add any number of entities.

To add a relation from the tools palette:

1. Click on one of the relation symbols on the tools palette.
2. Move the mouse over the InterAct. The mouse cursor will change shape to a hand with a pointer.
3. Click and hold down the left mouse button on an entity.
4. Holding the left mouse button down, drag the mouse. A line will be drawn as you drag the mouse to indicate the area which will be occupied by the new relation. When you move the mouse over a valid target entity the mouse cursor will become inverted.
5. Release the left mouse button when over a valid target entity. A new relation will be added to the diagram between the source and target entities.
6. Repeat this procedure to add any number of relations.

For more information about the tools palette, see *Chapter 4: The Tools Palette*.

Exploring Classes and Rules

As the previous section showed, loading and using different palettes is an important feature of InterAct. Palettes are just a visual representation of internal classes which InterAct maintains. Each class is simply a series of conditions which determine how an entity will behave and look. Being able to create and maintain these classes is an important part of using InterAct.

Altering an existing class:

Click on the **Options** button of the Tools Palette. Select the menu item **Manage Classes**. The *Manage Classes* dialog will be displayed. This dialog lists all the classes available in InterAct.



The Manage Classes dialog.

In the **Entity Classes** groupbox, select the class “Generic Rectangle” and press the **Redefine Class** button. A property page similar to the one displayed when you edit properties on an entity is displayed.

Display the **Colors** property page.

Set the **Background Color** to “Light Blue.”

Display the **Styles** property page.

Set **3D Effect** to “Shadow.”

Click **OK**. These changes will now be accepted. Notice that all the entities of the class “Generic Rectangle” now have the light blue background and shadow as a 3D effect.

Creating a new class:

In the **Entity Classes** groupbox, click the **Define New Class** button. The *Register New Class* dialog is displayed.

Enter the **Class Name** “Tour.”

For the **Class Icon**, press the **Select** button. This allows you to select a bitmap from disk to act as an icon.

Note: Bitmaps displayed on the tools palette should be 16 x 16 pixels in size to appear properly.

Select the bitmap “IGTOUR.BMP” from InterAct directory.



Defining a new class.

Click **Apply**. A property page similar to the one displayed when you redefined an entity class is displayed with the caption “Register New Class: Tour.”

On the **Colors** property page, set the **Background Color** to be “Green.”

On the **Styles** property page, set the **Shape** to be “Circle.”

Click **Done** to **Close** the *Manage Classes* dialog and have these changes reflected on the tools palette. You can now select the newly created class from the tools palette and place objects of that class in the diagram.

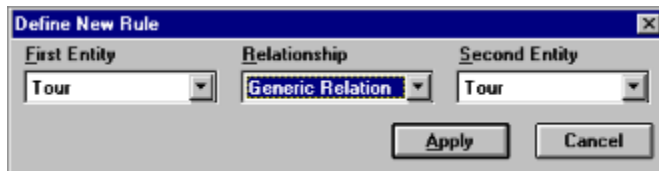
Enabling Rules:

Select the menu item **Options | Enforce Rules**. Rules are now being enforced in the diagram. Select **Options | Manage Classes** from the Tools Palette. Notice that a button **Rules** is now visible in the *Manage Classes* dialog. Click this button to go to the *Manage Rules* dialog.

Select all the items in the **Diagramming Rules** list and click the **Delete Rule** button. When asked to confirm the delete select **Yes**.

Adding a Rule:

1. Click the **Add Rule** button.
2. In the **First Entity** list select “Tour,” the class we defined earlier.
3. In the **Relationship** list select “Generic Relation.”
4. In the **Second Entity** list select “Generic Rectangle.”



Adding a rule.

5. Click **Apply**. Click **Done** to leave the *Manage Rules* dialog. Click **Close** to close the *Manage Classes* dialog.

Enforcing Rules:

InterAct will now enforce any and all rules within a diagram. Select the menu **File | Reset**. All the entities and relations will be removed from the diagram. Now add three entities, two of type “Tour” and one of type “Generic Rectangle.”

Now if you try to create a “Generic Relation” between the two “Tour” entities InterAct will not let you. It will however allow a “Generic Relation” to connect a “Tour” entity to a “Generic Rectangle.” This support is maintained completely by InterAct.

For more information about classes and rules, see *Chapter 5: Classes and Rules*.

Summary

This concludes the guided tour of InterAct with the *InterAct Diagramming Assistant*. The tutorial is designed to show some of the basic capabilities of InterAct which a programmer of InterAct can make use of. For tutorials which contain programming samples of InterAct for C, C++ and Visual Basic users, see Section II of this manual.

Overview

InterAct is a diagramming object which contains two basic types of objects - entities and relations.

Entities

Entities are a rectangular region in a diagram which can be manipulated. Entities can have different shapes and colors. Optionally, an entity can contain text and/or a graphic. Entities can be moved and resized throughout a diagram.

Relations

Relations are lines which connect entities. Lines can have two, three or four points which can be positioned, and a color. Optionally, symbols can be added to either or both endpoints, and text can be associated with a line.

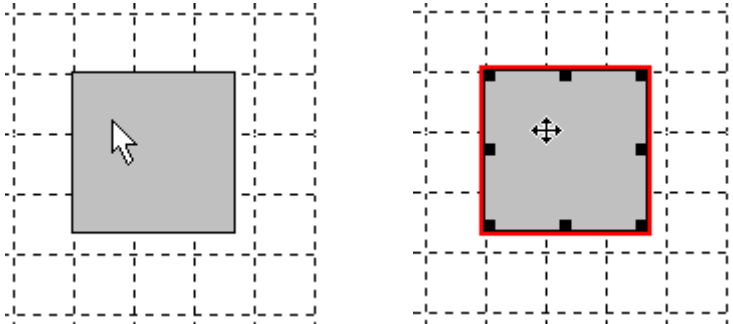
Diagrams

Using only these two basic objects, you will be able to create a wide array of detailed diagrams to suite almost any need. Learning how to manipulate these objects using the mouse and keyboard will allow you to create diagrams faster and easier.

How to Modify the InterAct Environment

To select an entity:

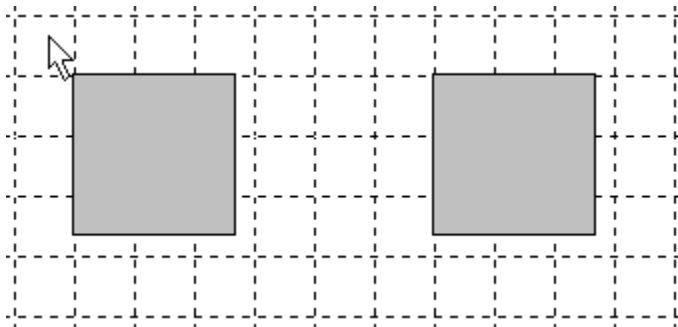
Left-click the mouse anywhere within the rectangular region of the entity. A selection rectangle with handles in the corners and on the sides of the entity will appear.



An entity before and after it is selected.

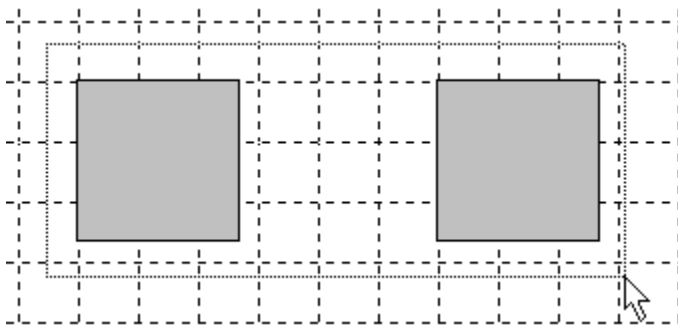
To select more than one entity:

Left-click the mouse anywhere within the InterAct's client area, but not within a region occupied by an entity.



Left click to start drawing a selection rectangle.

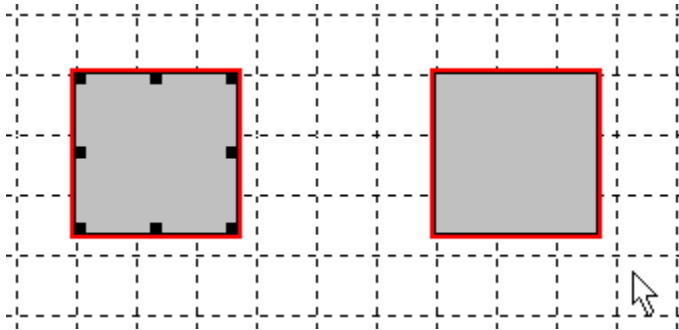
Keeping the left mouse button depressed, drag the mouse. A rectangular “rubber-band” will be drawn on the screen as you drag the mouse.



Encircle entities with the selection rectangle.

Release the left mouse button. Any entities contained within the rectangular “rubber-band” will be

selected. A rectangle will be drawn around all the entities in the group. Notice that only one of the selected entities has “track handles” surrounding it. This entity is the anchor of the group, and is the main entity in the selected group.



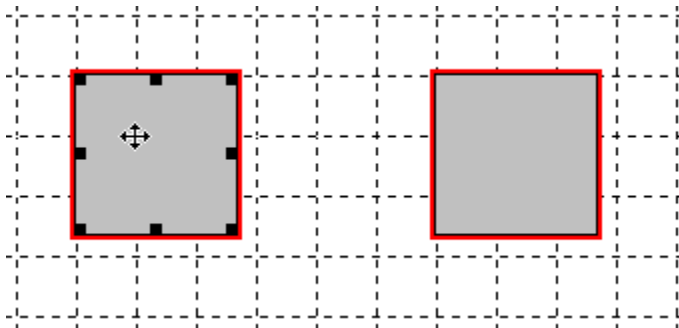
Release the mouse button. The entities within the selection rectangle are now selected.

To add or remove entities to a selection group:

Holding the **Shift** key down, left-click the mouse on an entity. The entity will be added to the selected group, and have a rectangle drawn around it. If the entity is already contained in the group, it will be removed from the group and the rectangle around it will be removed.

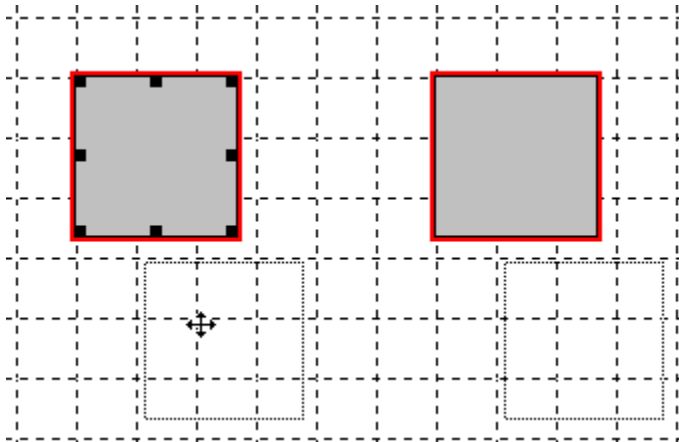
To move an entity:

Left-click the mouse within the area of a selected entity.



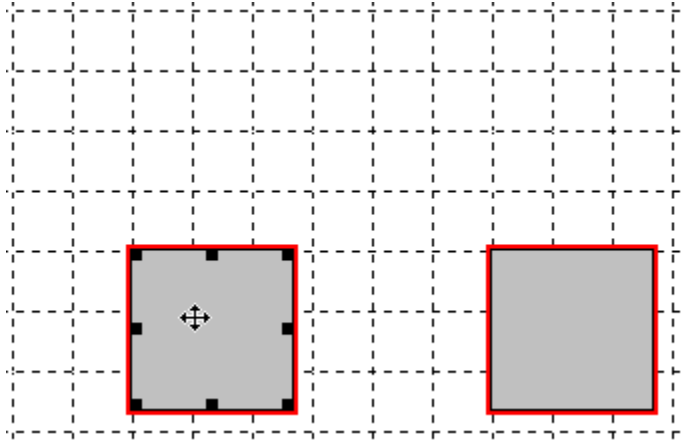
Notice the different shape of the mouse cursor when moved over a selected entity.

Keeping the left mouse depressed, drag the mouse. A shadow image of the entity or entities being moved will be displayed.



Shadows of the selected objects will be drawn as you drag them to a new location.

When you have positioned the entities in their new position, release the mouse button. The selected entity or entities will be moved.

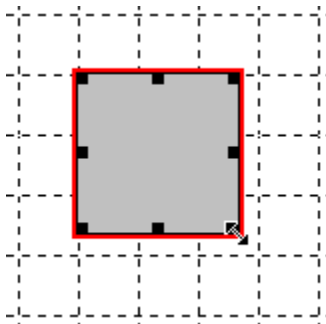


The selected objects are now moved to their new location.

To resize an entity:

Select an entity. A selection rectangle with handles will be drawn around the entity.

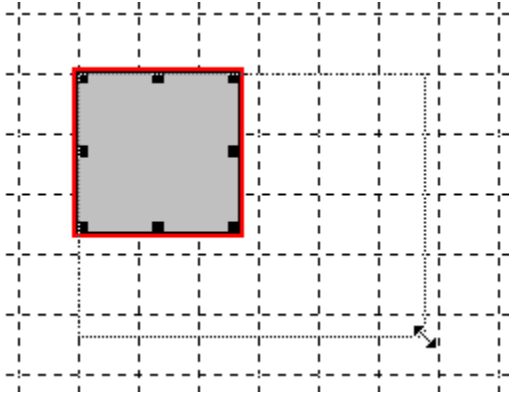
Move the mouse over a handle positioned at a corner or side of the entity (these are called resizing handles). The mouse cursor will change shape when it is moved over the resizing handles.



Notice the cursor shape when moved over an entity corner.

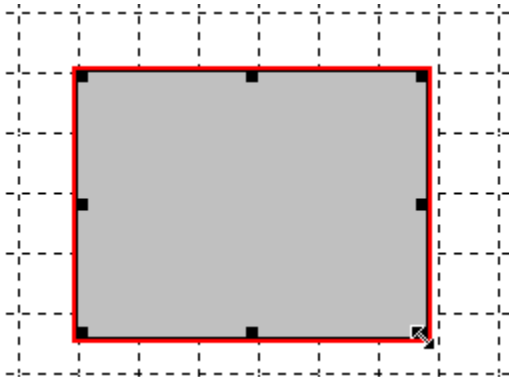
Press the left mouse button.

Keeping the left mouse button depressed, drag the mouse to a new location. As you drag the mouse, a rectangle depicting the entity's new size will be drawn.



A shadow image of the entity will be displayed when dragging to a new size.

Release the mouse button when you have reached the desired size. The entity will be redrawn at the new size.

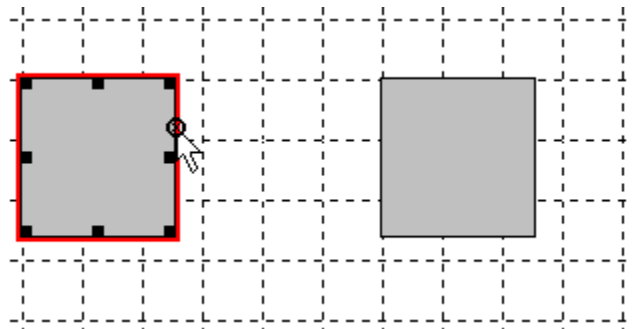


The entity is now resized.

To connect a relation between two entities:

Select an entity. A selection rectangle with handles will be drawn around the entity.

Move the mouse over the side of the entity, but not over a resizing handle. The mouse cursor will change shape when it is moved over these handles.

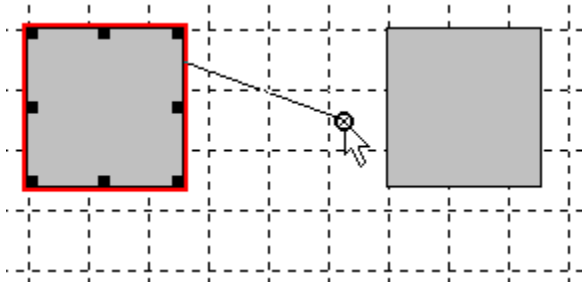


Notice the shape of the cursor when moved over the entity side.

Press the left mouse button. This entity is the *source entity*.

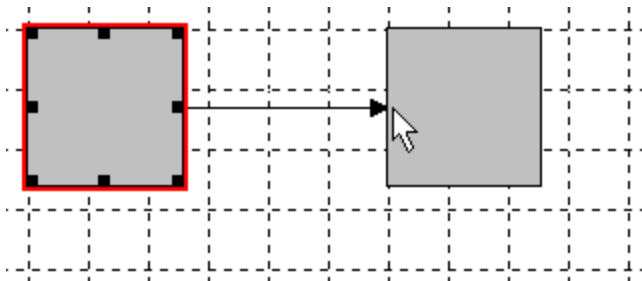
Keeping the left mouse button depressed, drag the mouse to a new location on top of another entity. This entity is the *destination entity*. As you drag the mouse, a line depicting the relation being created will be

drawn. As the mouse is dragged over entities in the diagram, it will change color to let you know you are over a valid target entity.



A shadow of the relation will be drawn as you select a target entity.

Release the mouse button when you have reached the desired destination entity to connect the relation to. A relation will be drawn between the source and destination entities.



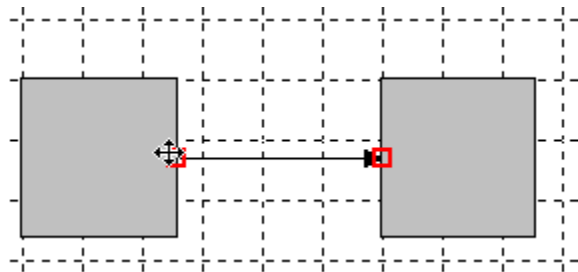
The relation is now created between the source and target entities.

To edit the text on an entity:

Double-click the left mouse button on an entity. A window with the entity's current text will appear. Enter the new text. Hit the **Enter** key to accept this change. Press the **Escape** key to abort the change.

To select a relation:

Left-click the mouse anywhere along the line of the relation. Selection rectangles will appear at the end- and mid- points of the line. You can click and drag these end-or mid- points to reposition the line.



A relation with selection rectangles at its ends.

To change the shape of a relation:

Select a relation. Red selection markers will be drawn at the end points and midpoints of the line to represent the drag-handles of the line. Some relations will not have midpoints. Move the mouse over a drag-handle and press the left mouse button. Holding the left mouse button down, drag the handle to a new location. Releasing the left mouse button will move the point to the new location. Pressing **Escape** while keeping the left mouse button down will abort the operation.

To edit the text on a relation:

Double-click the left mouse button on a relation. A window with the relation's current text will appear. Enter the new text. Hit the **Enter** key to accept this change. Press the **Escape** key to abort the change.

To add an entity to a diagram:

Click on one of the entity symbols on the tools palette.

Move the mouse over InterAct. The mouse cursor will change shape to a hand with a pointer.

Click and hold down the left mouse button.

Holding the left mouse button down, drag the mouse. A rectangle will be drawn as you drag the mouse to indicate the area which will be occupied by the new entity.

Release the left mouse button. A new entity will be added to the diagram.

Repeat this procedure to add any number of entities.

To add a relation to a diagram:

Click on one of the relation symbols on the tools palette.

Move the mouse over an entity in InterAct. The mouse cursor will change shape to a hand with a pointer.

Click and hold down the left mouse button.

Holding the left mouse button down, drag the mouse. A line will be drawn as you drag the mouse to indicate the area which will be occupied by the new relation.

Drag the mouse over a target entity.

Release the left mouse button. A new relation will be added to the diagram connecting the first entity clicked with the target entity.

Repeat this procedure to add any number of relation.

To delete a relation:

Select a relation. Press the **Delete** key. The selected relation will be deleted.

To delete an entity or group of entities:

Select one or more entities. Press the **Delete** key. The entities will be deleted. Any relations pointing to or from the deleted entities will also be removed.

Summary

This chapter has described the keyboard and mouse commands available to manipulate with InterAct. With the mouse and keyboard you can completely customize an InterAct diagram. This chapter has only briefly talked about the tools palette. The next chapter will discuss this component in much more detail.

Introduction

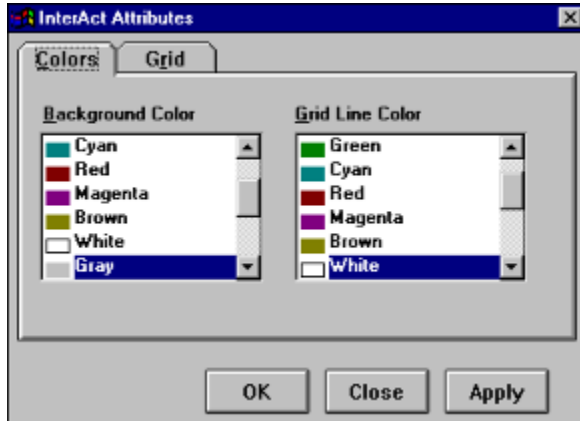
InterAct allows access to built-in property pages which provide access to styles and features of InterAct as well as individual entities and relations.

InterAct Property Pages

InterAct has two property pages - Colors and Grid.

InterAct Colors

Allows selection of colors for InterAct.



InterAct Colors property page.

Background Color

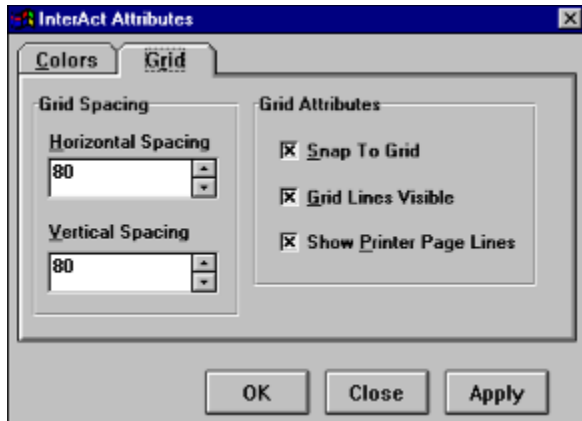
A list of available colors is displayed to choose from. This color is used when drawing InterAct's background.

Grid Line Color

A list of available colors is displayed to choose from. This color is used when drawing InterAct's grid lines, if grid lines are being displayed.

InterAct Grid

The InterAct Grid property page allows you to set properties which define the gridlines for the diagram. Spacing, whether snap-to-grid is enabled, and whether gridlines are visible are all defined on the property page.



InterAct Grid property page.

Horizontal Spacing

Allows you to specify the horizontal spacing for grid lines, if grid lines are being displayed. Even if grid lines are not being displayed, the width is used by Snap-To-Grid to snap an entity to the nearest “invisible” grid line. When the cursor is in this field, the value can be incremented/decremented by pressing the **Up Arrow** and **Down Arrow** keys, respectively.

Vertical Spacing

Allows you to specify the vertical spacing for grid lines, if grid lines are being displayed. Even if grid lines are not being displayed, the height is used by Snap-To-Grid to snap an entity to the nearest “invisible” grid line. When the cursor is in this field, the value can be incremented/decremented by pressing the **Up Arrow** and **Down Arrow** keys, respectively.

Snap to Grid

Toggles if InterAct will snap entities’ upper-left corner to the nearest grid line intersection. A useful feature for easily arranging entities. Does not depend on Grid Lines being visible.

Grid Lines Visible

Toggles whether InterAct will display grid lines.

Show Printer Page Lines

Toggles whether InterAct will show page lines in a large diagram. Printer page lines are useful if printing a diagram is necessary and the user needs to see the approximate pagination of the print. InterAct must be able to locate the default printer of the workstation for these lines to be drawn correctly.

Entity Property Pages

Entities have four property pages - Text, Colors, Styles and Graphic.

Entity Text

The Text property page allows you to set properties which affect the display of an entity. Of course the most obvious is the actual text displayed by an entity. You can also select the font and orientation of the text. Of equal importance, you can see the ID and Name assigned to the entity. Usually, you will provide the ID or Name when you create the entity, but if you do not supply one, InterAct will create default values. You will also see the class type of the entity.



The Entity Text property page.

Class Type

This is the class of the entity. The class determines what the entity will look like and how it will behave when it is initially created. If the attributes of a class are modified, the entities of that class will automatically assume the new attributes. If rules are being used InterAct, class name will also be important, as rules refer to how classes can interact.

User ID

A numeric value which uniquely identifies the entity.

User Name

A string which uniquely identifies the entity.

Font

Click this button to select a new font for the entity. Different point sizes and styles can be selected for this font. Each entity can have a different font, if selected.

Left / Center / Right

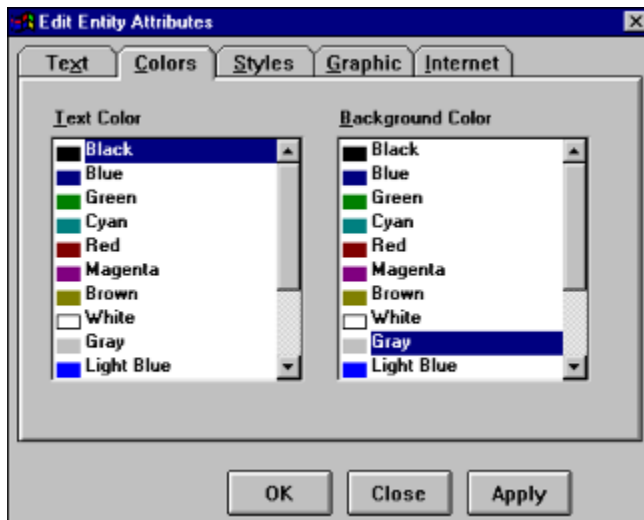
Select the orientation of the text when aligned in the entity.

Text

The actual text which will be displayed by the entity. The text will be displayed in the selected font and orientation. Text will always be clipped within the region defined by the entity. If the entity is selected to automatically size to accommodate all the text, all the text will be visible. See the Entity Styles property page for details about the **AutoResize** option. If the entity is displaying a graphic the text will not be displayed inside the entity. It will be displayed above, below, to the left or right of the entity. See the **Graphic** property page for an explanation of these options.

Entity Colors

Allows selection of colors for the entity.



The Entity Colors Property Page.

Text Color

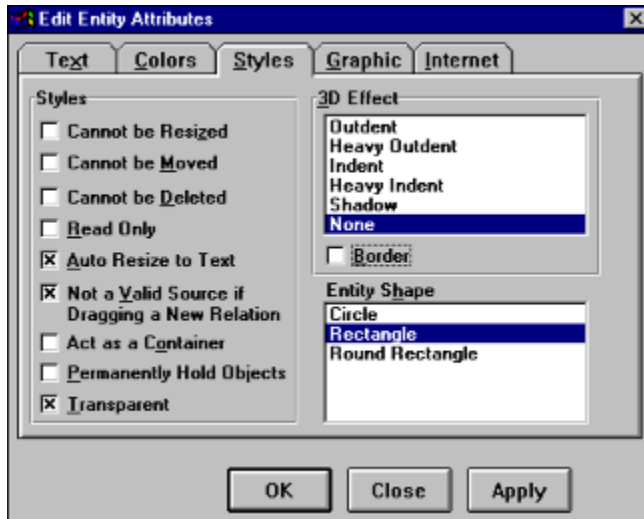
A list of available colors is displayed to choose from to use when drawing the entity's text. If the entity is also displaying a border, the border will be drawn in this color as well. See the Entity Styles property page for details about the **Border** option.

Background Color

A list of available colors is displayed to choose from to use when drawing the entity's interior. If the entity is selected to be transparent, this color selection will have no effect. See the Entity Styles property page for details about the **Transparent** option.

Entity Styles

A variety of styles can be applied to each entity that have major effects on the look and behavior of the entity. Any number of styles can be applied, and their effects are cumulative. As an example, if you want to create a "Simple Text" entity, you would select the styles **Transparent**, **AutoResize to Text**, and **Not a Valid Relation Source**.



The Entity Styles property page.

Cannot be Resized

The entity cannot be resized using the mouse. Attempts to use the drag handles to resize the entity will be ignored.

Note: You can still resize an entity by setting properties on the entity at runtime. This style only prevents the use of the mouse to resize the entity.

Cannot be Moved

The entity cannot be moved from its current location using the mouse.

Note: You can still reposition an entity by setting properties on the entity at runtime. This style only prevents the use of the mouse to reposition the entity.

Cannot be Deleted

The entity cannot be deleted if the entity is currently selected and the **Delete** key is pressed.

Note: You can destroy an entity at runtime by invoking the InterAct method **DeleteEntity** method. This style only prevents the use of the **Delete** key to delete the entity.

Read Only

Normally, double-clicking an entity will allow you to edit the entity's text. This style suppresses this behavior, creating a "read-only" entity.

Note: You can still change the text on an entity by setting properties on the entity at runtime. This style only prevents the use of the mouse to edit the entity's text.

Auto Resize to Text

The entity will automatically resize to encompass the text in the entity. If you stretch an entity horizontally, the entity will resize horizontally to the closest width, and then resize vertically to include all text. Resizing an entity only vertically will be ignored.

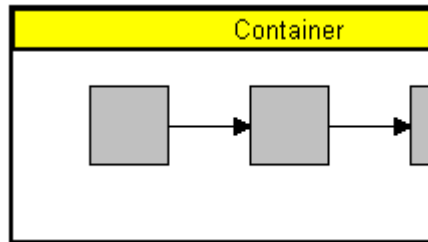
Not a Valid Source if Dragging a New Relation

The entity cannot be used as a source for a relation when using the mouse.

Note: You can use this entity as a relation source if you add relations at runtime invoking InterAct methods. This style only prevents the use of the mouse to drag a relation from the entity.

Act as a Container

The entity will behave as a container. A container is drawn in a special way. A hollow frame is drawn which can surround entities. A caption at the top of the frame is drawn which contains the text. Any entities inside a container will move with the container when the container is moved.



A container entity.

Permanently Hold Objects

This option is only available if **Act as a Container** is enabled. Any objects contained within the container can never be removed from the container. If the user attempts to move an entity from the container using the mouse InterAct will force the entity back into the container.

Transparent

The entity will be transparent - allowing objects “behind” the entity to be seen. This option also affects a graphic which is being displayed. The color white is considered to be the “transparent” color of a bitmap. If a graphic is being displayed in an entity which has this option selected, the image cannot be stretched. See the **Graphics** property page for a description of how graphics are treated in an entity.

3D Effect

A variety of 3D effects can add depth to a diagram. Each entity can have no 3D effect or one of the following: Indent, Heavy Indent, Outdent, Heavy Outdent, Shadow. 3D effects can be used in addition to a **Border**, they are not mutually exclusive.

Border

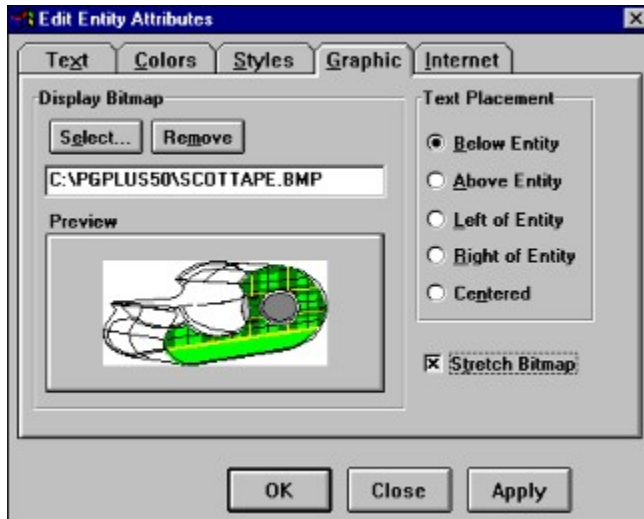
A thin border will be drawn around the entity. The color for this border is the same as the color selected for **Text Color**. See this option on the Colors property page. A border can be used in addition to a **3D Effect**, they are not mutually exclusive.

Entity Shape

Determines the basic shape of an entity. Choices include: Rectangle, Circle (Elliptic) and Round Rectangle. **Border** and **3D Effects** can be applied to all these shapes. If you are displaying a graphic in an entity, it draws as a Rectangle shape regardless of this selection.

Entity Graphic

Allows selection of a bitmap to be displayed in an entity. If a graphic is displayed in an entity, the image will be stretched to fill the entire area of the entity unless the entity is selected to be transparent. In this case, the image will be shown the same size as when it was created. See the Entity Styles property page for more information about the **Transparent** option.



The Entity Graphic property page.

Display Bitmap

The name of a BMP file. May also include a full path to the BMP file. If no path is supplied, the BMP file must be able to be located by InterAct, either in the same directory as InterAct or in a directory in the machine's path. If the BMP can be loaded, it will be loaded and displayed in the window directly below for you to preview the bitmap.

To remove a bitmap, simply remove the bitmap name from the **Display Bitmap** field.

Select

Click the button below the **Display Bitmap** field to select a BMP file on disk. If a BMP file is selected, it will be loaded and displayed in the window. Otherwise the text "Click here to select bitmap." will be displayed.

Remove

Removes a graphic from the entity, clearing the **Preview** window and the **Display Bitmap** field.

Text Placement

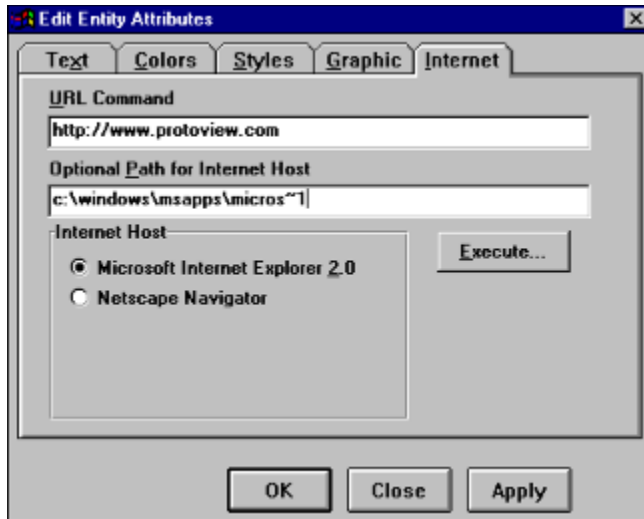
This affects text when a graphic is being displayed in an entity. The text may not be drawn within the borders of the entity. Instead, you can have the text displayed to the left, right, above or below (the default) of the entity. If you select **Centered**, the text will be painted on top of the entity.

Stretch Bitmap

Toggles whether the bitmap will be stretched to occupy the entire area of the entity. If this option is off, the bitmap will be centered in the entity, or clipped if the bitmap is larger than the entity. If the style **Transparent** is selected, **Stretch Bitmap** has no effect.

Internet

The internet property page allows properties to be set to allow InterAct to initiate an internet session. An entities internet capabilities are initiated on a double-click of the entity. Currently most internet hosts allow a URL command to be passed when the host is started. In the future, other internet hosts will allow a continuous stream of URL commands to be passed and processed.



The Entity Internet property page.

URL Command

This is the actual command issued to the internet host.

Optional Path for Internet Host

Often the internet host may not be in the path of the system. In this case it is desirable to supply a path. InterAct will try to locate the internet host both with and without this path, and through any other means available to it.

Internet Host

This is the executable which will provide access to the internet.

Execute

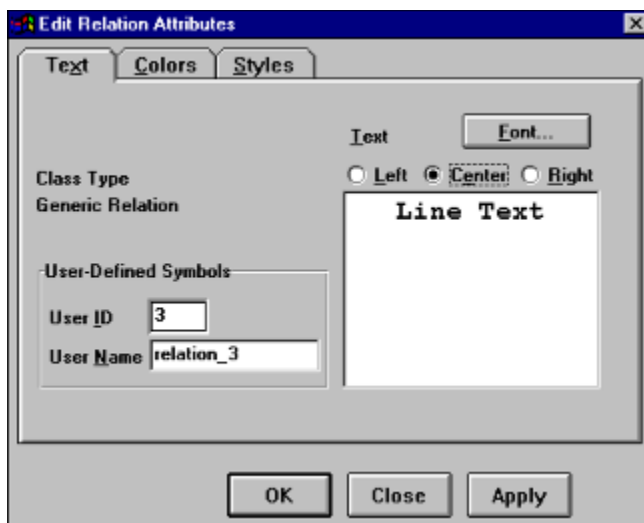
This will execute the URL Command immediately. This is primarily meant as a means to test the properties set on this page.

Relation Property Pages

Relations have three property pages - Text, Colors, and Styles. Relations have a special object of “text” associated with it - called a Line Text. The Line Text is a full-fledged entity in its own right, and can have properties set on it just like a normal entity. However, certain properties in the relation property pages relate directly to this independent Line Text.

Relation Text

The Text property page allows you to set properties which affect the display of a relation. Of course the most obvious is the actual text displayed by the relation, displayed in the Line Text. You can also select the font and orientation of the text. Of equal importance, you can see the ID and Name assigned to the relation. Usually, you will provide the ID or Name when you create the relation, but if you do not supply one, InterAct will create default values. You will also see the class type of the relation.



The Relation Text property page.

Class Type

This is the class of the relation. The class determines what the relation will look like and how it will behave when it is initially created. If the attributes of a class are modified, the relations of that class will automatically assume the new attributes. If rules are being used in InterAct, class name will also be important, as rules refer to how classes can interact.

User ID

A numeric value which uniquely identifies the relation.

User Name

A string which uniquely identifies the relation.

Font

Click this button to select a new font for the relation. Different point sizes and styles can be selected for this font. Each relation's Line Text can have a different font, if requested.

Left / Center / Right

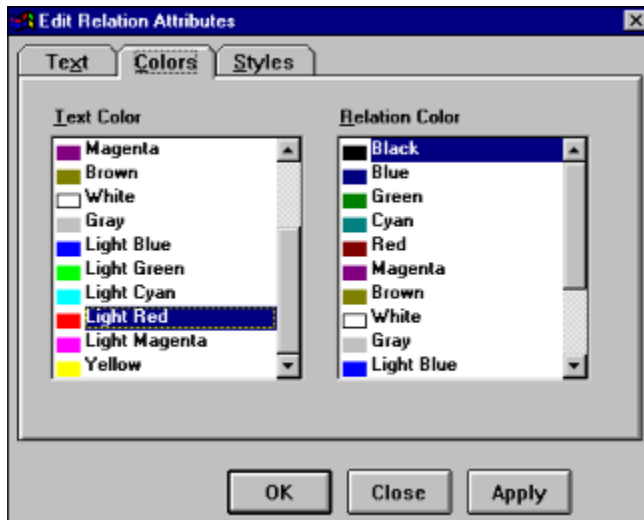
Select the orientation of the text when aligned in the relation's Line Text.

Text

The actual text which will be displayed by the relation's Line Text. The text will be displayed in the selected font and orientation. Text will always be clipped within the region defined by the relation's Line Text, unless the Line Text is selected to automatically size to accommodate all the text. See the Entity Styles property page for details about the **AutoResize** option.

Relation Colors

Allows selection of colors for the relation.



The Relation Colors property page.

Text Color

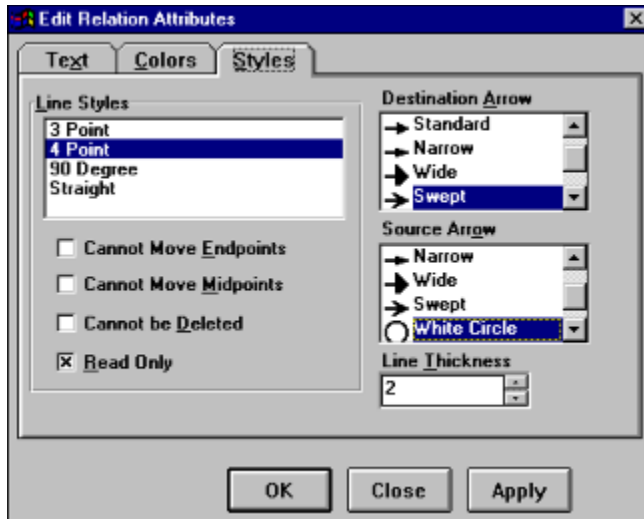
A list of available colors is displayed to choose from when drawing the relation's Line Text. This color will not affect the lines drawn by the relation.

Relation Color

A list of available colors is displayed to choose from when drawing the relation's lines and arrow symbols. The relation color will not affect the Line Text.

Relation Styles

A variety of styles can be applied to each relation which affect the look and behavior of the relation. Any number of styles can be applied, and their effects are cumulative.



The Relation Styles property page.

Line Styles

Determines the basic shape of a relation. Choices include: Straight, Three Point, Four Point and Ninety Degree. Straight lines will draw a straight line from the source to the destination entity. The Three and Four Point lines will have 1 or 2 midpoints which can be positioned. Lines will be drawn from the endpoints to the midpoints, and if necessary, between two midpoints. Ninety Degree lines will always draw lines from the endpoints to the midpoints and between the midpoints at ninety degree angles.

Cannot Move Endpoints

The relation's end points cannot be moved with the mouse. Any attempt to drag an endpoint with the mouse will be ignore.

Cannot Move Midpoints

The relation's mid points, if any, cannot be moved with the mouse. Any attempt to drag a midpoint with the mouse will be ignore.

Cannot be Deleted

The relation cannot be deleted if the relation is currently selected and the **Delete** key is pressed.

Read Only

Normally, double-clicking a relation will allow you to edit the relation's Line Text. This style suppresses this behavior, creating a "read-only" relation.

Destination Arrow

A number of different arrow heads are available to display at the relation's destination point. Choices include: None, Standard, Narrow, Wide, Swept, Light Circle, and Dark Circle.

Standard	→	Swept	→
Narrow	→	White Circle	○
Wide	→	Dark Circle	●

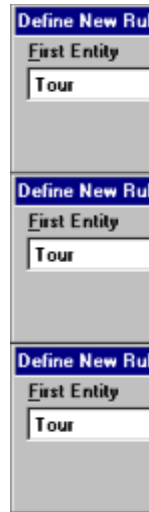
Source Arrow

A number of different arrow heads are available to display at the relation's source point. Choices include: None, Standard, Narrow, Wide, Swept, White Circle, and Dark Circle.

Standard

A vertical stack of four rectangular boxes. The top box has a blue header 'Define New Rule' and contains the text 'First Entity' and 'Tour'. The second box is empty. The third box is empty. The bottom box is empty.

Swept

A vertical stack of four rectangular boxes. The top box has a blue header 'Define New Rule' and contains the text 'First Entity' and 'Tour'. The second box is empty. The third box is empty. The bottom box is empty.

Narrow

A vertical stack of four rectangular boxes. The top box has a blue header 'Define New Rule' and contains the text 'First Entity' and 'Tour'. The second box is empty. The third box is empty. The bottom box is empty.

White Circle

A vertical stack of four rectangular boxes. The top box has a blue header 'Define New Rule' and contains the text 'First Entity' and 'Tour'. The second box is empty. The third box is empty. The bottom box is empty.

Wide

A vertical stack of four rectangular boxes. The top box has a blue header 'Define New Rule' and contains the text 'First Entity' and 'Tour'. The second box is empty. The third box is empty. The bottom box is empty.

Dark Circle

A vertical stack of four rectangular boxes. The top box has a blue header 'Define New Rule' and contains the text 'First Entity' and 'Tour'. The second box is empty. The third box is empty. The bottom box is empty.

Line Thickness

Varies the thickness of the lines being drawn. This can be in the range of 1-5 units wide. When the cursor is in this field, the value can be incremented/decremented by pressing the **Up Arrow** and **Down Arrow** keys, respectively.

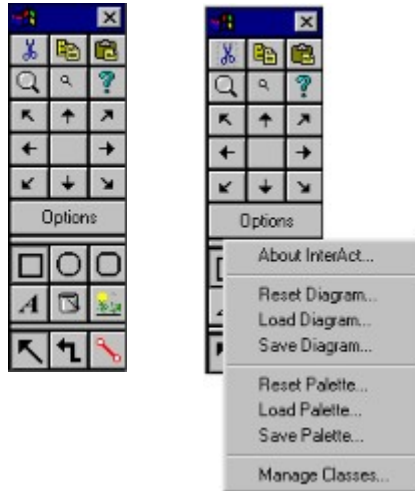
Summary

This chapter has explained all the property pages which InterAct can display for itself, and the entities and relations in a diagram. It has explained each option of all the different property pages for each of the objects contained in a diagram.

Overview

The tools palette is an important interface for performing operations on InterAct. With the tools palette you can

- Cut/Copy/Paste items to the Windows Clipboard.
- Zoom the perspective of InterAct.
- Display Help
- Select default classes of entities and relations to paint in InterAct.
- Load/Save a diagram.
- Load/Save a palette.
- Display a dialog to manage classes and rules.



The tools palette contains the following buttons:

Cut

Delete the selected items and place them in the clipboard. You cannot cut a relation if it is selected. However, if a relation connects two entities which are selected and cut, the relation will also be cut to the clipboard.

Copy

Copy the selected items into the clipboard. You cannot copy a relation if it is selected. However, if a relation connects two entities which are selected and copied, the relation will also be copied to the clipboard.

Paste

Copy items from the clipboard to InterAct. Selected entities which were cut or copied to the clipboard will be pasted into the diagram. If a relation connected any two entities being pasted, then the relation will also be pasted into the diagram.

Zoom In

Zoom the InterAct window for more detail.

Zoom Out

Zoom the InterAct window to see more of the diagram but less detail.

Help

Display the InterAct help file.

Note: If you wish to display a help file which you have defined and not the help file provided with InterAct, you can respond to a notification from InterAct and display your help file instead. See *Chapter 10: InterAct Events*.

Options

Display a list of options for the InterAct environment. Of particular importance is the ability to load and save diagrams, palettes, and display the *ManageClasses* dialog.

Note: It is possible to change the text on the Options button and customize the menu which drops down when this button is clicked. See Chapter 10: InterAct Events and Chapter 12: InterAct Methods.

The Speed Buttons

The speed buttons provide a quick way to add entities and relations to a diagram. Clicking on any of the speed buttons adds an entity of the *default entity class*, relative to the current entity. It also adds a relation of the *default relation class* from the current entity to the new entity. (See the items **Entity Classes** and **Relation Classes** in this section for a description of default classes.) If no entity is currently selected in the diagram, the new entity is placed in the middle of the current view of the diagram, and no new relation is created.

Entity Classes

The tools palette will display all the available entity classes as buttons, one button per class. Classes usually display a bitmap to convey their use or representation. These buttons can be toggled, meaning that when clicked, they will remain in a depressed position. Only one button of the entity classes will ever be depressed at any time, and one button will always be depressed. This depressed button represents the *default entity class*. The default entity class is the class which will be added to a diagram when one of the tool palette's **speed buttons** is clicked.

In addition, when you click one of the buttons on the palette, you can immediately add an entity of the selected class to the diagram using the mouse. Click on one of the entity symbols on the tools palette. Move the mouse over InterAct. The mouse cursor will change shape to a hand with a pointer. Click and hold down the left mouse button. Holding the left mouse button down, drag the mouse. A rectangle will be drawn as you drag the mouse to indicate the area which will be occupied by the new entity. Release the left mouse button. A new entity will be added to the diagram. Repeat this procedure to add any number of entities.

Relation Classes

The tools palette will display all the available relation classes as buttons, one button per class. Classes usually display a bitmap to convey their use or representation. These buttons can be toggled, meaning that when clicked, they will remain in a depressed position. Only one button of the relation classes will ever be depressed at any time, and one button will always be depressed. This depressed button represents the *default relation class*. The default relation class is the class which will be added to a diagram when one of the tool palette's **speed buttons** is clicked. An entity in the diagram must be selected for a relation to be added in this manner. The new relation will be added pointing from this selected entity to the new entity added to the diagram.

In addition, when you click one of the buttons on the palette, you can immediately add a relation of the

selected class to the diagram using the mouse. Click on one of the relation symbols on the tools palette. Move the mouse over InterAct. The mouse cursor will change shape to a hand with a pointer. Click and hold down the left mouse button on an entity. This will be the relation source. Holding the left mouse button down, drag the mouse. A line will be drawn as you drag the mouse to indicate the area which will be occupied by the new relation. Release the left mouse button when over a target entity. A new relation will be added to the diagram connecting the source and destination entities. Repeat this procedure to add any number of relation.

Customizing the Tools Palette

The tools palette is a major interface with InterAct. With the tools palette you can easily add entities and relations to a diagram and manage all the different classes available within a diagram. You can move the tools palette anywhere on the screen, not just clipped within InterAct's window.

Summary

This chapter has introduced InterAct's tool palette. The tools palette is an optional component for interacting with InterAct. With it, the user can quickly and easily add entities and relations to a diagram and manage all the classes used in the diagram.

Introduction

InterAct allows you define classes and rules which affect its built-in behavior. Classes describe how particular entities and relations will look and behave and rules govern when particular relations can be used to connect entities.

Entity Classes

These describe how a particular entity will look and behave when created. For example, if you want all of your entities to appear with black text on a white background with a shadow 3D effect, you can define a class with these attributes. Whenever an entity of that class is created it will automatically be created with these attributes. In addition, if you change the definition for a class, all the entities of that class will automatically inherit that change.

Relation Classes

These describe how a particular relation will look when created. For example, if you want all of your relations to appear red with a wide arrow shape, you can define a class with these attributes. Whenever a relation of that class is created it will automatically be created with these attributes. In addition, if you change the definition for a class, all the relations of that class will automatically inherit that change.

Rules

Rules define when it is legal to have a relation connect two entities. For example, if you have two entities, one of a class called “Person” and the second from a class called “Job,” you can define a rule which allows a relation class “Works” to be connected from “Person” to “Job.” This can be described with the notation:

Object <Action> Object.

In our example, the rule would be *Person <Works> Job.*

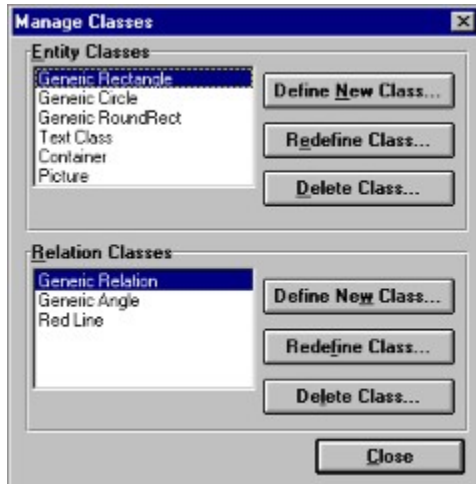
With only this single rule, InterAct would prevent the creation of a relation of class “Works” from a class “Jobs” to a class “Person,” or from “Person” to “Person” or class “Job” to “Job.” Rules and classes are an important component for allowing InterAct to validate the visual selections made by a user.

What do classes and rules provide for your diagram? When representing internal data structures in a visual way, it is often necessary to restrict the types of objects that can be displayed, and how they can interact. Rules and classes provide the mechanism for this type of validation.

Also, when you change an attribute on a class, all the entities or relations of that class will automatically inherit the new attributes. This provides a convenient method to make global changes in a diagram without having to iterate through all the objects in the diagram.

Entity Classes

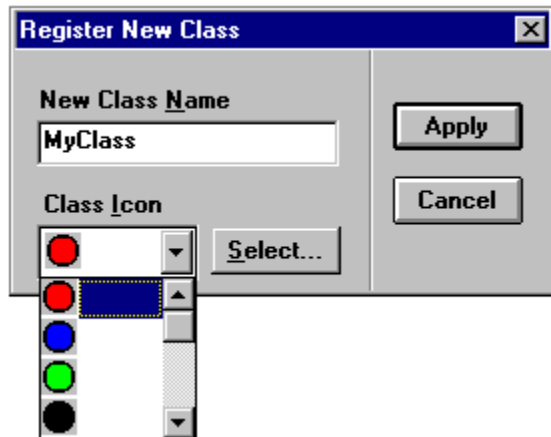
You can define an entity class by selecting the option *Manage Classes* from the tools palette **Options** button. The *Registered Classes* dialog will appear which allows you to manage all your entity classes.



In the **Entity Classes** group is a list of all the available classes.

How to Define a New Entity Class

Click the **Define New Class** button in the **Entity Classes** group. The *Register New Class* dialog will appear.



Enter the name of the new class in the **New Class Name** field. The class name must be unique. You can also select a class icon in the **Class Icon** combobox. The class icon is displayed on the Tools Palette. InterAct supplies a set of predefined class icons, or you can press the **Select** button to select your own bitmap graphic to represent the class icon.

Note: Bitmaps displayed on the tools palette should be 17 x 17 pixels in size to appear properly.

When satisfied with your selections, click the **Apply** button. Click the **Cancel** button to abort the class definition.

If you click the **Apply** button, the dialog *Register New Class: <classname>* will appear. This dialog has a series of property pages allowing you to customize the look of your new entity class. When finished customizing these details, click the **OK** button to accept the new class or click the **Cancel** button to abort the process.

How to Modify an Entity Class

Select a class in the **Entity Classes** group list. Click the **Modify Class** button. You will be taken to the *Redefine Class: <classname>* dialog. This dialog contains a series of property pages allowing you to customize the look of your existing entity class. When finished making changes, click the **OK** button to accept the changes or click the **Cancel** button to abort the changes.

If you click **OK**, any entities of this class in your diagram will automatically be updated with the new changes.

How to Delete an Entity Class

Select a class in the **Entity Classes** group list. Click the **Delete Class** button. You will be prompted to make sure you want to delete the selected class. Click **Yes** to delete, **No** to abort.

How to Select a Default Entity Class

Click on the upper-left corner of the **Entity Classes** button. By holding the left mouse button down, a list of all the available classes will be displayed under the button. Keeping the left mouse button depressed, move the mouse cursor over the desired class you wish to make the default class. The classes will be highlighted as you move the mouse cursor over them. Once you have selected the desired entity class, release the mouse button. If you wish to abort the class selection, press the **Escape** key.

Relation Classes

You can define a relation class by selecting the option *Manage Classes* from the tools palette **Options** button. The *Registered Classes* dialog will appear, which allow you to manage all your **Relation Classes**.

How to Define a New Relation Class

Click the **Define New Class** button in the **Relation Classes** group. The *Register New Class* dialog will appear.



Enter the name of the new class in the **New Class Name** field. The class name must be unique. You can also select a class icon in the **Class Icon** combobox. The class icon is displayed on the Tools Palette. InterAct supplies a set of predefined class icons, or you can press the **Select** button to select your own bitmap graphic to represent the class icon.

Note: Bitmaps displayed on the tools palette should be 16 x 16 pixels in size to appear properly.

When satisfied with your selections, click the **Apply** button. Click the **Cancel** button to abort the class definition.

If you click the **Apply** button, the *Register New Class: <classname>* dialog will appear. This dialog contains a series of property pages allowing you to customize the look of your new relation class. When finished supplying these details, click the **OK** button to accept the new class or click the **Cancel** button to abort the process.

How to Modify a Relation Class

Select a class in the **Relation Classes** group list. Click the **Modify Class** button. You will be taken to the *Redefine Class: <classname>* dialog. This dialog contains a series of property pages allowing you to customize the look of your existing relation class. When finished making changes, click the **OK** button to accept the changes or click the **Cancel** button to abort the changes.

If you click **OK**, any relations in your diagram of this class will automatically be updated with the new changes.

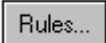
How to Delete a Relation Class

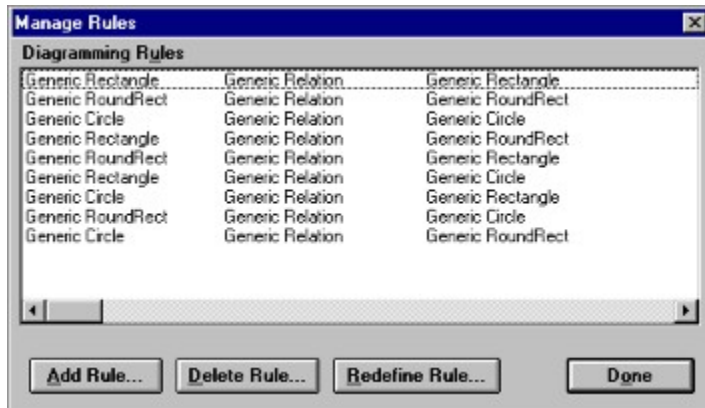
Select a class in the **Relation Classes** group list. Click the **Delete Class** button. You will be prompted to make sure you want to delete the selected class. Click **Yes** to delete, **No** to abort.

How to Select a Default Relation Class

Click on the upper-left corner of the **Relation Classes** button. By holding the left mouse button down, a list of all the available classes will be displayed under the button. Keeping the left mouse button depressed, move the mouse cursor over the desired class you wish to make the default class. The classes will be highlighted as you move the mouse cursor over them. Once you have selected the desired relation class, release the mouse button. If you wish to abort the class selection, press the **Escape** key.

Rules

Press the **Rules** button  on the *Manage Classes* dialog to display the *Manage Rules* dialog. The *Manage Rules* dialog allows you to manage all your rules by adding new rules, modifying existing rules or deleting rules.



In the center of the dialog is the **Diagramming Rules** list which contains all the Rules in effect for the diagram.

How to Add a Rule

To add a new rule, press the **Add Rule** button. The *Define New Rule* dialog will be displayed.



You can select the source entity class, relation class, and destination entity class that compose the rule. When satisfied, click the **Apply** button.

How to a Delete Rule

Select one or more rules in the **Diagramming Rules** list. Click the **Delete Rule** button. You will be prompted to make sure you want to delete the selected rule. Click **Yes** to delete, **No** to abort.

How to Modify a Rule

Select a rule in the **Diagramming Rules** list. Click the **Redefine Rule** button. Or double-click on a rule in the **Diagramming Rules** list. The *Redefine Rule* dialog will be displayed, allowing you to modify the existing rule.

Enforcing Rules

Often, it is not desirable to have rules enforced in a diagram. This level of support can be toggled by using the property **RulesEnforced**. See *Chapter 11: InterAct Properties* for a description of this property. By

default, rules are not enforced in a diagram.

Summary

This chapter has introduced the concepts of *Entity Classes*, *Relation Classes*, and *Rules*. With these features of InterAct, very sophisticated interactions can be achieved with little effort.

Introduction

This tutorial will show how to use InterAct using Visual C++ 4.x. This tutorial will show how you can:

- register InterAct in the Developer Studio's Component Gallery
- add InterAct to a dialog and set properties on InterAct
- programmatically add entities and relations to a diagram
- respond to mouse clicks within the InterAct environment

If you want a tutorial regarding how to use InterAct using the mouse and keyboard, see *Chapter 1: A Guided Tour of InterAct*.

Adding InterAct to an Application

Creating a Generic Program

Choose the **File | New** menu item to invoke the *New* dialog. Choose “Project Workspace” from the list and press **OK**. The *New Project Workspace* dialog is now visible. Choose “MFC AppWizard (exe)” from the **Type** list box. Enter “idodemo” in the **Name** edit box, and press the **Create** button. On the *MFC AppWizard - Step 1* dialog, choose **Dialog Based** as the type of application and press the **Next** button. On the *MFC AppWizard - Step 2* dialog, select the **OLE Controls** checkbox and press the **Finish** button. On the *New Project Information* dialog, press the **OK** button.

Installing InterAct in Microsoft's Developer Studio

The 32-bit InterAct ActiveX can be installed into Microsoft's Developer Studio's Component Gallery. If the InterAct ActiveX is properly registered it will automatically be added to the component gallery.

Note: VC++ 4.0 did not automatically add registered controls to the component gallery. You must explicitly add InterAct to the component gallery. To add InterAct to the Component Gallery, choose the menu item **Insert | Component** to invoke the *Component Gallery* dialog. From the **OLE Controls** tab page, choose the "ProtoView InterAct Object" icon, and press the **Insert** button. Press the **OK** button on the *Confirm Class* dialog. Press the **Close** button on the *Component Gallery* dialog.

To add the InterAct ActiveX to an application you must be sure to insert InterAct from the Component Gallery into your application. In Visual C++ select **Insert | Component** to invoke the *Component Gallery* dialog. Click the **OLE Controls** page of the page of the *Component Gallery*.

Highlight InterAct and click the **Insert** button. VC++ will confirm that you want to add the InterAct ActiveX to your project. Including the InterAct ActiveX will add several source and header files to your project. Click **OK** to confirm the addition of these files and click **Close** to close the *Component Gallery*.

Adding InterAct to a Dialog

From the resource page of the *Project Workspace* window, double-click the entry “IDD_IDODEMO_DIALOG”.



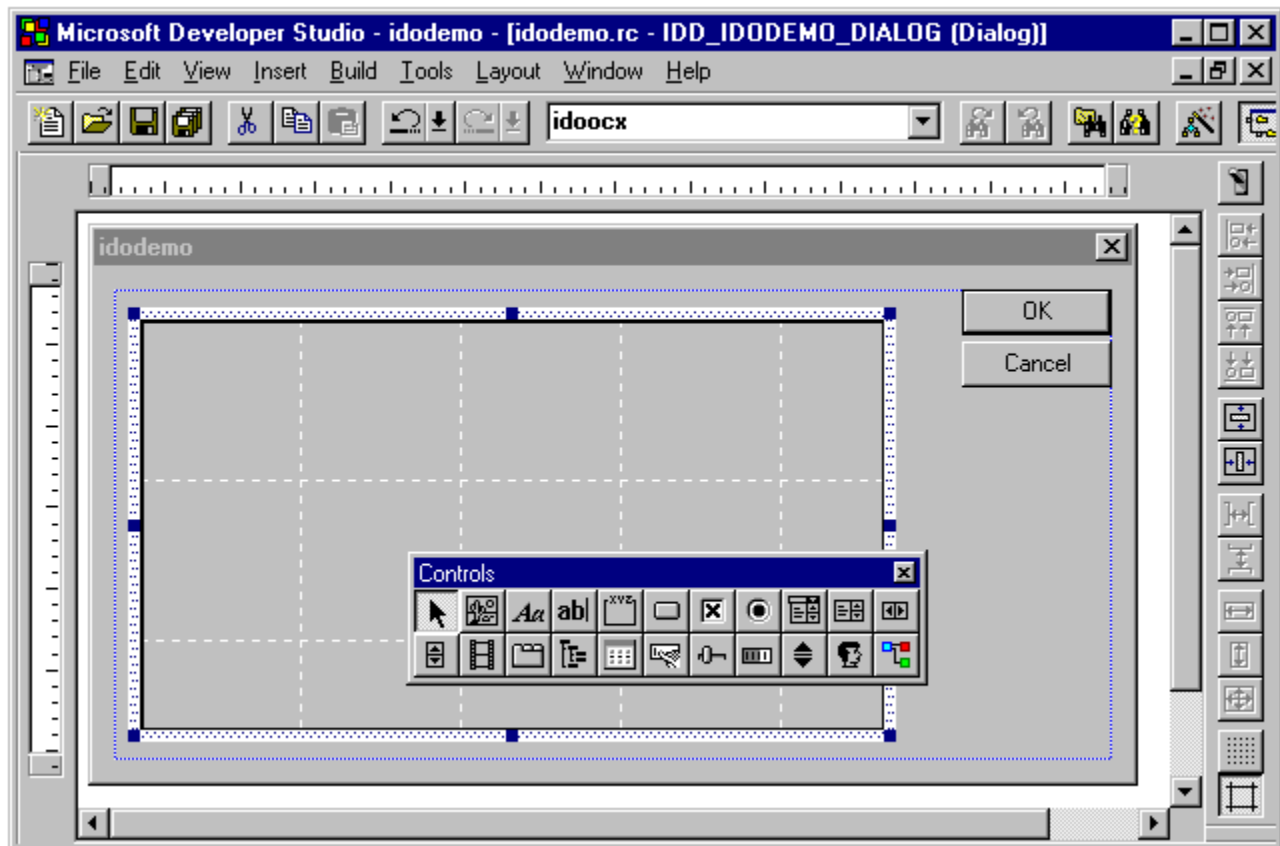
The Project Workspace window.

Delete the “TODO: Place dialog controls here.” static text field. Stretch the new dialog so that it is a bit bigger and move the **OK** and **Cancel** buttons so they are still on the right edge of the dialog.

Select the InterAct icon from the control palette and place the control on the dialog.

Note: If InterAct does not appear on your control palette see the section **Installing the IDO in Microsoft’s Developer Studio**.

Size the InterAct control so that it takes up most of the dialog.



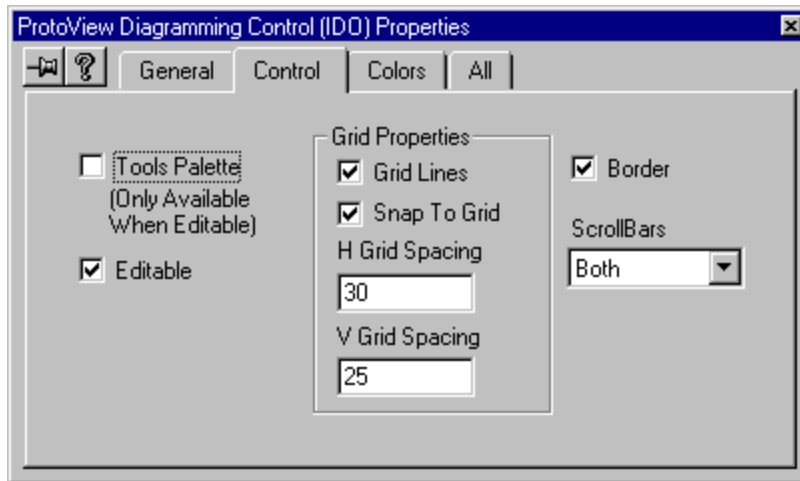
InterAct when designing in AppStudio.

InterAct will be drawn as a simple rectangle with gridlines drawn at even intervals.

Setting Properties on InterAct

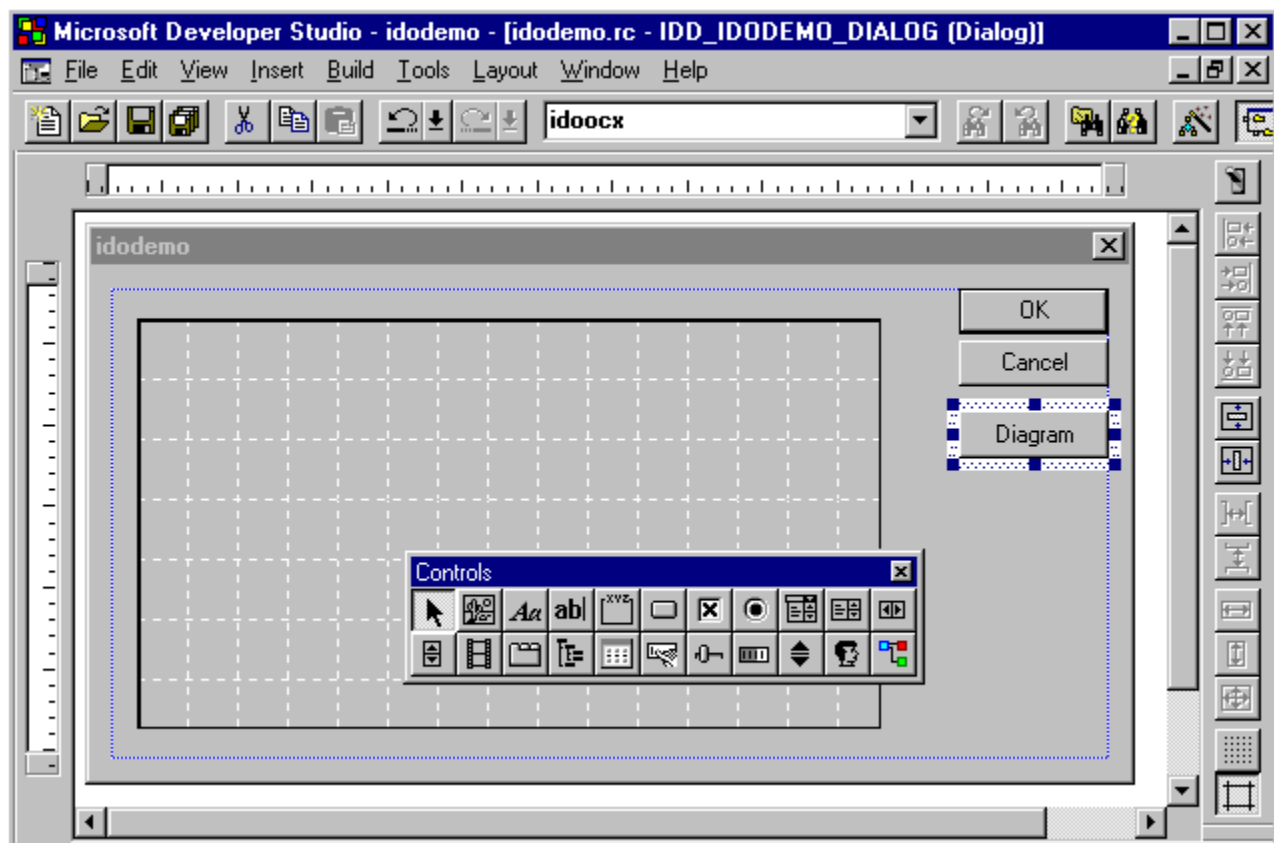
Double click on the InterAct control to invoke the properties window. InterAct has several properties which can be set. Switch to the **Control** property page.

Check off the property **Tools Palette**. In the **Grid Properties** group, set **H Grid Spacing** to “30” and **V Grid Spacing** to “25.” Set **ScrollBars** to “Both.”



The InterAct Properties sheet.

Close the property page. These properties will now be applied to InterAct. Notice that InterAct is now displaying the grid lines with the specified spacing.



InterAct after the properties have been set.

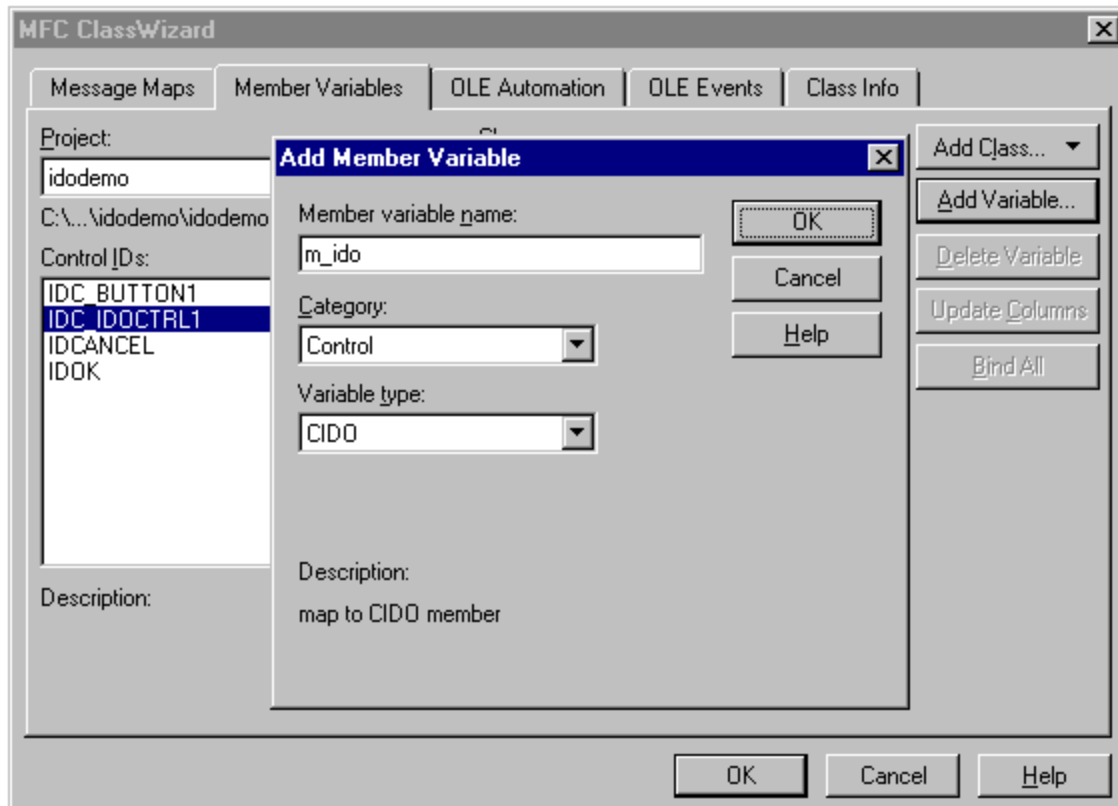
Programming InterAct

Most applications will want to load their own diagram into InterAct. We will now demonstrate how to dynamically create a diagram in InterAct.

Adding Elements to a Diagram

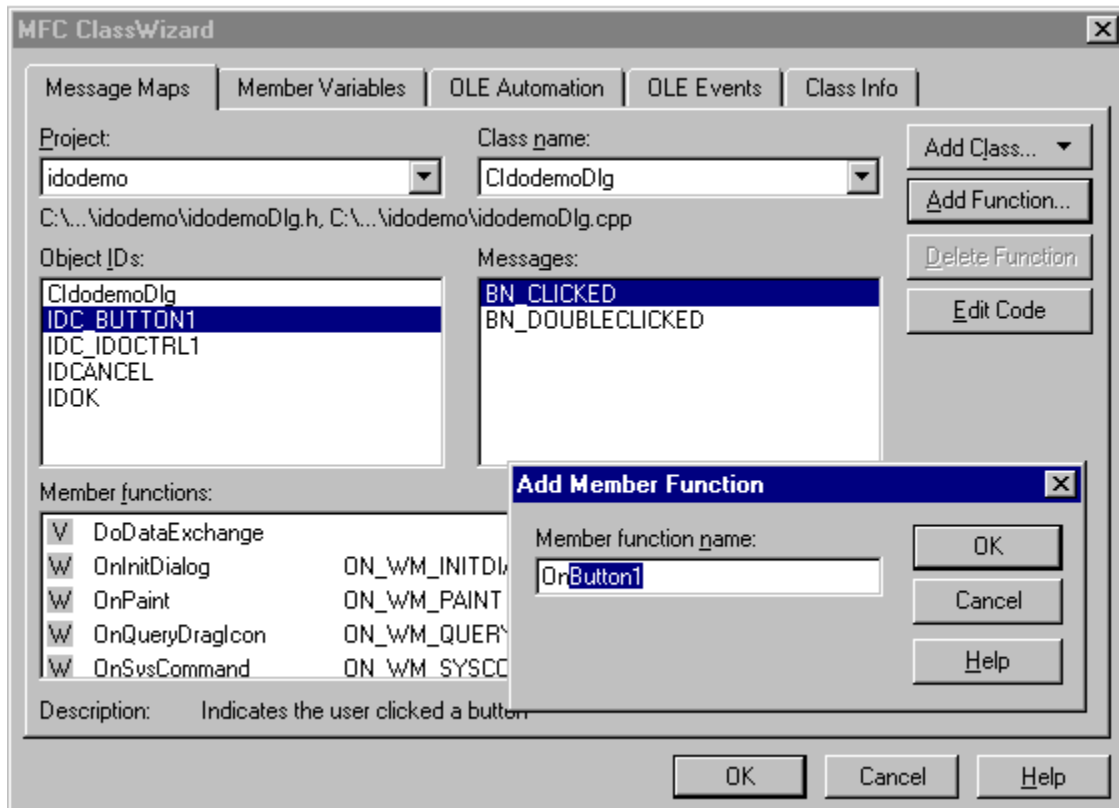
Add a button to the dialog under the **Cancel** button. Double-click the new button and set the **Caption** to “Diagram.” Close the property window to accept this change.

Invoke ClassWizard by pressing CTRL+W. Select the **Member Variables** tab page. Select IDC_IDOCTRL1 from the **Control IDs** list box and press the **Add Variable** button. On the **Add Member Variable** dialog box, enter “m_ido” in the **Member variable name** edit box. Press the **OK** button.



Add a member variable to the project in Class Wizard.

Switch to the **Message Maps** tab page, select “IDC_BUTTON1” from the **Object IDs** list box and “BN_CLICKED” from the **Messages** list box. Press the **Add Function** button.



Add a function to a button in Class Wizard.

Press **OK** on the *Add Member Function* dialog. Press **Edit Code** on the *ClassWizard* dialog.

```
void CidodemoDlg::OnButton1()
{
    m_ido.ResetDiagram();

    // add two entities connected with a relation
    m_ido.AddEntityFromClass(1, "", "Generic Rectangle",
        "First Node", 100, 100, 50, 50);
    m_ido.AddEntityFromClass(2, "", "Generic Rectangle",
        "Last Node", 200, 100, 50, 50);
    m_ido.AddRelationFromClass(0, "Link", "Generic Relation",
        "", 1, "", 2, "");

    // disable InterAct's popup menu for entities
    m_ido.SetPopupMenu(IDOMENU_ENTITY, FALSE);
}
```

Code for adding objects to a diagram.

At the top of the file, add a helper header file provided with InterAct to define the constants used by InterAct.

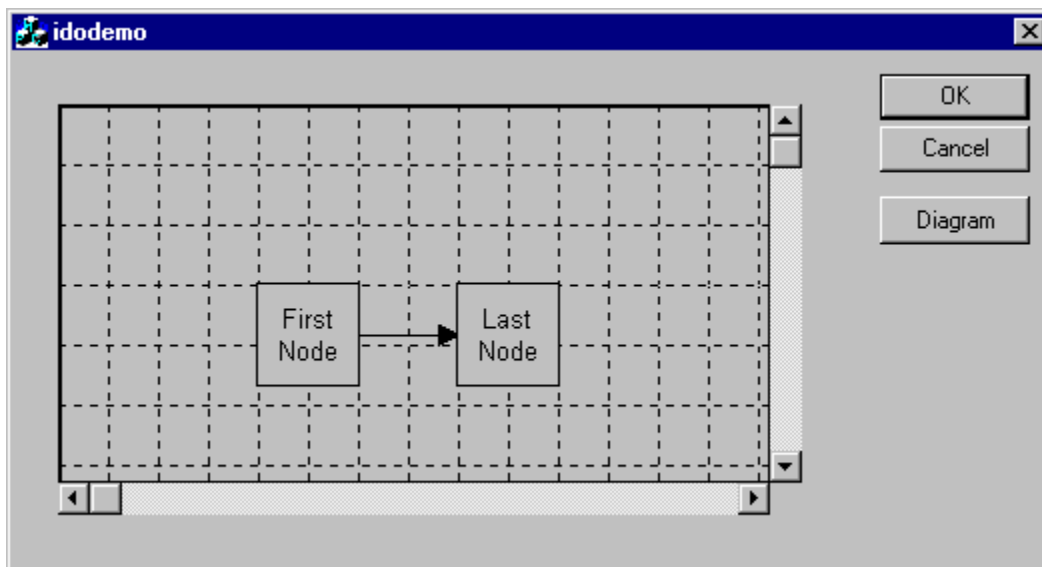
```
#include "idodef.h" // InterAct constant header file
```

The button **Command1** will clear the diagram by calling the InterAct's **ResetDiagram** method and then add two entities and a relation connecting these entities.

The method **AddEntityFromClass** takes several parameters. The first two allow you to uniquely identify the entity we are about to add either with a number or [a string](#). In this example we are just supplying a number. Next is an entity class name. The class “Generic Rectangle” is a default class available with a new InterAct control. Classes are simply an easy way to specify how an entity will look and behave without having to supply all these details when you create the entity. Next, text which will be displayed by the entity is passed. Finally, the last four digits are the coordinates of the new entity.

The method **AddRelationFromClass** also takes several parameters. Like the **AddEntityFromClass** method, the first two allow you to uniquely identify the relation we are about to add either with a number or [a string](#). In this example we are just supplying [a string](#). Next is a relation class name. The class “Generic Relation” is a default class available with a new InterAct control. Next, the ID or name of a source and destination entity are needed. Note that we are passing the IDs assigned to the entities created with the **AddEntityFromClass** methods.

Compile and run the program by selecting the menu items **Build | Build idodemo.exe** and **Build | Execute idodemo.exe**. Press the button **Diagram**. A simple diagram of two entities connected by a relation will be created. One entity will contain the text “First Node” and the other will contain the text “Last Node.” You can use the mouse and keyboard to move and resize the entities. Stop the program by clicking the **OK** button and return to the programming environment.



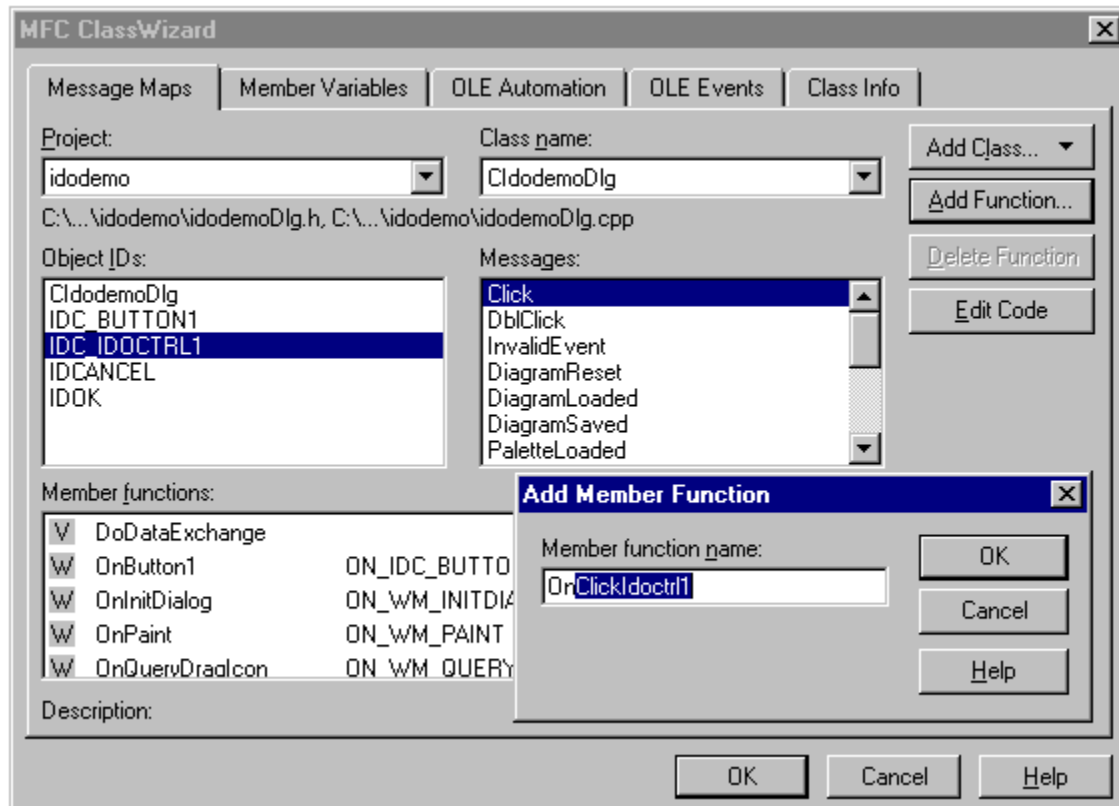
InterAct at runtime with a diagram.

Responding to Events

Next we are going to respond to a click event to query the diagram. We will determine which entity was clicked and retrieve the text of that entity. We will then display that text to the user in a message box.

Add an Event Procedure

Invoke ClassWizard again by pressing CTRL+W. Select the **Message Maps** tab page. Select “IDC_IDOCTRL1” from the **Object IDs** list box and “ClickEntity” from the **Messages** list box. Press the **Add Function** button..



Adding an event with Class Wizard.

Press **OK** on the *Add Member Function* dialog. Press **Edit Code** on the *ClassWizard* dialog.

```
void CIdodemoDlg::OnClickEntityIdoctrl1(long MouseStatus)
{
    // TODO: Add your control notification handler code here

    CEntity Entity;
    CString Text;

    if ( MouseStatus == IDO_RIGHTCLICK )
    {
        Entity = m_ido.GetNotifyEntity();
        Text = Entity.GetText();

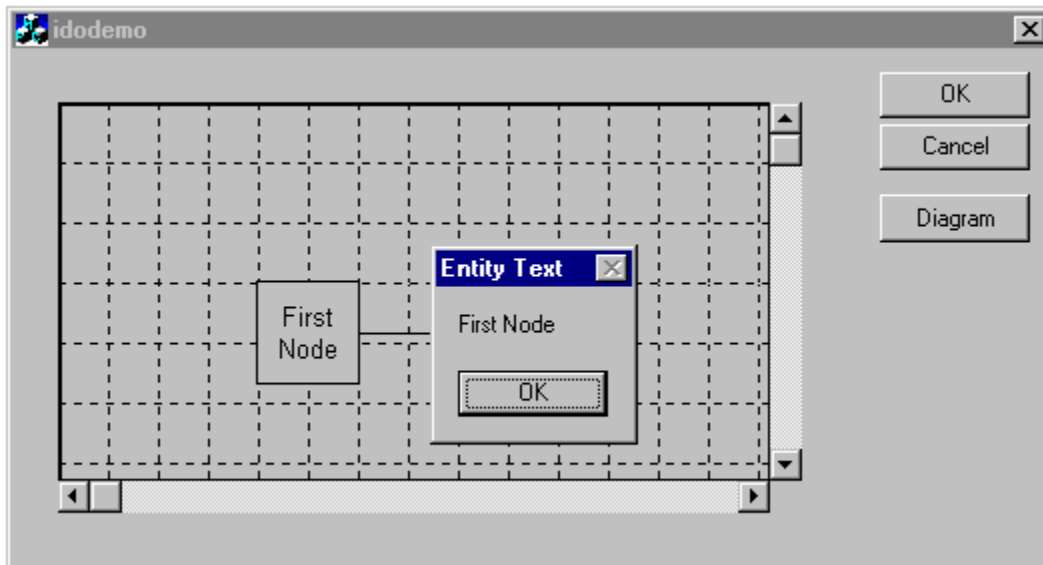
        MessageBox((LPSTR)Text.GetBuffer(0), "Entity Text", MB_OK);
    }
}
```

Code to respond to a click event.

At the top of the file, add another header for the Entity object.

```
#include "entity.h"           // Entity object header
```

Recompile and run the program. Click the **Diagram** button. The program output will appear the same but by right-clicking on an entity, a message box with the entity's text will appear.



*InterAct when the entity **First Node** is clicked.*

Conclusion

This concludes the Visual C++ tutorial of InterAct. We have shown just some of the basic features and capabilities of InterAct. When using InterAct in Visual C++ you will use these same four techniques:

- Insert InterAct from the Component Gallery into your application.
- Place InterAct on a form and set properties for InterAct.
- Add code to invoke methods and properties of InterAct to create or alter diagrams.
- Respond to events from InterAct to perform special actions.

Introduction

This tutorial will show how to use InterAct using Visual Basic 4.0 32-bit. This tutorial will show how you can:

- add InterAct to a Visual Basic project
- add InterAct to a form and set properties on it
- programmatically add entities and relations to a diagram
- respond to mouse clicks within the InterAct environment

If you want a tutorial regarding how to use InterAct using the mouse and keyboard, see *Chapter 1: A Guided Tour of InterAct*.

Installing InterAct in Visual Basic

Adding InterAct to the default Visual Basic Project:

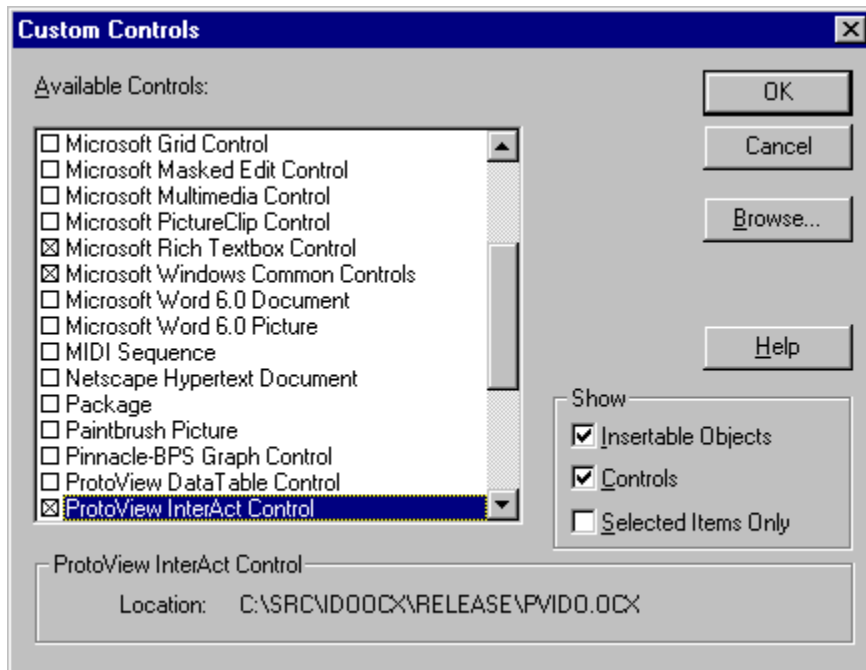
To add InterAct to the default project, you must first open the default project. Choose the **File | Open Project** menu item to invoke the *Open Project* dialog. Select the project “Auto32ld.vbp” from your Visual Basic directory and press the **OK** button. Now when you create a new project as described under *Create a New Project*, InterAct will automatically be available to that project. Just follow the steps beginning at *Adding InterAct to a Form*.

Create a New Project:

Start a new project by choosing the menu **File | New Project**. To use InterAct in Visual Basic, InterAct must be added to your project. The ActiveX can either be added to each project that uses InterAct or added to the default project. By adding the ActiveX to the default project, the InterAct ActiveX is automatically loaded whenever you start Visual Basic or create a new project.


Adding InterAct to a Visual Basic Project:

If InterAct is not part of the default project and you wish to add it to an existing or a new project, choose the **Tools | Custom Controls** menu item to invoke the *Custom Controls* dialog to add the InterAct ActiveX file.



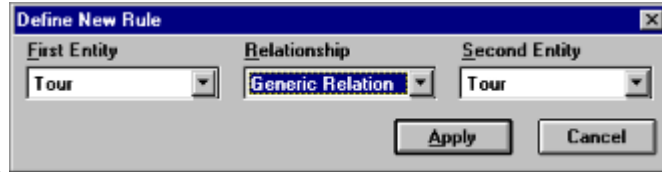
Adding InterAct in Visual Basic

Select “ProtoView InterAct Control” from the **Available Controls** list box and press the **OK** button. If “ProtoView InterAct Control” is not visible in the list, use the **Browse** button to locate and register the PVIDEO32.OCX.

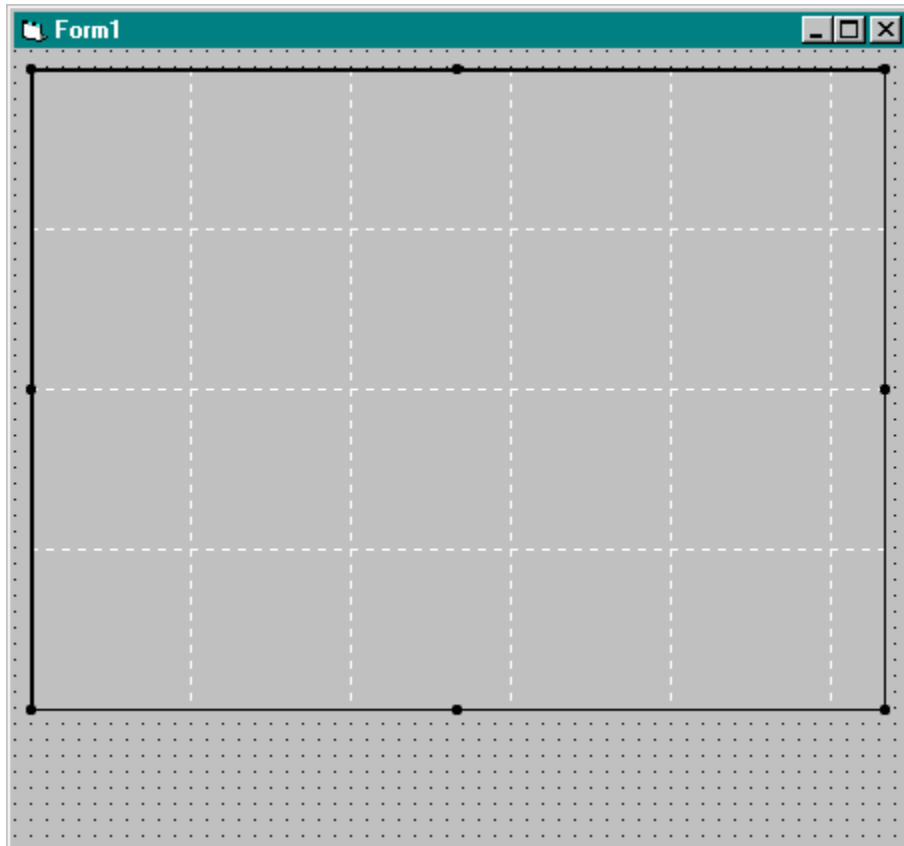
The InterAct icon, , is now displayed in the *Toolbox*. InterAct behaves like any other control. It can be dragged from the *Toolbox* to a form, its properties can be retrieved or set, and event procedures can be

created.

Adding InterAct to a Form



Select the InterAct icon, from the *Toolbox* and draw the InterAct control on the form. Resize InterAct so that it takes up most of the form.



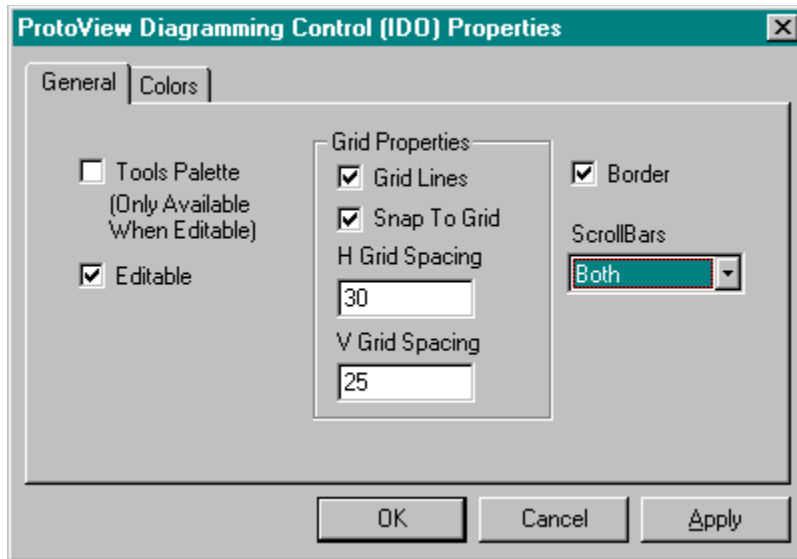
InterAct during design mode.

InterAct when drawn will be a simple rectangle with gridlines drawn at even intervals.

Setting Properties on InterAct:

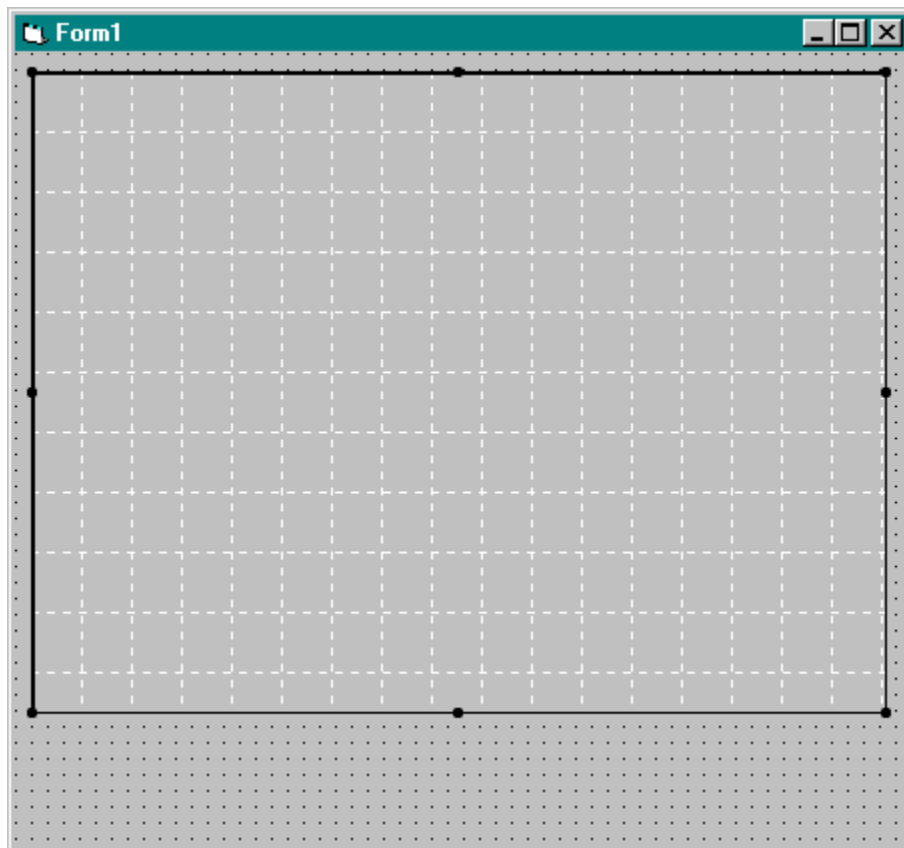
Right click on InterAct and choose “Properties” from the floating menu. InterAct has several properties which can be set. Switch to the **General** property page.

Check off the property **Tools Palette**. In the **Grid Properties** group, set **H Grid Spacing** to “30” and **V Grid Spacing** to “25.” Set **ScrollBars** to “Both.”



InterAct Properties sheet

Click the **OK** button. These properties will now be applied to InterAct. Notice that InterAct is now displaying the grid lines with the specified spacing.



InterAct after the properties have been set.

Programming InterAct

Most applications will want to create their own diagrams in InterAct. We will demonstrate how to dynamically create a simple diagram in InterAct.

Adding Code to Command1:

Add a button to the form. By default it will be called **Command1**. Double click on the button to add an event procedure to the *command1_click* event. Add the following code to the button **Command1**:

```
Rem clear any objects from the diagram
ido1.ResetDiagram

Rem add two entities and a relation
ido1.AddEntityFromClass 1, "", "Generic Rectangle", "First Node", 100, 100, 50, 50

ido1.AddEntityFromClass 2, "", "Generic Rectangle", "Last Node", 200, 100, 50, 50

ido1.AddRelationFromClass 0, "Link", "Generic Relation", "", 1, "", 2, ""

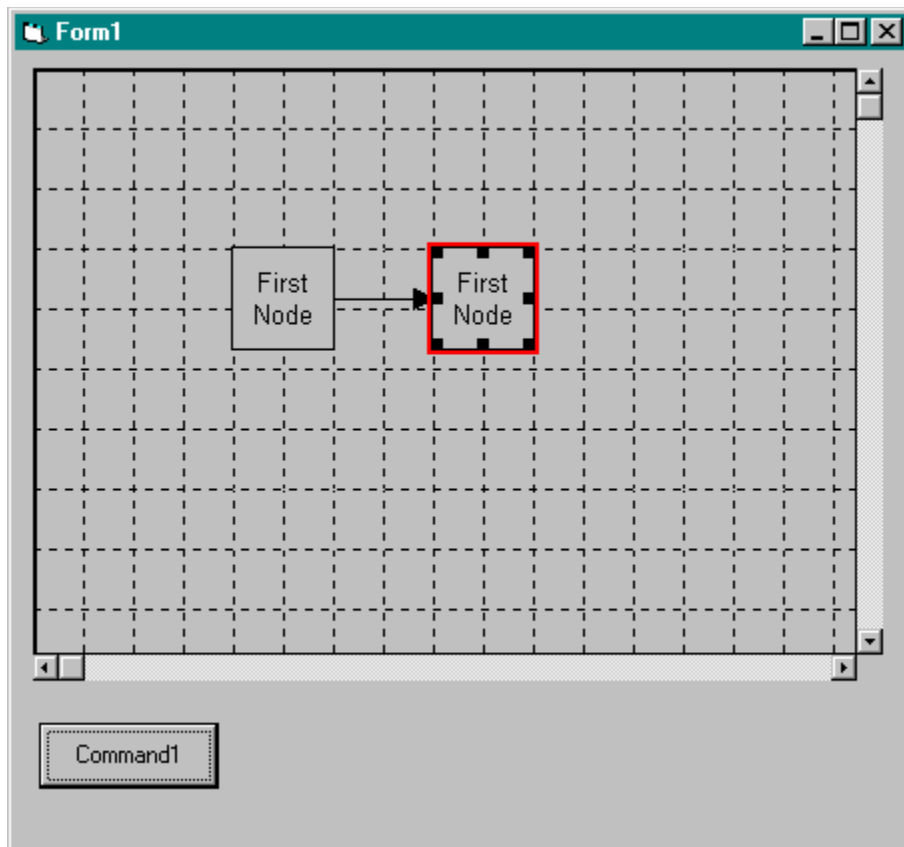
Rem disable the InterAct's default menu for entity objects
ido1.PopupMenu (IDOMENU_ENTITY) = FALSE
```

The button **Command1** will clear the diagram by calling the InterAct's **ResetDiagram** method and then add two entities and a relation connecting these entities.

The method **AddEntityFromClass** takes several parameters. The first two allow you to uniquely identify the entity we are about to add either with a number or [a string](#). In this example we are just supplying a number. Next is an entity class name. The class "Generic Rectangle" is a default class available with a new InterAct control. Classes are simply an easy way to specify how an entity will look and behave without having to supply all these details when you create the entity. Next, text which will be displayed by the entity is passed. Finally, the last four digits are the coordinates of the new entity.

The method **AddRelationFromClass** also takes several parameters. Like the **AddEntityFromClass** method, the first two allow you to uniquely identify the relation we are about to add either with a number or [a string](#). In this example we are just supplying [a string](#). Next is a relation class name. The class "Generic Relation" is a default class available with a new InterAct control. Next, the ID or name of a source and destination entity are needed. Note that we are passing the IDs assigned to the entities created with the **AddEntityFromClass** methods.

Run the program. Press the button **Command1**. A simple diagram of two entities connected by a relation will be created. One entity will contain the text "First Node" and the other will contain the text "Last Node." You can use the mouse and keyboard to move and resize the entities. Stop the program and return to the programming environment.



*InterAct after the **Command1** button is clicked.*

Responding to Events

Invoke the *Code Window* by left double-clicking on InterAct. Choose “ClickEntity” from the **Procedure** listbox.

Add the following code to the InterAct’s ClickEntity event:

```
Dim E1 As Entity
Dim Text As String

If (MouseStatus = IDO_RIGHTCLICK) Then

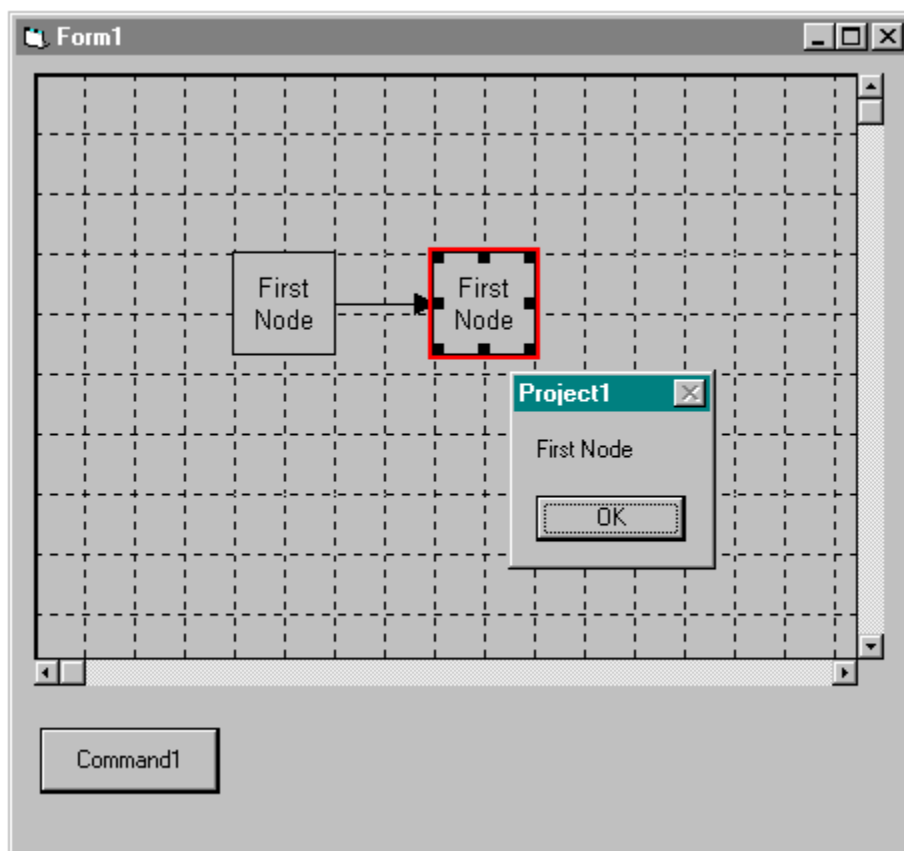
    Rem retrieve the entity which was just clicked
    Set E1 = ido1.GetNotifyEntity

    Rem get the text from the clicked entity
    Text = E1.Text

    Rem display the text
    MsgBox (Text)

End If
```

Run the program now. Press the button **Command1**. Our simple diagram of two entities connected by a relation will be created. Now right click the entity marked “First Node.” The program output will respond with a message with the text “First Node.”



InterAct when the entity “First Node” is clicked.

Conclusion

This concludes the Visual Basic tutorial of InterAct. We have shown just some of the basic features and capabilities of InterAct. When using InterAct in Visual Basic you will use these same three techniques:

- Add InterAct to a specific Visual Basic project or the default Visual Basic project.
- Place InterAct on a form and set properties for InterAct.
- Add code to invoke methods and properties on InterAct to create or alter diagrams.
- Respond to events from InterAct to perform special actions.

Introduction

This tutorial is meant to show how to use the InterAct DLL as a custom control using ProtoGen+. This tutorial can be used by any programmer wishing to use C or C++ and InterAct and does not rely on any capabilities provided solely by ProtoGen+.

This tutorial will show how you can:

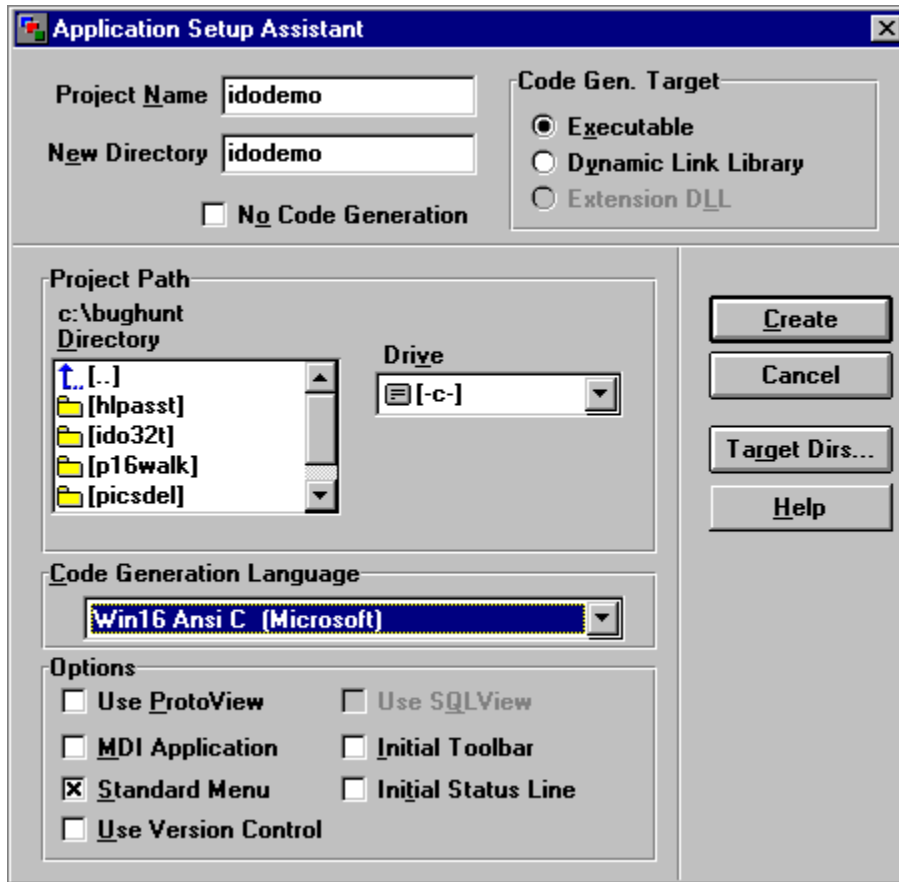
- add InterAct as a component to ViewPaint's control palette
- add InterAct to a dialog and set styles on InterAct
- programmatically add entities and relations to a diagram
- respond to mouse clicks within the InterAct environment

If you want a tutorial regarding how to use InterAct using the mouse and keyboard, see *Chapter 1: A Guided Tour of InterAct*.

Adding InterAct to an Application

Creating a Generic Program

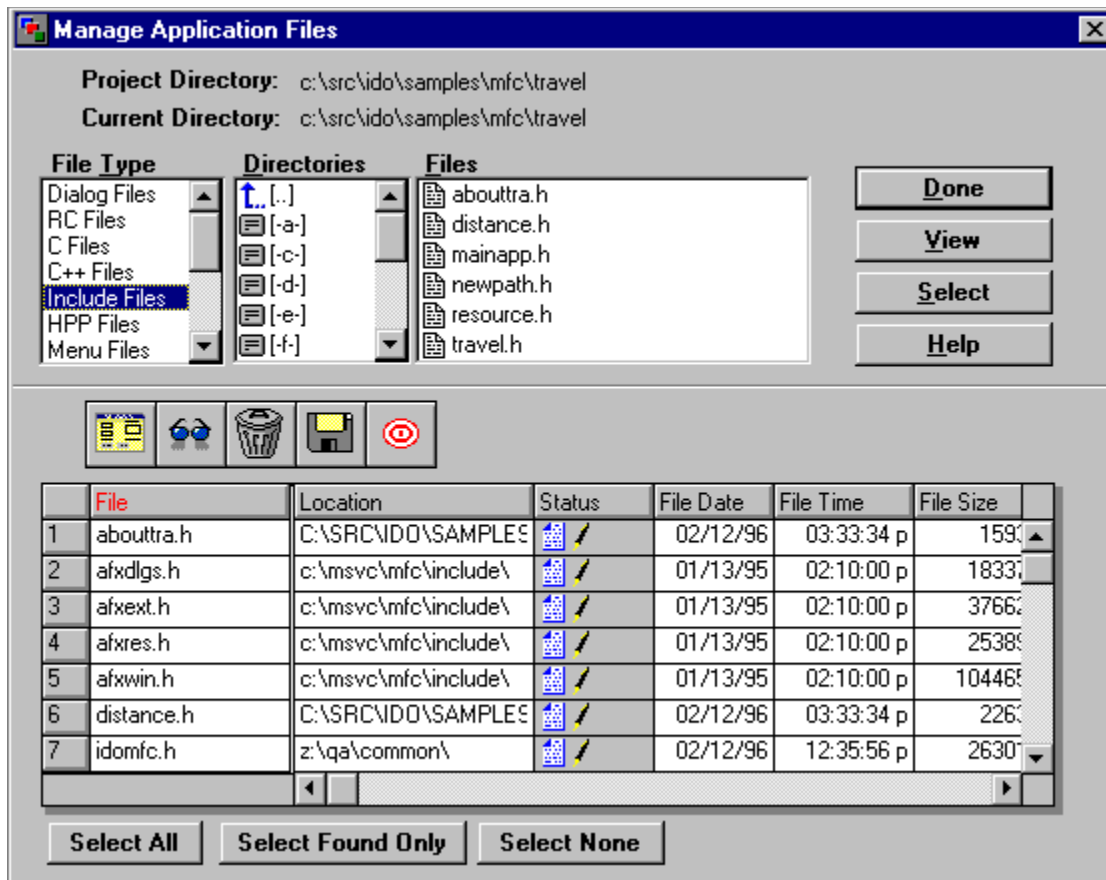
Start a new project by choosing **File | New Project**. In the *Application Setup Assistant*, enter “idodemo” as a **Project Name** and choose a **Directory** to store the project. Next, choose the **Code Generation Language** your computer supports. This tutorial will select “16bit Ansi.” In the **Options** group, make sure only **Standard Menu** is checked. Click **Create** to accept these options.



The ProtoGen+ Application Setup Assistant dialog.

Adding the InterAct header and library Files

When you compile an application which contains InterAct you must include the InterAct’s header and import library. From the PG+ main menu, select **Project | Manage Files**. This takes you to the *Manage Application Files* dialog.



The ProtoGen+ Manage Application Files dialog.

Select the entry “Include Files” in the **File Type** listbox. Navigate to the directory which contains the include file “pvido.h.” Highlight this file and click the **Select** button.

Select the entry “Library Files” in the **File Type** listbox. Navigate to the directory which contains the include file “pvido.lib.” Highlight this file and click the **Select** button.

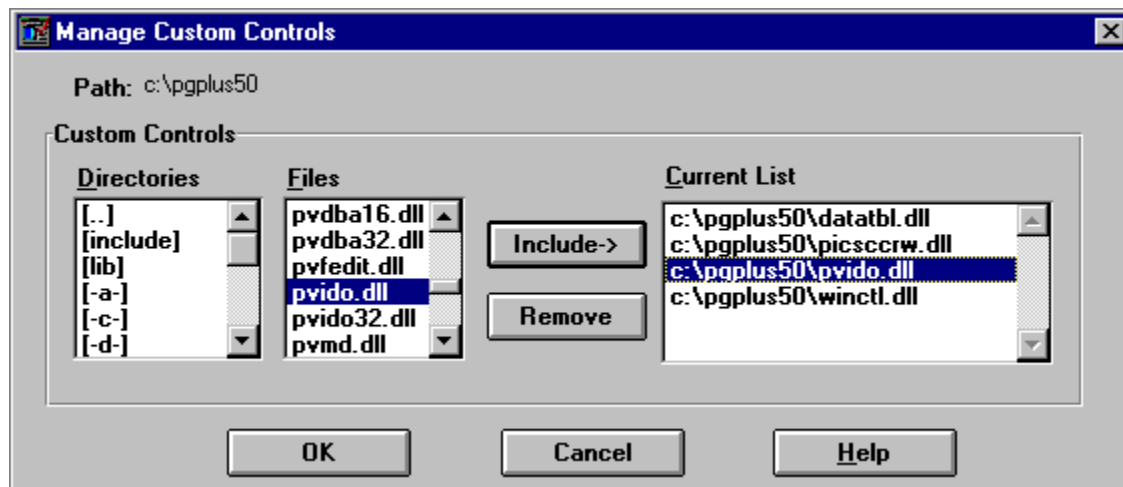
Click **Done**. All the necessary support files for InterAct have now been added.

Note: In future editions of ProtoGen+, the Workbench will automatically detect the IDO and add these support files.

We will now create a main window dialog which will contain InterAct. Right click on the main window. Choose **Select Main Window** from the floating menu. Click the **New Dialog** button in the *Main Window Dialog* screen. Double click the entry “MainWindowDlg” in the *Select New Dialog Resource* list. We are now in ViewPaint.

Installing InterAct in ProtoView's ViewPaint

If you have installed InterAct in ViewPaint prior to this tutorial, you do not need to perform these steps again. You need only install InterAct with ViewPaint once for it to be available for all new projects. You can skip to the step *Adding InterAct to a Dialog*. If you haven't installed InterAct in ViewPaint yet, select the **Add and Remove Custom Controls** menu item from the **Option** menu. This presents the *Manage Custom Controls* dialog.



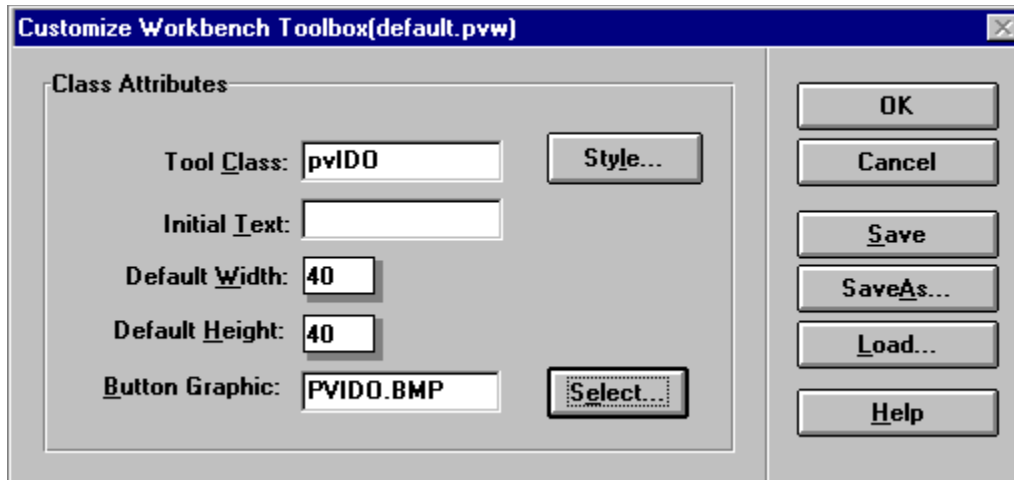
Manage Custom Controls dialog

If you are using the 16-bit ProtoGen+ Workbench, navigate to the directory that contains the PVIDO.DLL file in the **Directories** list box and select the PVIDO.DLL file in the **Files** list box. Click on the **Include** button. The PVIDO.DLL will show up in the **Current List**. Click on the **OK** button to close the dialog. ViewPaint now knows where to find the 16-bit InterAct dynamic link library and "PVIDO" will show up in the list of available controls.

If you are using the 32-bit ProtoGen+ Workbench, navigate to the directory that contains the PVIDO32.DLL file in the **Directories** list box and select the PVIDO32.DLL file in the **Files** list box. Click on the **Include** button. The PVIDO32.DLL will show up in the **Current List**. Click on the **OK** button to close the dialog. ViewPaint now knows where to find the 32-bit InterAct dynamic link library and "PVIDO32" will show up in the list of available controls.

Customizing the Tools Palette:

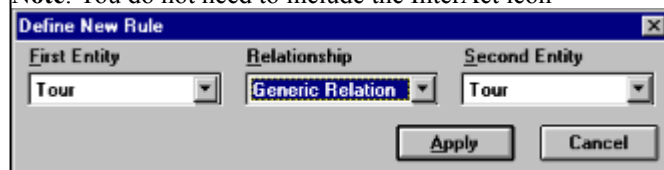
To include the InterAct icon on the ViewPaint tools palette, right-click on an existing or empty control icon. This presents the *Customize Workbench Toolbox* dialog.



Customizing the ViewPaint tools palette.

Depending on whether you are using the 16- or 32-bit ProtoGen+ Workbench, set the **Tool Class** to “PVIDO” or “PVIDO32,” respectively, and set **Button Graphic** to “PVIDO.BMP”. Click the **Save** button to permanently keep the InterAct control on ViewPaint’s tools palette. Click the **OK** button to close the *Customize Workbench Toolbox* dialog.

Note: You do not need to include the InterAct icon

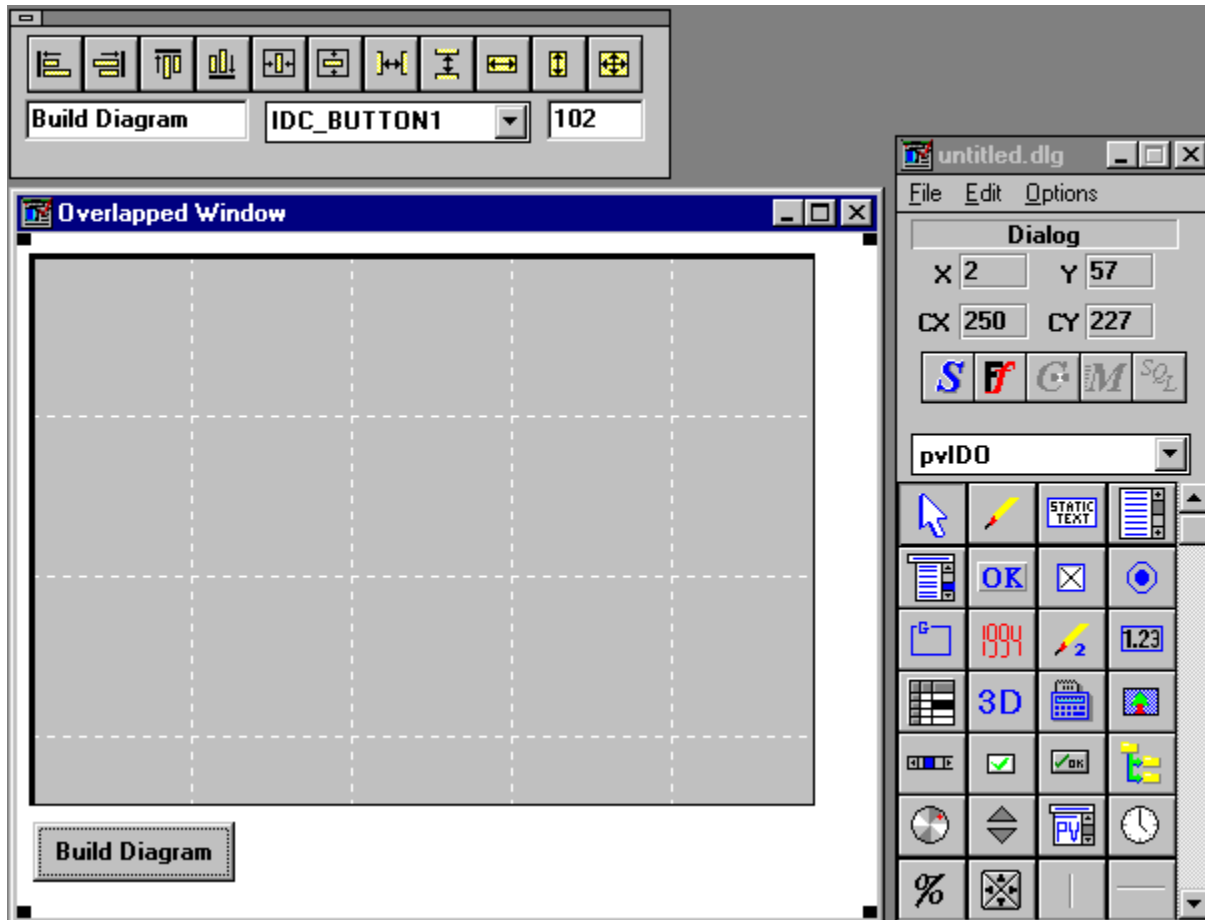


on the tool palette. Once it is installed into ViewPaint, it is always accessible through the tool palette’s **Control List**.

Adding InterAct to a Dialog



Select the InterAct icon, from the *Tools Palette* and draw the InterAct control on the dialog. Resize InterAct so that it takes up most of the dialog.



Designing InterAct in ViewPaint.

When drawn in ViewPaint, InterAct will be a simple rectangle with gridlines drawn at even intervals.

Setting Styles on InterAct

Right-click on InterAct. This takes you to the *InterAct Control Styles* dialog. Here you can apply standard Windows styles to the InterAct control. Make sure the following styles are applied:

- Visible
- Border
- Tabstop
- Horizontal Scroll
- Vertical Scroll


Click **OK** to return to ViewPaint. Add a button to the dialog. By default it will be called IDC_BUTTON1 and have the text “Button”. Right-click the button to display its styles dialog. Change the **Field Text** to “Build Diagram.” Click **OK**.

Select **File | Save and Exit** to save this dialog. Enter “MainView” in the field **Enter Resource Name** and click **OK**. Select the file “idodemo.dlg” as the dialog resource file and click **OK**.

We are now in the *Main Window Dialog* screen. Make sure “MainView” is selected and click **Select**. The main window of our application has now been selected and contains InterAct.

Programming InterAct

Most applications will want to create and manipulate their own diagrams in InterAct. We will demonstrate how to dynamically create a simple diagram in InterAct. First, generate code for our main window dialog

by clicking the **Express Generate** button  on the ProtoGen+ top toolbar.

Right-click on the main window background and select **Main Window Messages** from the floating menu. Highlight “WM_INITDIALOG” in the **Selected Messages** list and click **Edit Code**. We will be brought into the source code at the correct location.

Add code to WM_INITDIALOG:

Add the following code to the dialog initialization routine.

```
// hide the tools palette
idoSetToolsPalette(GetDlgItem(hWnd, IDC_IDO1), FALSE);

// make editable
idoSetEditMode(GetDlgItem(hWnd, IDC_IDO1), TRUE);

// set grid width
idoSetGridWidth(GetDlgItem(hWnd, IDC_IDO1), 25);

// set grid height
idoSetGridHeight(GetDlgItem(hWnd, IDC_IDO1), 20);

// hide InterAct default menus when items are right-clicked
idoSetPopupMenu(GetDlgItem(hWnd, IDC_IDO1),
                 IDOMENU_ENTITY, FALSE);
```

This code disables use of the InterAct’s built-in tools palette and menu for entity objects and sets the width and height of the grid lines in InterAct. The one parameter common to all these functions is a window handle which uniquely identifies InterAct. You can retrieve the window handle by using the Windows API function **GetDlgItem()**. Return to the Workbench.

Adding code to a button:

Right-click on the button and choose **Edit Button Code** from the floating menu to respond to the click event. Add the following code to the button IDC_BUTTON1 notification response:

```
// reset the diagram
idoResetDiagram(GetDlgItem(hWnd, IDC_IDO1));

// add an entity identified as 1
idoAddEntityFromClass(GetDlgItem(hWnd, IDC_IDO1),
                      1, "", "Generic Rectangle", "First Node", 100, 100, 50, 50);

// add an entity identified as 2
idoAddEntityFromClass(GetDlgItem(hWnd, IDC_IDO1),
                      2, "", "Generic Rectangle", "Last Node", 200, 100, 50, 50);

// connect entity 1 to entity2 with a relation "link"
idoAddRelationFromClass(GetDlgItem(hWnd, IDC_IDO1),
                        0, "Link", "Generic Relation", "", 1, "", 2, "");
```


The button **Build Diagram** will clear the diagram by calling InterAct’s **ResetDiagram** method. The only parameter is a window handle which uniquely identifies InterAct. You can retrieve the window handle by using the Windows API function **GetDlgItem()**. Next we add two entities and a relation connecting these

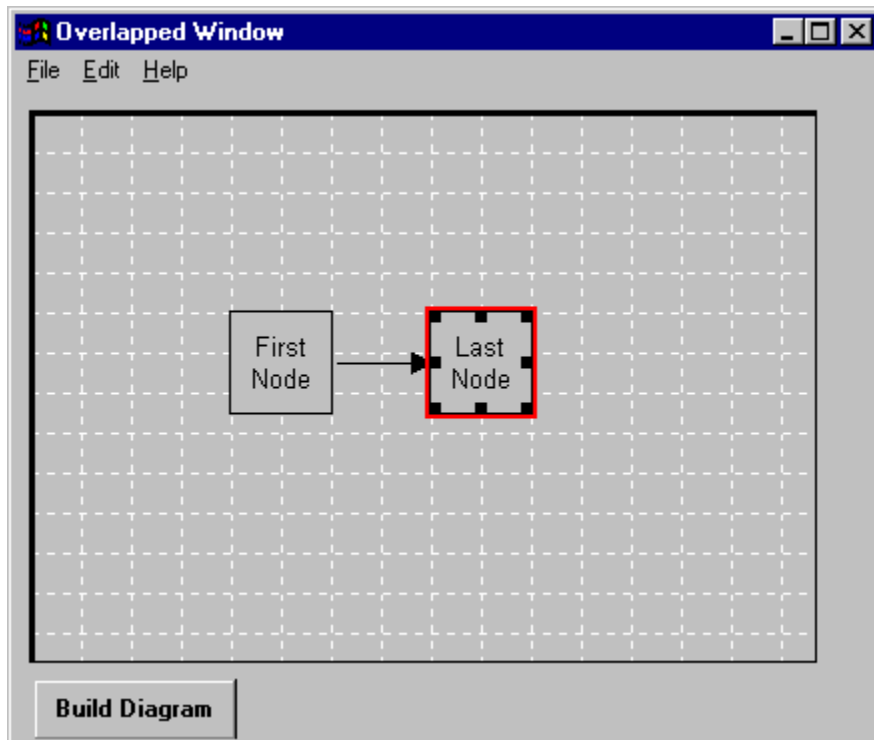
entities

The method **AddEntityFromClass** takes several parameters. The first is the window handle of InterAct. The next two allow you to uniquely identify the entity we are about to add either with a number or [a string](#). In this example we are just supplying a number. Next is an entity class name. The class “Generic Rectangle” is a default class available with a new InterAct control. Classes are simply an easy way to specify how an entity will look and behave without having to supply all these details when you create the entity. Next, text which will be displayed by the entity is passed. Finally, the last four digits are the upper left corner and width and height of the new entity.

The method **AddRelationFromClass** also takes several parameters. Like the **AddEntityFromClass** method, the first is the window handle of InterAct and the next two allow you to uniquely identify the relation we are about to add either with a number or [a string](#). In this example we are just supplying [a string](#). Next is a relation class name. The class “Generic Relation” is a default class available with a new InterAct control. Next is text to be displayed with the relation. We are not supplying any. Next, the ID or name of a source and destination entity are needed. Note that we are passing the IDs assigned to the entities created with the **AddEntityFromClass** methods.

Run the application:

Compile and execute the program by clicking the **Generate and Compile Source** button  on the PG+ top toolbar. Press the button **Build Diagram**. A simple diagram of two entities connected by a relation will be created. One entity will contain the text “First Node” and the other will contain the text “Last Node.” You can use the mouse and keyboard to move and resize the entities. Exit the program and return to the ProtoGen+ programming environment.



*InterAct after the **Add** button is clicked.*

Responding to Events

InterAct will send notification of key events as both notifications and messages to its parent dialog. We will respond to the message sent when an entity is clicked. Right-click on the background of the main window. Select **Main Window Messages** from the floating menu. Select **Message Category | Custom Messages**. Select the message “IDOM_N_CLICK_ENTITY” from the **Custom Messages** list. This message will now appear in the **Selected Messages** list. Highlight the message “IDOM_N_CLICK_ENTITY” in the **Selected Messages** list and click the button **Edit Code**. We will be brought into the source code at the correct location.

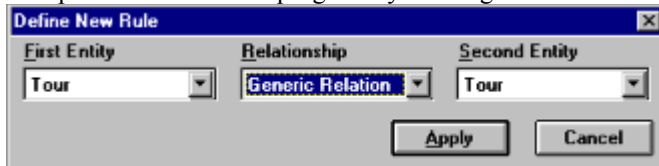
Note: If this message does not appear in your list, quit your PG+ Workbench and copy the file MSG.TAB from the InterAct directory to your PG+ directory and restart PG+.

```
case IDOM_N_CLICK_ENTITY :  
  
    //Regen_IDOM_N_CLICK_ENTITY  
  
    {  
        ENTITY entity;  
  
        switch(IParam)  
        {  
            case NOTIFY_CLICK_RIGHT :  
  
                // Get Clicked Entity  
                // if successful, print the entities text to the screen  
  
                if(idoGetNotifyEntity(GetDlgItem(hWnd, wParam), &entity))  
                    MessageBox(hWnd, idoEntityGetText(&entity),  
                                "Entity Text", MB_OK);  
  
                break;  
            }  
        }  
  
        //Regen_IDOM_N_CLICK_ENTITY  
        break;  
    }
```

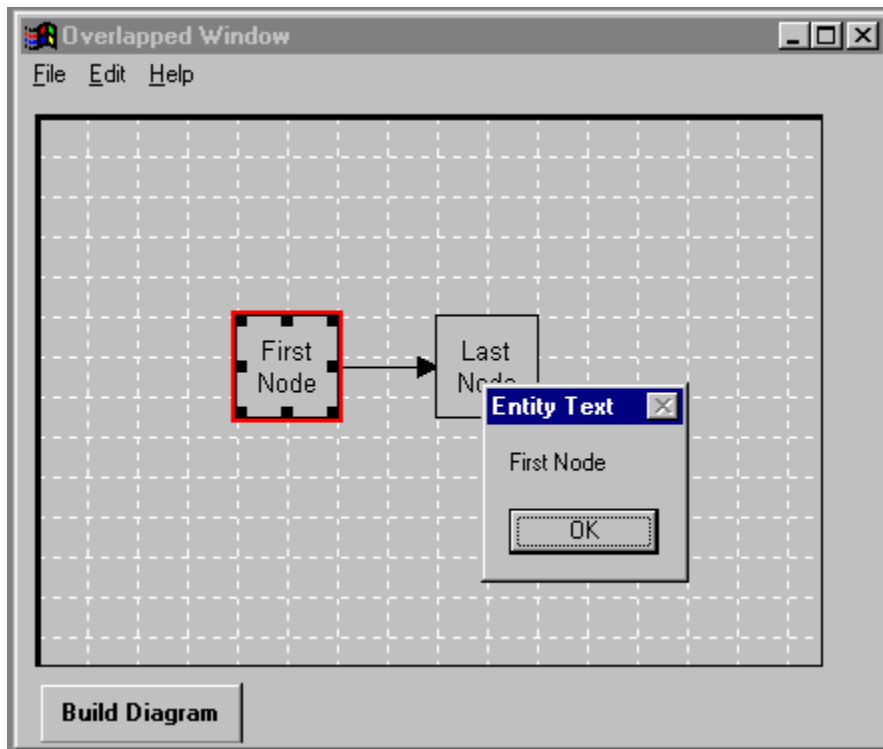
This code allows us to respond to the mouse click event on an entity. For further information, we evaluate the contents of the IParam, responding only when the entity is right clicked and ignoring all other clicks on entities. In response to this event we wish to display a message box with the text of the clicked entity. For this we call **idoGetNotifyEntity**. If this function succeeds, information about the entity will be stored in the variable **entity**. We can then use **idoEntityGetText** to retrieve the text associated with that entity.

Run the application:

Compile and execute the program by clicking the **Generate and Compile Source** button



on the PG+ top toolbar. Press the button **Build Diagram**. Our simple diagram will be created. Right-click on the background of one of the entities. A message box will appear with the text of that entity.



InterAct when an entity is right clicked.

Conclusion

This concludes the ProtoGen+ tutorial of InterAct. We have shown just some of the basic features and capabilities of InterAct. When using InterAct in ProtoGen+ you will use these same four techniques:

- Add the InterAct header and library files to your project.
- Install InterAct as a custom control in ViewPaint.
- Place InterAct on a dialog and set styles for the it.
- Add code to call functions which invoke methods and properties of InterAct to create or alter diagrams.
- Respond to events from InterAct through notifications and messages to perform special actions.

Introduction

InterAct is available as a 16 and 32-bit DLL or a 32-bit ActiveX. In order to keep the manual manageable and the programming interface consistent, the following reference section can be used by both the DLL and ActiveX. Where appropriate, if differences exist, notes will describe the difference between the DLL and ActiveX.

InterAct is manipulated from a program through invoking methods or setting properties. In addition, InterAct can fire events to its parent, notifying the parent of key events.

A property is usually a value that can be retrieved or set. An example of a property is whether to display grid lines in a diagram. Properties will usually be accessed with Get/Set functions. For the ActiveX, some properties can also be set when designing InterAct in a programming environment, such as Microsoft's AppStudio, Visual Basic or Borland's Delphi.

A method is a function that tells InterAct to perform an action. For example, you might call a method telling InterAct to save the current diagram to a file, passing a valid filename and optional path to InterAct. Methods cannot be invoked when designing InterAct in a programming environment, such as Microsoft's AppStudio, Visual Basic or Borland's Delphi, and can only be called at runtime.

InterAct has many properties that affect its appearance and many methods to add, manipulate and remove objects from a diagram.

Identifying Objects

Every object in the diagram needs to be uniquely identified to allow actions to be taken on specific items. There are two ways you can identify an object - by a name or an ID. IDs are stored as a C data type *long*, and can have a value between 0 and 32000. Names are a character string of up to 99 characters. An object can be identified by both a name and an ID.

When adding a new entity or relation programmatically, you can assign either a name or an ID to the new item. In the first sample below, a new entity is given the ID of 100 and the name "Object A." You do not have to supply both a name and an ID. If you supply neither, InterAct will provide a default name and ID.

```
// We have supplied both a name and an ID.
IDO.AddEntityFromClass(100, "Object A", "Rectangle", "", 10, 10, 50, 50);

// We have supplied only an ID. InterAct will supply a default name.
IDO.AddEntityFromClass(100, "", "Rectangle", "", 10, 10, 50, 50);

// We have supplied only a name. InterAct will supply a default ID.
IDO.AddEntityFromClass(0, "Object A", "Rectangle", "", 10, 10, 50, 50);

// We have supplied neither a name or an ID. InterAct will supply a default value for both.
IDO.AddEntityFromClass(0, "", "Rectangle", "", 10, 10, 50, 50);
```

Add objects using the ActiveX methods.

```
HWND hIDO = GetDlgItem(hWnd, IDC_IDO1);

// We have supplied both a name and an ID.
idoAddEntityFromClass(hIDO, 100, "Object A", "Rectangle", "", 10, 10, 50, 50);

// We have supplied only an ID. InterAct will supply a default name.
idoAddEntityFromClass(hIDO, 100, "", "Rectangle", "", 10, 10, 50, 50);

// We have supplied only a name. InterAct will supply a default ID.
idoAddEntityFromClass(hIDO, 0, "Object A", "Rectangle", "", 10, 10, 50, 50);

// We have supplied neither a name or an ID. InterAct will supply a default value for both.
idoAddEntityFromClass(hIDO, 0, "", "Rectangle", "", 10, 10, 50, 50);
```

Adding objects using the ANSI C functions.

InterAct Architecture

InterAct is a combination of three distinct objects - the InterAct window itself, and a collection of entities and relations. InterAct has a list of properties and methods which can be invoked, and each entity and relation also has a list of properties and methods which can be invoked. You will always have access to the InterAct object and its list of properties and methods. You must request access to one of the entities or relations from InterAct before interfacing with these objects. For example, if you want to place some text on an entity, you must first get a reference to that entity.

```
ENTITY entity;

if(idoGetEntity(hWndIdo, 25, NULL, &entity))
    idoEntitySetText(&entity, "New Text");
if(idoGetEntity(hWndIdo, 0, "Object A", &entity)
    idoEntitySetText(&entity, "New Text");
```

DLL example.

```
IDO.GetEntity(25, NULL).SetText("New Text");
IDO.GetEntity(0, "Object A").SetText("New Text");
```

ActiveX example.

In the DLL example code, a method and a property must be invoked. The method **idoGetEntity** takes four parameters:

- 1) A window handle to identify the InterAct object.
- 2) An ID to uniquely identify an entity. Pass 0 (zero) if you are going to supply a unique name.
- 3) [A string](#) to uniquely identify an entity. Pass NULL if you are supplying a unique ID.
- 4) The address of [an entity object](#) which will receive information about the entity we are selecting.

The InterAct DLL will return a reference to this entity. Next, the property **idoEntitySetText** is invoked. This property takes two parameters:

- 1) The address of an entity reference object.
- 2) [A string](#) to assign to the identified entity.

For the ActiveX, you are really calling a method and property at once. **IDO.GetEntity** is a method which returns [an entity object](#). This method takes two parameters:

- 1) An ID to uniquely identify an entity. Pass 0 (zero) if you are going to supply a unique name.
- 2) [A string](#) to uniquely identify an entity. Pass NULL if you are supplying a unique ID.

You can now set a property on the entity object. The Entity object exposes the property **Text**. **Entity.SetText** takes [a string](#) to assign to the entity. You could also perform this task on the ActiveX in two steps:

```
EntityObject = IDO.GetEntity(25, NULL);
EntityObject.SetText("New Text");
```

Using Fonts

InterAct allows you to set fonts on entities and relations. This font is used to display text for that entity or relation. For both the DLL and ActiveX, [a font object](#) is used to describe a font. For the DLL, a structure [IDO_FONT](#) is used. [IDO_FONT](#) has the following members:

char	szFontName
int	cSize
BOOL	bBold
BOOL	bItalic
BOOL	bUnderline
BOOL	bStrikeout

With these members you can describe a font to be used by an entity or relation.

For the ActiveX an OLE Font object is used. Each entity or relation object has a font property which you can get or set. The OLE font object has the following properties:

STRING	Name
CURRENCY	Size
BOOL	Bold
BOOL	Italic
BOOL	Underline
BOOL	Strikethrough
short	Weight
short	Charset

With these members you can describe a font to be used by an entity or relation.

Using Pictures

InterAct allows you to display pictures within entities or associate a picture with a class displayed on the tools palette. These pictures must reside on disk as a BMP file. InterAct will store a path to the file and attempt to load the picture using that path. If this fails, InterAct will attempt to open the file anywhere on disk where it can search. The directories searched will include the current directory, the Windows system directory, and any directory in the PATH environment variable.

Programming with the DLL

Support Components

When programming with the DLL, you need to include the header file PVIDEO.H and link with the import library PVIDEO.LIB if you are using the 16-bit DLL. If you are using the 32-bit DLL you must link with PVIDEO32.LIB for Visual C++, PVIDEO32B.LIB for Borland C++, or PVIDEO32S.LIB for the Symantec compiler.

Function API

Normally a custom control in a DLL like InterAct exposes a single Window Procedure and a list of vendor-defined messages which tell the control to perform a specific action. However, a message-based architecture is limiting, since a message can allow only two parameters of information to be passed when the message is sent. This leads to situations where structures of data are passed when a message is called and not just single values. In order to get around this limitation and avoid the need for cumbersome structures, InterAct also exports a wide range of functions.

This set of functions exposed by the DLL can be called by an application. Just like sending a message, many of these functions will require passing a window handle as the first parameter to identify InterAct you wish to perform the action on. If an invalid window handle is passed, the invoked function will not have any effect.

You can get a window handle from an existing InterAct control in a dialog by using **GetDlgItem()**, or saving the result of **CreateWindow()** if you dynamically create the InterAct control. See your Windows API help file for information about these two functions.

```
HWND hIDO;  
  
// get the window handle to InterAct  
hIDO = GetDlgItem(hWnd, IDC_IDO);  
  
// set properties on InterAct, using the window handle as a parameter  
idoSetPaletteVisible(hIDO, FALSE);  
idoSetGridHeight(hIDO, 25);  
idoSetGridWidth(hIDO, 25);
```

ANSI C example of accessing [an InterAct object](#) using a Window handle.

When referring to entities or relations, a structure is used instead of a window handle. These are declared as types ENTITY and RELATION, respectively. When setting properties or invoking methods on these objects, you do not need a window handle to InterAct. You need only pass a pointer to these structures as the first parameter.


```

HWND hIDO;
ENTITY entity;

// get the window handle of InterAct
hIDO = GetDlgItem(hWnd, IDC_IDO);

// retrieve a reference to the entity with the ID of one
// return that reference in the variable 'entity'
idoGetEntity(hIDO, 1, NULL, &entity);

// set the text on the entity
// we need pass only the address of the entity object
idoEntitySetText(&entity, "Text.");

```

ANSI C example of using an ENTITY structure.

If you really wish to use the messages and not the functions exported from the DLL, you should look at the PVIDO.H header file which defines all the messages and look at CIFACE.C, a source file from the InterAct DLL which shows how the exposed functions are packed into corresponding messages.

Some DLL functions return a [long](#) pointer to a [string](#) (LPSTR). This string is a *temporary* pointer returned by the DLL. You must immediately copy the contents of this string into your own allocated character array. You must *never* copy anything into this internal string array or save this pointer.

Using the DLL in C++

You can use the message and function API in C++ just as you would in ANSI C. The InterAct DLL also supports the MFC and OWL class libraries.

For MFC, include the header file IDOMFC.H. Four classes are defined: CIDO, CEntity, CRelation and CIDORule. For OWL, include the header file IDOOWL.H. Four classes are defined: TIDO, TEntity, TRelation and TIDORule. All four classes are defined inline so you do not need to link with any libraries.

All the standard function APIs are available in the C++ wrappers. Where functions exist to retrieve an entity or relation object, these functions have been overloaded to accept or return CEntity and CRelation objects, or TEntity and TRelation objects. Some functions which accept long pointers to strings have been overloaded to accept CString objects.

For functions which require a hWnd to InterAct, drop the prefix 'ido' from the function and drop the required window handle parameter. For entity or relation functions, drop the prefix 'idoEntity' and 'idoRelation' from the function and drop the required pointer to a ENTITY or RELATION object. See the sample below:

```

ENTITY entity;
HWND hIDO;

// get a window handle to InterAct
hIDO = GetDlgItem(hWnd, IDC_IDO);

// get a reference to an entity
idoGetEntity(hIDO, 1, NULL, &entity);

// set the text on the entity
idoEntitySetText(&entity, "Text");

```

ANSI C example using the Message API.

Now the same sample using the MFC wrappers.

```
CEntity * pEntity;  
CIDO *pIDO;  
  
// create a new instance of an entity C++ object  
pEntity = new CEntity();  
  
// get a reference to the InterAct object  
pIDO = (CIDO *)GetDlgItem(IDC_IDO);  
  
// retrieve a reference to an entity object from InterAct  
pIDO->GetEntity(1, NULL, pEntity);  
  
// set the text on the entity object  
pEntity->SetText("Text");  
  
// delete the entity C++ object  
delete pEntity;
```

MFC example using the C++ classes.

What do I need to ship with my finished Application?

If you are developing a 16-bit application, you need to ship PVIDEO.DLL. If you are planning to display the property pages for InterAct, entities or relation, you also need to ship PGHT.DLL.

If you are developing a 32-bit application, you need to ship PVIDEO32.DLL. If you are planning to display the property pages for InterAct, entities or relation, you also need to ship PGHT32.DLL.

Programming with the OCX

Support Components

When using the ActiveX, the programming environment, such as Microsoft's Developer Studio, Visual Basic or Borland's Delphi, will be able to query the ActiveX and create a programming interface for the ActiveX automatically. Visual C++ will generate a class header and source file, while Delphi will generate a Pascal unit file, for each object defined in the InterAct ActiveX. These files will have functions for each property and method defined for each object.

Visual Basic is able to read and store all the constants defined by the InterAct control. Other environments, such as Visual C++ and Delphi, do not. For Visual C++ InterAct provides an additional header file called IDODEF.H with these defines. For Delphi, IDODEF.PAS is included. Be sure to include these header files to define these constants.

Using the ActiveX in C++

At the time of printing, Visual C++ does not declare constants used by InterAct in a header file. These constants are used throughout InterAct's properties and methods and must be declared in order to be used. InterAct therefore supplies a header file called IDODEF.H. This header file contains the declarations for these constants and must be included in the C++ application.

Using the ActiveX in Delphi

At the time of printing, Delphi does not declare constants used by InterAct in a pascal unit file. These constants are used throughout InterAct's properties and methods and must be declared in order to be used. InterAct therefore supplies a header file called IDODEF.PAS. This header file contains the declarations for these constants and must be included in the C++ application.

What do I need to ship with my finished Application?

If you are developing a 16-bit application, you need to ship PVIDO.OCX. If you are planning to display the property pages for InterAct, entities or relation, you also need to ship PGHT.DLL.

If you are developing a 32-bit application, you need to ship PVIDO32.OCX. If you are planning to display the property pages for InterAct, entities or relation, you also need to ship PGHT32.DLL.

Introduction

InterAct will fire events to its parent notifying it of key events. These events allow you to provide sophisticated responses to InterAct. Events which a program is notified of include left and right mouse clicks on entities, relations, or InterAct background itself; when an entity or relation is added or removed to a diagram; when a diagram or palette is loaded or saved. Often you will receive two notifications, one before and one after the event. In some instances you can even stop an action by responding to the notification fired before the event.

Responding to Events from the DLL

For some notifications InterAct follows the Windows standard of sending the notification as a WM_COMMAND message.

However, many notifications also need to convey some information to the program about that notification. For example, if a diagram is about to be saved, it may be important for the program to know the path and file name the diagram is about to be saved to. Using the traditional WM_COMMAND notification is not suitable since no other information can be passed with the notification. Therefore, many notifications will be sent as messages.

Note: These messages are in the range of WM_USER + 300 - WM_USER + 399.

The wParam will contain the InterAct's control ID. The lParam will contain additional information or a pointer to additional information.

In some cases, where more information is needed, InterAct can be queried for that information. For example, when you receive the event that an entity has been clicked, the notification will specify which mouse button was used to click on the control, but the identity of the clicked entity is unknown. The InterAct DLL will store the clicked entity's ID and Name, which you can then access using the function **idoGetNotifyEntity** and **idoGetNotifyRelation**.

Some events actually allow you to affect the behavior of the event. For example, if the application receives the notification that an entity is about to be deleted it is possible to tell InterAct to abort the event and not have the entity deleted. This can be done by calling **idoCancelAction**.

Standard Events

InterAct will send the following notifications using the traditional WM_COMMAND mechanism.

IDON_SELCHANGE

The user has added or removed the selection from an entity or relation.

IDON_SETFOCUS

Input focus has been set to InterAct.

IDON_KILLFOCUS

InterAct has lost input focus.

Diagram Events

The application containing InterAct will receive two notifications when a diagram is saved or loaded - one before the event and one after the event. It is possible to abort the file load or file save when responding to the notification before the event. The application will also receive notifications before and after a diagram is reset. It is possible to abort the reset command when responding to the notification sent before the event.

BeforeDiagramSaved

This event is fired before a diagram is saved.

This event can be used to cancel InterAct from saving a diagram file.

ANSI C

IDOM_N_BEFOREDIAGRAMSAVED

IParam

Long pointer to [a string](#) (LPSTR) which contains the path to the diagram file (.IDO) about to be saved.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDiagramSaved(*FileName* As String, *Status* as Long)

Parameters

[A string](#) which contains the path to the diagram file (.IDO) about to be saved.

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDiagramSaved(LPCSTR * *FileName*, long FAR * *Status*);

Parameters

[A long](#) pointer to [a string](#) which contains the path to the diagram file (.IDO) about to be saved.

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

DiagramSaved

This event is fired after a diagram file is saved.

ANSI C

IDOM_N_DIAGRAMSAVED

IParam

Long pointer to [a string](#) (LPSTR) which contains the path to the diagram file (.IDO) just saved.

Visual Basic

DiagramSaved(*FileName* As String)

Parameters

[A string](#) which contains the path to the diagram file (.IDO) just saved.

C++

DiagramSaved(LPCSTR * *FileName*);

Parameters

[A long](#) pointer to [a string](#) which contains the path to the diagram file (.IDO) just saved.

BeforeDiagramLoaded

This event is fired before a diagram file (.IDO) is loaded.

This event can be used to cancel InterAct from loading a diagram file.

ANSI C

IDOM_N_BEFOREDIAGRAMLOADED

IParam

Long pointer to [a string](#) (LPSTR) which contains the path to the diagram file (.IDO) about to be loaded.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDiagramLoaded(*FileName* As String, *Status* as Long)

Parameters

[A string](#) which contains the path to the diagram file (.IDO) about to be loaded.

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDiagramLoaded(LPCSTR * *FileName*, long FAR * *Status*);

Parameters

A [long](#) pointer to a [string](#) which contains the path to the diagram file (.IDO) about to be loaded.

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

DiagramLoaded

This event is fired after InterAct has loaded a diagram file.

ANSI C

IDOM_N_DIAGRAMLOADED

IParm

Long pointer to a [string](#) (LPSTR) which contains the path to the diagram file (.IDO) just loaded.

Visual Basic

DiagramLoaded(*FileName* As String)

Parameters

A [string](#) which contains the path to the diagram file (.IDO) just loaded.

C++

DiagramLoaded(LPCSTR * *FileName*);

Parameters

A [long](#) pointer to a [string](#) which contains the path to the diagram file (.IDO) just loaded.

BeforeDiagramReset

This event is fired before InterAct is about to reset a diagram. All relations and entities within a diagram will be deleted on reset.

This event can be used to cancel InterAct from resetting the diagram.

ANSI C

IDOM_N_BEFOREDIAGRAMRESET

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDiagramReset(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDiagramReset(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

DiagramReset

This event is fired after InterAct has reset the diagram.

ANSI C

IDOM_N_DIAGRAMRESET

IParam

No meaningful value. Reserved for future.

Visual Basic

DiagramReset()

Parameters

None.

C++

DiagramReset();

Parameters

None.

Palette Events

The application containing InterAct will receive two notifications when a palette is saved or loaded - one before the event and one after the event. It is possible to abort the file load or file save when responding to the notification before the event. The application will also receive notifications before and after a palette is reset. It is possible to abort the reset command when responding to the notification sent before the event.

BeforePaletteSaved

This event is fired before a palette is saved.

This event can be used to cancel InterAct from saving a palette file.

ANSI C

IDOM_N_BEFOREPALETTESAVED

IParam

Long pointer to [a string](#) (LPSTR) which contains the path to the palette file (.PLT) about to be saved.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforePaletteSaved(*FileName* As String, *Status* as Long)

Parameters

[A string](#) which contains the path to the palette file (.PLT) about to be saved.

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforePaletteSaved(LPCSTR * *FileName*, long FAR * *Status*);

Parameters

[A long](#) pointer to [a string](#) which contains the path to the palette file (.PLT) about to be saved.

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

PaletteSaved

This event is fired after a palette file is saved.

ANSI C

IDOM_N_PALETTEAVED

IParam

Long pointer to [a string](#) (LPSTR) which contains the path to the palette file (.IDO) just saved.

Visual Basic

PaletteSaved(*FileName* As String)

Parameters

[A string](#) which contains the path to the palette file (.PLT) just saved.

C++

PaletteSaved(LPCSTR * *FileName*);

Parameters

[A long](#) pointer to [a string](#) which contains the path to the palette file (.PLT) just saved.

BeforePaletteLoaded

This event is fired before a palette file (.PLT) is loaded.

This event can be used to cancel InterAct from loading a palette file.

ANSI C

IDOM_N_BEFOREPALETTELOADED

IParam

Long pointer to [a string](#) (LPSTR) which contains the path to the palette file (.PLT) about to be loaded.

How to Cancel

idoCancelAction(HWND *hIDO*);

The palette will not be loaded.

Visual Basic

BeforePaletteLoaded(*FileName* As String, *Status* as Long)

Parameters

[A string](#) which contains the path to the palette file (.PLT) about

to be loaded.

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

The palette will not be loaded.

C++

BeforePaletteLoaded(LPCSTR * *FileName*, long FAR * *Status*);

Parameters

[A long](#) pointer to [a string](#) which contains the path to the palette file (.PLT) about to be loaded.

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

The palette will not be loaded.

PaletteLoaded

This event is fired after InterAct has loaded a palette file.

ANSI C

IDOM_N_PALETTELOADED

lParam

Long pointer to [a string](#) (LPSTR) which contains the path to the palette file (.PLT) just loaded.

Can be canceled:

No.

Visual Basic

PaletteLoaded(*FileName* As String)

Parameters

[A string](#) which contains the path to the palette file (.PLT) just loaded.

C++

PaletteLoaded(LPCSTR * *FileName*);

Parameters

[A long](#) pointer to [a string](#) which contains the path to the palette file (.PLT) just loaded.

BeforePaletteReset

This event is fired before the palette is reset. Resetting a palette will remove all the entity and relation classes in the diagram.

This event can be used to cancel the palette reset operation.

ANSI C

IDOM_N_BEFOREPALETTERESET

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforePaletteReset(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforePaletteReset(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

PaletteReset

This event is fired after the palette has been reset.

ANSI C

IDOM_N_PALETTERESET

IParam

No meaningful value. Reserved for future.

Visual Basic

PaletteReset()

Parameters

None.

C++

```
PaletteReset();
```

Parameters

None.

Tools Palette Events

The application containing InterAct will receive two notifications when events on the tools palette occur. One is when the help button on the tools palette is pressed. By default, InterAct will display the help file which is installed with InterAct. By responding to this notification, you can suppress this help file from being displayed and display your own help file. A second event occurs if the **Options** button is pressed on the InterAct tools palette. By default, InterAct will display a popup menu with some common options to alter the diagram. By responding to this notification you can prevent InterAct from displaying its popup menu. It is suggested you display your own popup menu with your own menu items. This notification passes the coordinates to be used when displaying your popup menu.

BeforeDisplayHelp

This event is fired when the user clicks the help button on InterAct's built-in tools palette. InterAct will by default display its help file.

This event can be used to cancel the display of the InterAct help file.

If you cancel the event InterAct will not display its help file. It is assumed the container application will display a different help file using the Windows API **WinHelp()**.

ANSI C

```
IDOM_N_BEFOREDISPLAYHELP
```

iParam

No meaningful value. Reserved for future.

How to Cancel

```
idoCancelAction(HWND hIDO);
```

Visual Basic

```
BeforeDisplayHelp(Status As Long)
```

Parameters

A status code so you can return information to InterAct.

How to Cancel

```
Set Status = IDO_CANCELACTION
```

C++

BeforeDisplayHelp(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

BeforeDisplayOptionsMenu

This event is fired when the user clicks the **Options** button on the tools palette. By default InterAct displays a menu of options to choose from.

This event can be used to cancel the display of the menu by InterAct.

If you cancel this event you should display your own options menu or dialog. The x coordinate and y coordinate are passed with this event to pass to the Windows API

TrackPopupMenu() to have a container specified popup menu appear at the correct location.

ANSI C

IDOM_N_BEFOREDISPLAYOPTIONSMENU

IParam

A pack of two integers containing the coordinates of the tools palette. Use HIWORD to get the x coordinate and LOWORD to get the y coordinate to pass to the Windows API

TrackPopupMenu() to have a container specified popup menu appear at the correct location.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDisplayOptionsMenu(*X* As Short, *Y* As Short, *Status* as Long)

Parameters

The x coordinate for a container specified menu.

The y coordinate for a container specified menu.

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDisplayOptionsMenu(short *X*, short *Y*, long FAR *

Status);

Parameters

The x coordinate for a container specified menu.

The y coordinate for a container specified menu.

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

Dialog Events

The application containing InterAct will receive several notifications when InterAct is about to display an internal dialog. This gives the container application an opportunity to suppress the display of these dialogs and substitute their own dialogs or simply not have any dialog appear on these events. Whenever InterAct is about to display its *Manage Classes* dialog or its *Manage Rules* dialog, an event is sent. By default, the *Manage Classes* dialog has a button which displays the *Manage Rules* dialog. A special event is sent when the *Manage Classes* dialog is displayed. By responding to this event it is possible to prevent the display of the button to display the *Manage Rules* dialog on the *Manage Classes* dialog. This is essential if a container application wants to allow users to take advantage of the *Manage Classes* dialog but not allow users access to the *Manage Rules* dialog.

InterAct will also notify the container if InterAct's Tools Palette is about to be hidden or shown. This is useful primarily for container applications which want to display their own tools palette for InterAct.

BeforeDisplayToolsPalette

This event is fired when InterAct is about to display its built-in tools palette.

This event can be used to cancel the display of the tools palette by InterAct.

If you cancel this event InterAct will not display its Tools Palette. It is assumed the container application will display its own tools palette.

ANSI C

IDOM_N_BEFOREDISPLAYTOOLSPALETTE

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDisplayToolsPalette(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDisplayToolsPalette(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

BeforeHideToolsPalette

This event is fired when InterAct is about to hide its built-in tools palette.

This event can be used to cancel the hiding of the tools palette by InterAct.

If you cancel this event InterAct will not hide its Tools Palette. This mainly used to override a click of the system menu on InterAct's tools palette.

ANSI C

IDOM_N_BEFOREHIDETOOLSPALETTE

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeHideToolsPalette(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeHideToolsPalette(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

BeforeDisplayManageClasses

This event is fired when InterAct is about to display its *Manage Classes* dialog.

This event can be used to cancel the display of the *Manage Classes* dialog by InterAct.

If you cancel this event InterAct will not show its *Manage Classes* dialog. It is assumed the container application will display its own dialog for managing classes in InterAct.

ANSI C

IDOM_N_BEFOREDISPLAYMANAGECLASSES

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDisplayManageClasses(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDisplayManageClasses(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

BeforeDisplayManageRules

This event is fired when InterAct is about to display its *Manage Rules* dialog.

This event can be used to cancel the display of the *Manage Rules* dialog by InterAct.

If you cancel this event InterAct will not show its *Manage Rules* dialog. It is assumed the container application will display its own dialog for managing rules in InterAct.

ANSI C

IDOM_N_BEFOREDISPLAYMANAGERULES

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDisplayManageRules(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDisplayManageRules(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

BeforeDisplayManageRulesButton

This event is fired when InterAct is about to display its *Manage Classes* dialog. The *Manage Classes* dialog has a button **Manage Rules** which will display the *Manage Rules* dialog. Some container applications will not want this button displayed if they do not want the user access to the *Manage Rules* dialog.

This event can be used to cancel the display of the **Manage Rules** button by InterAct.

If you cancel this event InterAct will not show the **Manage Rules** button on the *Manage Classes* dialog. This is needed if the container application wants to limit the users access to the *Manage Rules* dialog but still allow users access to the *Manage Classes* dialog.

ANSI C

IDOM_N_BEFOREDISPLAYMANAGERULESBUTTON

IParam

No meaningful value. Reserved for future.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

BeforeDisplayManageRulesButton(*Status* as Long)

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set *Status* = IDO_CANCELACTION

C++

BeforeDisplayManageRulesButton(long FAR * *Status*);

Parameters

A status code so you can return information to InterAct.

How to Cancel

Set **Status* = IDO_CANCELACTION

Mouse Events

InterAct will notify the container application when the mouse is clicked on an entity, relation or the InterAct background. The container application will be notified of a click or double-click with the left or right mouse button. InterAct has built-in behavior for the mouse right click and left mouse double click. InterAct will either display a popup menu or allow the user to edit the items text, respectively. These default actions can be overridden.

ClickEntity

This event is fired when the mouse is clicked or double-clicked on an entity.

ANSI C

IDOM_N_CLICK_ENTITY

IParam

A status code so you can determine the state of the mouse.

NOTIFY_CLICK_LEFT
NOTIFY_CLICK_RIGHT
NOTIFY_CLICK_LEFT_DBL
NOTIFY_CLICK_RIGHT_DBL

Additional Information

idoGetNotifyEntity(HWND *hIDO*, LPENTITY *lpEntity*);

Stores information about the clicked entity in LPENTITY.

Visual Basic

ClickEntity(*MouseStatus* as Long)

Parameters

A status code so you can determine the state of the mouse.

idoLeftClick
idoRightClick
idoLeftDbClick
idoRightDbClick

Additional Information

Entity = IDO.NotifyEntity

Returns [an entity object](#) referring to the clicked entity.

C++

ClickEntity(long *MouseStatus*);

Parameters

A status code so you can determine the state of the mouse.

idoLeftClick
idoRightClick
idoLeftDbClick
idoRightDbClick

Additional Information

IDO.NotifyEntity(*Entity*);

Stores information about the clicked entity in [an entity object](#).

ClickRelation

This event is fired when the mouse is clicked or double-clicked on a relation.

ANSI C

IDOM_N_CLICK_RELATION

IParam

A status code so you can determine the state of the mouse.

NOTIFY_CLICK_LEFT
NOTIFY_CLICK_RIGHT
NOTIFY_CLICK_LEFT_DBL
NOTIFY_CLICK_RIGHT_DBL

Additional Information

idoGetNotifyRelation(HWND *hIDO*, LPRELATION
lpRelation);

Stores information about the clicked relation in LPRELATION..

Visual Basic

ClickRelation(*MouseStatus* as Long)

Parameters

A status code so you can determine the state of the mouse.

idoLeftClick
idoRightClick
idoLeftDbClick
idoRightDbClick

Additional Information

Relation = *IDO.NotifyRelation*

Returns [a relation object](#) referring to the clicked relation.

C++

ClickRelation(long *MouseStatus*);

Parameters

A status code so you can determine the state of the mouse.

idoLeftClick
idoRightClick
idoLeftDbClick
idoRightDbClick

Additional Information

IDO.NotifyRelation(*Relation*);

Stores information about the clicked relation in [a relation object](#).

ClickInterAct

This event is fired when the mouse is clicked or double-clicked on the InterAct background..

ANSI C

IDOM_N_CLICK_IDO

IParam

A status code so you can determine the state of the mouse.

NOTIFY_CLICK_LEFT
NOTIFY_CLICK_RIGHT
NOTIFY_CLICK_LEFT_DBL
NOTIFY_CLICK_RIGHT_DBL

Visual Basic

ClickInterAct(*MouseStatus* as Long)

Parameters

A status code so you can determine the state of the mouse.

idoLeftClick
idoRightClick
idoLeftDbClick
idoRightDbClick

C++

ClickInterAct(long *MouseStatus*);

Parameters

A status code so you can determine the state of the mouse.

idoLeftClick
idoRightClick
idoLeftDbClick
idoRightDbClick

Entity Events

EntityAddRequest

This event is fired when an entity object is about to be added to a diagram. If necessary, you can abort the entity addition.

ANSI C

IDOM_N_ENTITYADDREQUEST

IParam

A long pointer to an entity object describing the new entity.
Note you cannot set properties or invoke methods on this entity as it does not exist in the diagram yet.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

EntityAddRequest(*ID* as String, *Name* as String, *ClassName* as String, *Text* as String, *Top* as Integer, *Bottom* as Integer, *Left* as Integer, *Right* as Integer, *Status* as Long)

C++

EntityAddRequest(long FAR * *ID*, BSTR FAR * *Name*, BSTR FAR * *ClassName*, BSTR FAR * *Text*, short FAR * *Top*, short FAR * *Bottom*, short FAR * *Left*, short FAR * *Right*, long FAR * *Status*);

Parameters

The ID of the new entity.

The name of the new entity.

The classname of the entity.

The text of the entity.

The top, bottom, left, and right coordinate of the entity.

A status code.

How to Cancel

Set *Status* = IDO_CANCELACTION

EntityAdded

This event is fired after an entity object has been added to a diagram.

ANSI C

IDOM_N_ENTITYADDED

IParam

A long pointer to an entity object describing the new entity.
Note you can set properties or invoke methods on this entity.

Visual Basic

EntityAdded(*Entity* as Object)

C++

EntityAdded(LPDISPATCH *Entity*);

Parameters

A reference to the entity object.

EntityDeleteRequest

This event is fired before an entity object is about to be deleted from a diagram. This action can be canceled.

ANSI C

IDOM_N_ENTITYDELETEREQUEST

IParam

A long pointer to an entity object describing the entity about to be deleted. Note you can set properties or invoke methods on this entity.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

EntityDeleteRequest(*Entity* as Object, *Status* as Long)

C++

EntityDeleteRequest(LPDISPATCH *Entity*, long FAR * *Status*);

Parameters

A reference to the entity object.

A status code.

How to Cancel

Set *Status* = IDO_CANCELACTION

EntityDeleted

This event is fired after an entity object has been deleted from a diagram.

ANSI C

IDOM_N_ENTITYDELETED

IParam

A long pointer to an entity object describing the entity about to be deleted. Note you cannot set properties or invoke methods on this entity as it no longer exists in the diagram.

Visual Basic

EntityDeleted(*ID* as Long, *Name* as String)

C++

EntityDeleted(long *ID*, LPCSTR *Name*);

Parameters

The ID of the entity just deleted.

The name of the entity just deleted.

EntityBeforeMove

This event is fired when the mouse is used to move and entity in a diagram. This event passes the old and new coordinates of the entity and gives the container an opportunity to change where the entity will be placed after being moved.

ANSI C

IDOM_N_ENTITYBEFOREMOVE

IParam

A long pointer to a RECT structure. This RECT structure contains the left, right, top and bottom coordinates of the entity's new location.

Additional Information

idoGetNotifyEntity(HWND *hIDO*, LPENTITY *lpEntity*);

Stores information about the moved entity in LPENTITY.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

EntityBeforeMove(*Entity* as Object, *Left* as Integer, *Right* as Integer, *Top* as Integer, *Bottom* as Integer, *MoveStatus* as Long)

C++

EntityBeforeMove(LPDISPATCH *Entity*, short FAR * *Left*, short FAR * *Right*, short FAR * *Top*, short FAR * *Bottom*, long FAR * *MoveStatus*);

Parameters

A reference to the entity object.

The new left, right, top, and bottom coordinates of the entity.

A status code to override the move.

How to Cancel

Set *Status* = IDO_CANCELACTION

EntityMoved

This event is fired after the mouse is used to move and entity in a diagram. This event passes a reference to the moved entity, which the user can then query for its new coordinates.

ANSI C

IDOM_N_ENTITYMOVED

lParam

No meaningful value. Reserved for future.

Additional Information

idoGetNotifyEntity(HWND *hIDO*, LPENTITY *lpEntity*);

Stores information about the moved entity in LPENTITY.

Visual Basic

EntityMoved(*Entity* as Object)

C++

EntityBeforeMove(LPDISPATCH *Entity*);

Parameters

A reference to the entity object.

Relation Events

RelationAddRequest

This event is fired when a relation object is about to be added to a diagram. If necessary, you can abort the relation addition.

ANSI C

IDOM_N_RELATIONADDREQUEST

IParam

A long pointer to a relation object describing the new relation.
Note you cannot set properties or invoke methods on this relation as it does not exist in the diagram yet.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

RelationAddRequest(*ID* as String, *Name* as String, *ClassName* as String, *Text* as String, SourceEntity as Object, DestinationEntity as Object, *Status* as Long)

C++

RelationAddRequest(long FAR * *ID*, BSTR FAR * *Name*, BSTR FAR * *ClassName*, BSTR FAR * *Text*, LPDISPATCH *SourceEntity*, LPDISPATCH *DestinationEntity*, long FAR * *Status*);

Parameters

The ID of the new relation.

The name of the new relation.

The classname of the relation.

The text of the relation.

A reference to the source entity.

A reference to the destination entity.

A status code.

How to Cancel

Set *Status* = IDO_CANCELACTION

RelationAdded

This event is fired after a relation object has been added to a diagram.

ANSI C

IDOM_N_RELATIONADDED

IParam

A long pointer to a relation object describing the new relation.
Note you can set properties or invoke methods on this relation.

Visual Basic

RelationAdded(*Relation* as Object)

C++

RelationAdded(LPDISPATCH *Relation*);

Parameters

A reference to the relation object.

RelationDeleteRequest

This event is fired before a relation object is about to be deleted from a diagram. This action can be canceled.

ANSI C

IDOM_N_RELATIONDELETEREQUEST

IParam

A long pointer to a relation object describing the relation about to be deleted. Note you can set properties or invoke methods on this relation.

How to Cancel

idoCancelAction(HWND *hIDO*);

Visual Basic

RelationDeleteRequest(*Relation* as Object, *Status* as Long)

C++

RelationDeleteRequest(LPDISPATCH *Relation*, long FAR *
Status);

Parameters

A reference to the relation object.

A status code.

How to Cancel

Set *Status* = IDO_CANCELACTION

RelationDeleted

This event is fired after a relation object has been deleted from a

diagram.

ANSI C

IDOM_N_RELATIONDELETED

IParam

A long pointer to a relation object describing the entity about to be deleted. Note you cannot set properties or invoke methods on this relation as it no longer exists in the diagram.

Visual Basic

RelationDeleted(*ID* as Long, *Name* as String)

C++

RelationDeleted(long *ID*, LPCSTR *Name*);

Parameters

The ID of the relation just deleted.

The name of the relation just deleted.

Introduction

Properties are simple attributes exposed by an object like InterAct to allow a program to tailor the way InterAct will look and behave. Properties can be set or queried. InterAct exposes properties that allow you to specify colors for the diagram, whether a border or scrollbars will be displayed, if a tools palette will be available, and various properties for gridlines.

BackColor

This property determines the color used to paint the InterAct background.

Syntax

ANSI C

```
idoSetBackColor ( IDO, value );  
value = idoGetBackColor ( IDO );
```

Visual Basic

```
IDO.BackColor [= value]  
value = IDO.BackColor
```

C++

```
IDO.SetBackColor ( value );  
value = IDO.GetBackColor ( );
```

The **BackColor** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>value</i>	A long value specifying a color for the InterAct background.

Settings

The settings for *value* are:

Setting	Description
Color Value	A long value describing the color to paint the InterAct background. This is often created using the RGB command.

See Also

GridLineColor

Border

This property determines whether InterAct will display a black rectangle around its edges to mark its border.

Syntax

ANSI C

```
idoSetBorder ( IDO, bEnabled );  
bEnabled = idoGetBorder ( IDO );
```

Visual Basic

```
IDO.Border [= bEnabled]  
bEnabled = IDO.Border
```

C++

```
IDO.SetBorder ( bEnabled );
```

```
bEnabled = IDO.GetBorder ( );
```

The **Border** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bEnabled</i>	A Boolean value describing if InterAct will display a border.

Settings

The settings for *bEnabled* are:

Setting	Description
TRUE	InterAct is displaying a border.
FALSE	InterAct is not displaying a border.

CurrentEntity

This property retrieves or sets the current entity in a diagram. The current entity is the one which contains the tracking rectangle.

Syntax

ANSI C

```
idoSetCurrentEntity ( IDO, Entity );  
idoGetCurrentEntity ( IDO, Entity );
```

Visual Basic

```
IDO.CurrentEntity [= Entity]  
Entity = IDO.CurrentEntity
```

C++

```
IDO.SetCurrentEntity ( Entity );  
Entity = IDO.GetCurrentEntity ( );
```

The **CurrentEntity** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>Entity</i>	An entity object returned by or passed to InterAct.

See Also

CurrentRelation

CurrentRelation

This property retrieves or sets the current relation in the diagram.

Syntax

ANSI C

```
idoSetCurrentRelation ( IDO, Relation );  
idoGetCurrentRelation ( IDO, Relation );
```

Visual Basic

```
IDO.CurrentRelation [=Relation]
```

```
Relation = IDO.CurrentRelation
```

C++

```
IDO.SetCurrentRelation (Relation );
```

```
Relation = IDO.GetCurrentRelation ( );
```

The **CurrentRelation** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>Relation</i>	A relation object returned by or passed to InterAct.

See Also

CurrentEntity

EditMode

This property returns or sets the edit mode for InterAct. When editing is disabled, entities and lines cannot be modified (moved or resized), right-clicking on InterAct will not display popup menus, and the tools palette will not be displayed.

Syntax

ANSI C

```
idoSetEditMode ( IDO, bEditable );
```

```
bEditable = idoGetEditMode ( IDO );
```

Visual Basic

```
IDO.EditMode [= bEditable]
```

```
bEditable = IDO.EditMode
```

C++

```
IDO.SetEditMode ( bEditable );
```

```
bEditable = IDO.GetEditMode ( );
```

The **EditMode** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bEditable</i>	A Boolean value describing if InterAct is editable.

Settings

The settings for *bEditable* are:

Setting	Description
TRUE	InterAct is editable.

FALSE

InterAct is not editable (read-only).

See Also

ToolsPalette

FileName

This property returns or sets the name of the diagram currently opened in InterAct. InterAct will update this value whenever the method **ReadDiagram** is invoked. If you invoke **SaveDiagram** and do not specify a file to save to, InterAct will use the value stored here to save the diagram, if a value is present. If you invoke **ResetDiagram** this value is not reset. InterAct does not actually read or save a file if you set this property programmatically - use the methods **ReadDiagram** and **SaveDiagram**. This value does not affect palette files in any way.

Syntax

ANSI C

```
idoSetFileName ( IDO, value );  
value = idoGetFileName ( IDO );
```

Visual Basic

```
IDO.FileName [= value]  
value = IDO.FileName
```

C++

```
IDO.SetFileName ( value );  
value = IDO.GetFileName ( );
```

The **FileName** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>value</i>	A string which can contain a path and file name for a diagram currently opened in InterAct.

See Also

ReadDiagram, SaveDiagram, ResetDiagram

GridHeight

This property determines the vertical spacing of gridlines in InterAct.

Syntax

ANSI C

```
idoSetGridHeight ( IDO, value );  
cHeight = idoGetGridHeight ( IDO );
```

Visual Basic

```
IDO.GridHeight [= cHeight]  
cHeight = IDO.GridHeight
```

C++

```
IDO.SetGridHeight ( cHeight );
cHeight = IDO.GetGridHeight ( );
```

The **GridHeight** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>cHeight</i>	An integer value describing the height of the grid displayed in InterAct.

Settings

The settings for *cHeight* are:

Setting	Description
1-999	The height of the grid in InterAct.

See Also

GridLines, GridWidth

GridLineColor

This property determines the color used to paint gridlines in InterAct.

Syntax

ANSI C

```
idoSetGridLineColor ( IDO, value );
value = idoGetGridLineColor ( IDO );
```

Visual Basic

```
IDO.GridLineColor [= value]
value = IDO.GridLineColor
```

C++

```
IDO. SetGridLineColor ( value );
value = IDO. GetGridLineColor ( );
```

The **GridLineColor** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>value</i>	A long value specifying a color for the InterAct gridlines.

Settings

The settings for *value* are:

Setting	Description
Color Value	A long value describing the color to paint the InterAct gridlines. This is often created using the RGB command.

See Also

BackColor, GridLines

GridLines

This property determines whether gridlines will be displayed in InterAct.

Syntax

ANSI C

```
idoSetGridLines ( IDO, bEnabled );  
bEnabled = idoGetGridLines ( IDO );
```

Visual Basic

```
IDO.GridLines [= bEnabled]  
bEnabled = IDO.GridLines
```

C++

```
IDO.SetGridLines ( bEnabled );  
bEnabled = IDO.GetGridLines ( );
```

The **GridLines** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bEnabled</i>	A Boolean value describing if InterAct is displaying gridlines.

Settings

The settings for *bEnabled* are:

Setting	Description
TRUE	InterAct is displaying gridlines.
FALSE	InterAct is not displaying gridlines.

See Also

GridLineColor, SnapToGrid

GridWidth

This property determines the horizontal spacing of gridlines in InterAct.

Syntax

ANSI C

```
idoSetGridWidth ( IDO, cWidth );  
cWidth = idoGetGridWidth ( IDO );
```

Visual Basic

```
IDO.GridWidth [= cWidth]  
cWidth = IDO.GridWidth
```

C++

```
IDO.SetGridWidth ( cWidth );
cWidth = IDO.GetGridWidth ( );
```

The **GridWidth** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>cWidth</i>	An integer value describing the width of the grid displayed in InterAct.

Settings

The settings for *cWidth* are:

Setting	Description
1-999	The width of the grid in InterAct.

See Also

GridLines, GridHeight

HorizontalScrollOffset

This property determines the horizontal offset of the InterAct window. This property affects the display of the scrollbars if scrollbars are visible. See the notes below on the InterAct coordinate system. Setting this property will not repaint the InterAct control. This is intentional to allow you to set both the horizontal and vertical offset without unnecessary flicker. See the InterAct **Redraw** property to force a repaint of the InterAct control.

Syntax

ANSI C

```
idoSetHorizontalScrollOffset ( IDO, value );
value = idoGetHorizontalScrollOffset ( IDO );
```

Visual Basic

```
IDO.HorizontalScrollOffset [= value ]
value = IDO.HorizontalScrollOffset
```

C++

```
IDO.SetHorizontalScrollOffset ( value );
value = IDO.GetHorizontalScrollOffset ( );
```

The **HorizontalScrollOffset** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>value</i>	A short value which determines the offset in logical units. This value cannot be set less than zero.

See Also

VerticalScrollOffset, Redraw

hWnd

Returns a handle to InterAct. Read-only at run time, not available at design time. The Microsoft Windows operating environment identifies each form and control in an application by assigning it a handle, or hWnd. The hWnd property is used with Windows API calls. Many Windows operating environment functions require the hWnd of the active window as an argument.

Note: Because the value of this property can change while a program is running, never store the hWnd value in a variable.

Syntax

Visual Basic

```
value = IDO.hWnd
```

C++

```
value = IDO.GetHWND ( );
```

The **hWnd** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>value</i>	A handle which corresponds to the Window handle of the InterAct control.

InetPath

This property sets an optional path InterAct will use to locate an internet host when executing a URL.

Syntax

ANSI C

```
idoSetInetPath ( IDO, Path );  
Path = idoGetInetPath ( IDO );
```

Visual Basic

```
IDO.InetPath [= Path]  
Path = IDO.InetPath
```

C++

```
IDO.SetInetPath ( Path );  
Path = IDO.GetInetPath ( );
```

The **InetPath** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>Path</i>	A string which contains a fully qualified or relative path for InterAct to use when looking for an internet host.

See Also

Modified

This property determines whether the InterAct has been modified.

Syntax

ANSI C

```
idoSetModified ( IDO, bModified );  
bModified = idoGetModified ( IDO );
```

Visual Basic

```
IDO.Modified [= bModified]  
bModified = IDO.Modified
```

C++

```
IDO.SetModified ( bModified );  
bModified = IDO.GetModified ( );
```

The **Modified** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bModified</i>	A Boolean which determines if the InterAct environment has been altered..

Settings

The settings for *bModified* are:

Setting	Description
TRUE	InterAct has been modified.
FALSE	InterAct has not been modified.

PopupMenu

This property determines whether InterAct will display its built-in menus when an entity, relation or the InterAct background is right-clicked. By setting these properties to TRUE, you are telling InterAct to display its own defined menus. By setting this value to FALSE you are telling InterAct not to display its built-in menus. If you wish to display your own menus you can respond to the mouse click or double-click events.

You can turn on or off InterAct's ability to display built-in menus differently for the three types of objects in InterAct. This means InterAct can display its built-in menus for entities and relations, but can be suppressed from displaying a menu when the InterAct background is clicked.

InterAct will not be displayed its own popup menus unless InterAct also has **EditMode** enabled.

Syntax

ANSI C

```
idoSetPopupMenu ( IDO, flag, bDisplayed );  
bDisplayed = idoGetPopupMenu ( IDO );
```

Visual Basic

```
IDO.PopupMenu ( flags, bDisplayed )  
bDisplayed = IDO.PopupMenu ( flag )
```

C++

```
IDO.SetPopupMenu ( flag, bDisplayed );  
IDO.GetPopupMenu ( flag );
```

The **PopupMenu** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>flag</i>	A value which specifies whether which menu is going to be enabled or disabled in InterAct.
<i>bDisplayed</i>	A Boolean value describing if InterAct should display its own defined menus.

Settings

The settings for *flag* are:

Setting	Description
IDOMENU_IDO	The constant <i>bDisplayed</i> determines whether the InterAct's build-in menu is displayed when the InterAct background is right-clicked.
IDOMENU_ENTITY	The constant <i>bDisplayed</i> determines whether the InterAct's build-in menu is displayed when an entity is right-clicked.
IDOMENU_LINE	The constant <i>bDisplayed</i> determines whether the InterAct's build-in menu is displayed when a relation is right-clicked.

The settings for *bDisplayed* are:

Setting	Description
TRUE	InterAct is displaying the tools palette.
FALSE	InterAct is not displaying the tools palette.

See Also

EditMode

PrinterLandscape

This property determines the orientation of the printer page lines displayed by the InterAct. In landscape mode the horizontal and vertical dimensions of the page lines are reversed.

Syntax

ANSI C

```
idoSetPrinterLandscape ( IDO, value );  
value = idoGetPrinterLandscape ( IDO );
```

Visual Basic

```
IDO.PrinterLandscape [= value ]  
value = IDO.PrinterLandscape
```

C++

```
IDO.SetPrinterLandscape ( value );  
value = IDO.GetPrinterLandscape ( );
```

The **PrinterLandscape** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>value</i>	A Boolean which determines if the InterAct will display page lines in landscape mode.

Settings

The settings for *bModified* are:

Setting	Description
TRUE	InterAct will display page lines in landscape mode.
FALSE	InterAct will not display page lines in landscape mode.

See Also

PrinterLines

PrinterLines

This property determines whether printer lines will be displayed in InterAct. Printer Lines are this solid lines displayed within a diagram. These provide a preview of a diagram when printed - showing where page breaks will occur if a multi-page diagram is printed. InterAct must be able to get access to the default printer of a workstation to properly calculate this value.

Syntax

ANSI C

```
idoSetPrinterLines ( IDO, bEnabled );  
bEnabled = idoGetPrinterLines ( IDO );
```

Visual Basic

```
IDO.PrinterLines [= bEnabled]  
bEnabled = IDO.PrinterLines
```

C++

```
IDO.SetPrinterLines ( bEnabled );
bEnabled = IDO.GetPrinterLines ( );
```

The **PrinterLines** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>bEnabled</i>	A Boolean value describing if InterAct is displaying printer page lines.

Settings

The settings for *bEnabled* are:

Setting	Description
TRUE	InterAct is displaying page lines.
FALSE	InterAct is not displaying page lines.

See Also

GridLines, PrinterLandscape

Redraw

This property determines if InterAct will repaint itself when a change within a diagram occurs. This is useful for turning off redraw when alot of changes are anticipated, such as adding several entities and relations at one time, and it is quicker to have InterAct redraw itself only after all the objects have been added rather than after each item is added. When you set redraw to TRUE InterAct will repaint itself immediately.

Syntax

ANSI C

```
idoSetRedraw ( IDO, bEnabled );
```

Visual Basic

```
IDO.Redraw [= bEnabled]
```

C++

```
IDO.SetRedraw ( bEnabled );
```

The **Redraw** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>bEnabled</i>	A Boolean value describing if redraw is on or off.

Settings

The settings for *bEnabled* are:

Setting	Description
TRUE	InterAct will redraw itself after a change.
FALSE	InterAct will not redraw itself after a change.

RulesEnforced

This property determines whether rules will be enforced in a diagram. Rules determine whether certain relations can be made between two entities. It may be desirable to have only certain entities be joined by specific relations.

Syntax

ANSI C

```
idoSetRulesEnforced ( IDO, bEnabled );  
bEnabled = idoGetRulesEnforced ( IDO );
```

Visual Basic

```
IDO.RulesEnforced [= bEnabled]  
bEnabled = IDO.RulesEnforced
```

C++

```
IDO.SetRulesEnforced( bEnabled );  
bEnabled = IDO.GetRulesEnforced ( );
```

The **RulesEnforced** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bEnabled</i>	A Boolean value describing if rules are being enforced for a diagram.

Settings

The settings for *bEnabled* are:

Setting	Description
TRUE	InterAct is enforcing rules.
FALSE	InterAct is not enforcing rules.

ScrollBars

This property determines whether horizontal and/or vertical scrollbars will be displayed on InterAct.

Syntax

ANSI C

```
idoSetScrollBars ( hIDO, value );  
value = idoGetScrollBars ( hIDO );
```

Visual Basic

```
IDO.ScrollBars [= value]  
value = IDO.ScrollBars
```

C++

```
IDO.SetScrollBars ( value );
```

```
value = IDO.GetScrollBars ( );
```

The **ScrollBars** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>value</i>	An integer specifying which scrollbars <i>IDO.ScrollBars</i> [= <i>value</i>] will display.

Settings

The settings for *value* are:

Setting	Description
0	None. InterAct will display no scrollbars. Note this does not prevent the user from scrolling InterAct using arrow keys.
1	Horizontal only. InterAct will display horizontal scrollbars only.
2	Vertical only. InterAct will display vertical scrollbars only.
3	Both. InterAct will display horizontal and vertical scrollbars.

SnapToGrid

This property determines whether entities will “snap” to the nearest grid lines for easy positioning. The upper-left corner of an entity will be moved to the nearest gridline. Even if gridlines are not being displayed, you can still have snap-to-grid enforced. Entities will be “snapped” when they are first created or whenever they are moved. If snap-to-grid is enabled in a diagram which previously did not have snap-to-grid enabled, all entities will be immediately “snapped.”

Syntax

ANSI C

```
idoSetSnapToGrid ( IDO, bEnabled );  
bEnabled = idoGetSnapToGrid ( IDO );
```

Visual Basic

```
IDO.SnapToGrid [= bEnabled]  
bEnabled = IDO.SnapToGrid
```

C++

```
IDO.SetSnapToGrid ( bEnabled );  
bEnabled = IDO.GetSnapToGrid ( );
```

The **SnapToGrid** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bEnabled</i>	A Boolean value describing if snap-to-grid is

enabled for InterAct.

Settings

The settings for *bEnabled* are:

Setting	Description
TRUE	InterAct is enforcing snap-to-grid.
FALSE	InterAct is not enforcing snap-to-grid.

See Also

GridLines

ToolsPalette

This property determines whether a tools palette is displayed. The tools palette is an interface that allows users to add entities and relations to a diagram, cut/copy/paste objects to the clipboard, zoom the diagram for different displays, and load or save diagrams. If the program does not display the tools palette but intends to allow the user to access these capabilities, then the program must provide its own UI which takes advantage of the InterAct's exposed methods.

The tools palette will not be displayed unless InterAct also has **EditMode** enabled.

Syntax

ANSI C

```
idoSetToolsPalette ( IDO, bDisplayed );  
bDisplayed = idoGetToolsPalette ( IDO );
```

Visual Basic

```
IDO.ToolsPalette [= bDisplayed]  
bDisplayed = IDO.ToolsPalette
```

C++

```
IDO.SetToolsPalette ( bDisplayed );  
bDisplayed = IDO.GetToolsPalette ( );
```

The **ToolsPalette** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>bDisplayed</i>	A Boolean value describing if the tools palette is displayed for InterAct.

Settings

The settings for *bDisplayed* are:

Setting	Description
TRUE	InterAct is displaying the tools palette.
FALSE	InterAct is not displaying the tools palette.

See Also

EditMode

This property determines the horizontal offset of the InterAct window. This property affects the display of the scrollbars if scrollbars are visible. See the notes below on the InterAct coordinate system. Setting this property will not repaint the InterAct control. This is intentional to allow you to set both the horizontal and vertical offset without unnecessary flicker. See the InterAct Redraw property to force a repaint of the InterAct control.

Syntax

ANSI C

```
idoSetHorizontalScrollOffset ( IDO, value );  
Path = idoGetHorizontalScrollOffset( IDO );
```

Visual Basic

```
IDO.HorizontalScrollOffset [= Path]  
Path = IDO.HorizontalScrollOffset
```

C++

```
IDO.SetHorizontalScrollOffset ( Path );  
Path = IDO.GetHorizontalScrollOffset ( );
```

The **HorizontalScrollOffset** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Path</i>	A short value which determines the offset in logical units. This value cannot be set less than zero.

See Also

VerticalScrollOffset

VerticalScrollOffset

This property determines the vertical offset of the InterAct window. This property affects the display of the scrollbars if scrollbars are visible. See the notes below on the InterAct coordinate system. Setting this property will not repaint the InterAct control. This is intentional to allow you to set both the horizontal and vertical offset without unnecessary flicker. See the InterAct **Redraw** property to force a repaint of the InterAct control.

Syntax

ANSI C

```
idoSetVerticalScrollOffset ( IDO, value );  
value = idoGetVerticalScrollOffset ( IDO );
```

Visual Basic

```
IDO.VerticalScrollOffset [= value ]
```


value = *IDO.VerticalScrollOffset*

C++

```
IDO.SetVerticalScrollOffset ( value );  
value = IDO.GetVerticalScrollOffset ( );
```

The **VerticalScrollOffset** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>value</i>	A short value which determines the offset in logical units. This value cannot be set less than zero.

See Also

HorizontalScrollOffset, Redraw

WindowStyle

This property sets or retrieves the style bits for the InterAct control. This property is available only for the InterAct DLL. It is suggested programmers use the other properties available to InterAct to change these styles. Refer to the **See Also** section.

Syntax

ANSI C

```
idoSetWindowStyle ( IDO, Style );  
Style = idoGetWindowStyle ( IDO );
```

The **WindowStyle** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>Style</i>	A long value describing the styles for the InterAct control.

Settings

The settings for *Style* can be any combination of the following:

Setting	Description
WS_VISIBLE	Control is visible.
WS_BORDER	A border is drawn.
WS_HSCROLL	A horizontal scrollbar is present.
WS_VSCROLL	A vertical scrollbar is present.

See Also

ScrollBars, Border

ZoomValue

This property determines the amount of zooming the InterAct control will perform on a diagram. The zoom value can be in the range 10 - 200. The

value 100 means no zooming. The greater the zoom value the less of the diagram you can see (you are zooming into the diagram to see more detail). The lesser the zoom value the more of the diagram you can see (you are zooming out of the diagram to see less detail but more area). InterAct by default multiplies the zoom value by 0.8 when zooming in and 1.2 when zooming out for smooth motion.

Syntax

ANSI C

```
idoSetZoomValue ( IDO, value );  
value = idoGetZoomValue ( IDO );
```

Visual Basic

```
IDO.ZoomValue [= value ]  
value = IDO.ZoomValue
```

C++

```
IDO.SetZoomValue ( value );  
value = IDO.GetZoomValue ( );
```

The **ZoomValue** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>value</i>	A short value describing which determines the amount to zoom the diagram

See Also

Zoom

Introduction

Methods are routines you can access to make the IDO perform special actions. You can add or remove entities or relations, position objects in a diagram, and save the diagram so you can load and display it at a later time.

AddEntityClass

This adds a new entity class to the diagram. A unique class name must be provided to allow you to identify the entity class later if you wish to define attributes on the class. In addition, a bitmap can be supplied with the class. This bitmap will be displayed on the tools palette for that class.

Syntax

ANSI C

```
idoAddEntityClass ( IDO, ClassName, BitmapName );
```

Visual Basic

```
IDO.AddEntityClass ClassName, BitmapName
```

C++

```
IDO.AddEntityClass ( ClassName, BitmapName );
```

The **AddEntityClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which contains a name to uniquely identify the entity class.
<i>BitmapName</i>	A string which contains the name of a bitmap file (with a path, if necessary) which will be loaded and displayed for this class on the tools palette.

See Also

DeleteEntityClass, DoesEntityClassExist, RedefineFromEntityClass, GetEntityClass, IterateEntityClassFirst, IterateEntityClassNext

AddEntityFromClass

This adds a new entity to the diagram. A unique ID or name must be provided to allow you to identify the entity later.

Syntax

ANSI C

```
idoAddEntityFromClass ( IDO, ID, Name, Class, Text, X, Y, cX, cY );
```

Visual Basic

```
IDO.AddEntityFromClass ID, Name, Class, Text, X, Y, cX, cY
```

C++

```
IDO.AddEntityFromClass ( ID, Name, Class, Text, X, Y, cX, cY );
```

The **AddEntityFromClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ID</i>	A long which will uniquely identify the entity.
<i>Name</i>	A string which will uniquely identify the entity.

<i>Class</i>	A string which determines the type of entity to add.
<i>Text</i>	A string which contains the text for the new entity.
<i>X</i>	An integer to contain the x position of the new entity.
<i>Y</i>	An integer to contain the y position of the new entity.
<i>cX</i>	An integer to contain the width of the new entity.
<i>cY</i>	An integer to contain the height of the new entity.

See Also

DeleteEntity, AddRelationFromClass, DeleteRelation

AddRelationClass

This adds a new relation class to the diagram. A unique class name must be provided to allow you to identify the relation class later if you wish to define attributes on the class. In addition, a bitmap can be supplied with the class. This bitmap will be displayed on the tools palette for that class.

Syntax

ANSI C

```
idoAddRelationClass ( IDO, ClassName, BitmapName );
```

Visual Basic

```
IDO.AddRelationClass ClassName, BitmapName
```

C++

```
IDO.AddRelationClass ( ClassName, BitmapName );
```

The **AddRelationClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>ClassName</i>	A string which contains a name to uniquely identify the relation class.
<i>BitmapName</i>	A string which contains the name of a bitmap file (with a path, if necessary) which will be loaded and displayed for this class on the tools palette.

See Also

DeleteRelationClass, DoesRelationClassExist, RedefineFromRelationClass, GetRelationClass, IterateRelationClassFirst, IterateRelationClassNext

AddRelationFromClass

This adds a new relation to the diagram. A unique ID or name must be

provided to allow you to identify the relation later.

Syntax

ANSI C

```
idoAddRelationFromClass ( IDO, ID, Name, Class, Text,  
                          SourceID, SourceName, DestID, DestName );
```

Visual Basic

```
IDO.AddRelationFromClass ID, Name, Class, Text, SourceID,  
                          SourceName, DestID, DestName
```

C++

```
IDO.AddRelationFromClass ( ID, Name, Class, Text, SourceID,  
                          SourceName, DestID, DestName );
```

The **AddRelationFromClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ID</i>	A long which will uniquely identify the new relation.
<i>Name</i>	A string which will uniquely identify the new relation.
<i>Class</i>	A string which determines the type of relation to add.
<i>Text</i>	A string which contains the text for the new relation.
<i>SourceID</i>	A long which will uniquely identify the source entity.
<i>SourceName</i>	A string which will uniquely identify the source entity.
<i>DestID</i>	A long which will uniquely identify the destination entity.
<i>DestName</i>	A string which will uniquely identify the destination entity.

See Also

DeleteRelation, AddEntityFromClass, DeleteEntity

AddRule

This adds a new rule to a diagram.

Syntax

ANSI C

```
idoAddRule ( IDO, SourceClass, RelationClass, TargetClass );
```

Visual Basic

```
IDO.AddRule SourceClass, RelationClass, TargetClass
```

C++

IDO.AddRule (SourceClass, RelationClass, TargetClass);

The **AddRule** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>SourceClass</i>	A string which contains a class name of an entity class.
<i>RelationClass</i>	A string which contains a class name of a relation class.
<i>TargetClass</i>	A string which contains a class name of an entity class.

See Also

DoesRuleExist, DeleteRule, IterateRuleFirst, IterateRuleNext

AllowAction

This method is available only to the InterAct DLL. This can be called in response to an InterAct event to override the default InterAct behavior. See *Chapter 10: InterAct Events* for more details about this function.

Syntax

ANSI C

idoAllowAction (*IDO*);

The **AllowAction** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

CancelAction

CancelAction

This method is available only to the InterAct DLL. This can be called in response to an InterAct event to override the default InterAct behavior. See *Chapter 10: InterAct Events* for more details about this function.

Syntax

ANSI C

idoCancelAction (*IDO*);

The **CancelAction** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

AllowAction

Copy

This copies the selected entity/entities and line(s) to the Windows clipboard.

Syntax

ANSI C

```
idoCopy ( IDO );
```

Visual Basic

```
IDO.Copy
```

C++

```
IDO.Copy ( );
```

The **Copy** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

Cut, Paste

Cut

This cuts the selected entity/entities and line(s) to the Windows clipboard.

Syntax

ANSI C

```
idoCut ( IDO );
```

Visual Basic

```
IDO.Cut
```

C++

```
IDO.Cut ( );
```

The **Cut** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

Copy, Paste

DeleteEntity

This deletes an entity from the diagram.

Syntax

ANSI C

```
idoDeleteEntity ( IDO, Entity );
```

Visual Basic

```
IDO.DeleteEntity Entity
```

C++

```
IDO.DeleteEntity ( Entity );
```


The **DeleteEntity** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Entity</i>	An entity object.

See Also

AddEntityFromClass, DoesEntityExist

DeleteEntityClass

This deletes an entity class definition from the diagram. This does not delete entities of this class which are currently being displayed in a diagram, nor does it remove any rules which refer to this class.

Syntax

ANSI C

```
idoDeleteEntityClass ( IDO, ClassName );
```

Visual Basic

```
IDO.DeleteEntityClass ClassName
```

C++

```
IDO.DeleteEntityClass ( ClassName );
```

The **DeleteEntityClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which contains the name of a class to delete.

See Also

AddEntityClass, RedefineFromEntityClass, DoesEntityClassExist, IterateEntityClassFirst, IterateEntityClassNext

DeleteRelation

This deletes a relation from the diagram.

Syntax

ANSI C

```
idoDeleteRelation ( IDO, Relation );
```

Visual Basic

```
IDO.DeleteRelation Relation
```

C++

```
IDO.DeleteRelation ( Relation );
```

The **DeleteRelation** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

Relation [A relation object.](#)

See Also

AddRelationFromClass, DoesRelationExist

DeleteRelationClass

This deletes an relation class definition from the diagram. This does not delete relations of this class which are currently being displayed in a diagram, nor does it remove any rules which refer to this class.

Syntax

ANSI C

```
idoDeleteRelationClass ( IDO, ClassName );
```

Visual Basic

```
IDO.DeleteRelationClass ClassName
```

C++

```
IDO.DeleteRelationClass ( ClassName );
```

The **DeleteRelationClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which contains the name of a class to delete.

See Also

AddRelationClass, RedefineFromRelationClass, DoesRelationClassExist, IterateRelationClassFirst, IterateRelationClassNext

DeleteRule

This deletes a rule from the diagram.

Syntax

ANSI C

```
idoDeleteRule ( Rule );
```

Visual Basic

```
IDO.DeleteRule Rule
```

C++

```
IDO.DeleteRule ( Rule );
```

The **DeleteRule** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Rule</i>	A rule object.

See Also

AddRule, DoesRuleExist, IterateRuleFirst, IterateRuleNext

DoesEntityExist

This confirms whether a specified entity exists in the diagram.

Syntax

ANSI C

```
idoDoesEntityExist ( IDO, ID, Name );
```

Visual Basic

```
IDO.DoesEntityExist ID, Name
```

C++

```
IDO.DoesEntityExist ( ID, Name );
```

The **DoesEntityExist** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ID</i>	A long which uniquely identifies the entity.
<i>Name</i>	A string which uniquely identifies the entity.

See Also

AddEntityFromClass, DeleteEntity

DoesEntityClassExist

This confirms whether a specified entity class definition exists in the diagram. This does not return whether an entity of the specified class exists within the diagram.

Syntax

ANSI C

```
idoDoesEntityClassExist ( IDO, ClassName );
```

Visual Basic

```
IDO.DoesEntityClassExist ClassName
```

C++

```
IDO.DoesEntityClassExist ( ClassName );
```

The **DoesEntityClassExist** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which identifies an entity class.

See Also

AddEntityClass, RedefineFromEntityClass, DeleteEntityClass, IterateEntityClassFirst, IterateEntityClassNext

DoesRelationExist

This confirms whether a specified relation exists in the diagram.

Syntax

ANSI C

```
idoDoesRelationExist ( IDO, ID, Name );
```

Visual Basic

```
IDO.DoesRelationExist ID, Name
```

C++

```
IDO.DoesRelationExist ( ID, Name );
```

The **DoesRelationExist** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ID</i>	A long which uniquely identifies the relation.
<i>Name</i>	A string which uniquely identifies the relation.

See Also

AddRelationFromClass, DeleteRelation

DoesRelationClassExist

This confirms whether a specified relation class definition exists in the diagram. This does not return whether a relation of the specified class exists within the diagram.

Syntax

ANSI C

```
idoDoesRelationClassExist ( IDO, ClassName );
```

Visual Basic

```
IDO.DoesRelationClassExist ClassName
```

C++

```
IDO.DoesRelationClassExist ( ClassName );
```

The **DoesRelationClassExist** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which identifies an entity class.

See Also

AddRelationClass, RedefineFromRelationClass, DeleteRelationClass, IterateRelationClassFirst, IterateRelationClassNext

DoesRuleExist

This confirms whether a specified rule exists in the diagram.

Syntax

ANSI C

```
idoDoesRuleExist ( IDO, SourceClass, RelationClass, TargetClass );
```

Visual Basic

IDO.DoesRuleExist SourceClass, RelationClass, TargetClass

C++

IDO.DoesRuleExist (SourceClass, RelationClass, TargetClass);

The **DoesRuleExist** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>SourceClass</i>	A string which contains a class name of an entity class.
<i>SourceClass</i>	A string which contains a class name of a relation class.
<i>TargetClass</i>	A string which contains a class name of an entity class.

See Also

AddRule, DeleteRule, IterateRuleFirst, IterateRuleNext

DragAddEntity

This places InterAct into a mode where the user can use the mouse to drag a rectangle in InterAct for a new entity to be placed. This mimics the exact operation InterAct performs if the user clicks on an entity class button on the tools palette. This method is extremely useful for users who do not want to display InterAct's tools palette but do want to design their own palette for interaction with InterAct.

Syntax

ANSI C

idoDragAddEntity (IDO, ClassName);

Visual Basic

IDO.DragAddEntity ClassName

C++

IDO.DragAddEntity (ClassName);

The **DragAddEntity** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which uniquely identifies an entity class name.

See Also

DragAddRelation

DragAddRelation

This places InterAct into a mode where the user can use the mouse to drag a new relation from a source to a destination entity. This method is extremely

useful for users who do not want to display InterAct's tools palette but do want to design their own palette for interaction with InterAct.

Syntax

ANSI C

```
idoDragAddRelation ( IDO, ClassName );
```

Visual Basic

```
IDO.DragAddRelation ClassName
```

C++

```
IDO.DragAddRelation ( ClassName );
```

The **DragAddRelation** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which uniquely identifies a relation class name.

See Also

DragAddEntity

GetEntity

This method selects an entity and prepares it to be manipulated. When using the DLL, the DLL stores information in a structure you pass as a parameter. When using the ActiveX, you actually retrieve an interface to the entity which you can store and use. It is not necessary to supply both an **ID** and a **Name**. If you supply 0 (zero) as the **ID**, only the **Name** will be used. If you supply NULL as the **Name**, only the **ID** will be used.

Syntax

ANSI C

```
idoGetEntity ( IDO, ID, Name, Entity );
```

Visual Basic

```
Entity = IDO.GetEntity ID, Name
```

C++

```
Entity = IDO.GetEntity ( ID, Name );
```

The **GetEntity** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ID</i>	A long which can uniquely identify an entity.
<i>Name</i>	A string which can uniquely identify an entity.
<i>Entity</i>	An entity object returned by InterAct.

See Also

GetRelation

GetEntityClass

This method selects an entity class and prepares it to be manipulated. When using the DLL, the DLL stores information in a structure you pass as a parameter. When using the ActiveX, you actually retrieve an interface to the entity class which you can store and use. Once you have this interface you can invoke entity properties and methods just like you would with an actual entity except you are defining the attributes of the class itself. Any future entities created with this class name will automatically be created with these attributes. If you want existing entities of this class to inherit these changes call **RedefineFromEntityClass** after making the changes.

Syntax

ANSI C

```
idoGetEntityClass ( IDO, ClassName, Entity );
```

Visual Basic

```
Entity = IDO.GetEntityClass ClassName
```

C++

```
Entity = IDO.GetEntityClass ( ClassName );
```

The **GetEntityClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which can uniquely identify an entity class.
<i>Entity</i>	An entity object returned by InterAct.

See Also

AddEntityClass, GetRelationClass

GetNotifyEntity

This method selects an entity and prepares it to be manipulated.

Syntax

ANSI C

```
idoGetNotifyEntity ( IDO, Entity );
```

Visual Basic

```
Entity = IDO.GetNotifyEntity
```

C++

```
Entity = IDO.GetNotifyEntity ( );
```

The **GetNotifyEntity** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Entity</i>	An entity object returned by InterAct.

See Also

GetNotifyRelation

GetNotifyRelation

This method selects an relation and prepares it to be manipulated.

Syntax

ANSI C

```
idoGetNotifyRelation ( IDO, Relation );
```

Visual Basic

```
Relation = IDO.GetNotifyRelation
```

C++

```
Relation = IDO.GetNotifyRelation ( );
```

The **GetNotifyRelation** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Relation</i>	A relation object returned by InterAct.

See Also

GetNotifyEntity

GetNumberOfSelectedEntities

This returns the number of selected entities in a diagram.

Syntax

ANSI C

```
idoGetNumberOfSelectedEntities ( IDO );
```

Visual Basic

```
IDO.GetNumberOfSelectedEntities
```

C++

```
IDO.GetNumberOfSelectedEntities ( );
```

The **GetNumberOfSelectedEntities** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

IterateSelectedEntityFirst, IterateSelectedEntityNext

GetNumberOfEntities

This returns the number of entities in a diagram.

Syntax

ANSI C

idoGetNumberOfEntities (IDO);

Visual Basic

IDO.GetNumberOfEntities

C++

IDO.GetNumberOfEntities ();

The **GetNumberOfEntities** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

GetNumberOfRelations, IterateEntityFirst, IterateEntityNext

GetNumberOfRelations

This returns the number of lines in a diagram.

Syntax

ANSI C

idoGetNumberOfRelations (IDO);

Visual Basic

IDO.GetNumberOfRelations

C++

IDO.GetNumberOfRelations ();

The **GetNumberOfRelations** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

GetNumberOfEntities, IterateRelationFirst, IterateRelationNext

GetRelation

This method selects an relation and prepares it to be manipulated. When using the DLL, the DLL stores information in a structure you pass as a parameter. When using the ActiveX, you actually retrieve an interface to the relation which you can store and use. It is not necessary to supply both an **ID** and a **Name**. If you supply 0 (zero) as the **ID**, only the **Name** will be used. If you supply NULL as the **Name**, only the **ID** will be used.

Syntax

ANSI C

idoGetRelation (IDO, ID, Name, Relation);

Visual Basic

Relation = IDO.GetRelation (ID, Name)

C++

Relation = IDO.GetRelation (ID, Name);

The **GetRelation** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ID</i>	A long which can uniquely identify a relation .
<i>Name</i>	A string which can uniquely identify a relation .
<i>Relation</i>	A relation object returned by InterAct.

See Also

GetEntity

GetRelationClass

This method selects a relation class and prepares it to be manipulated. When using the DLL, the DLL stores information in a structure you pass as a parameter. When using the ActiveX, you actually retrieve an interface to the relation class which you can store and use. Once you have this interface you can invoke relation properties and methods just like you would with an actual relation except you are defining the attributes of the class itself. Any future relations created with this class name will automatically be created with these attributes. If you want existing relations of this class to inherit these changes call **RedefineFromEntityClass** after making the changes.

Syntax

ANSI C

```
idoGetRelationClass ( IDO, ClassName, Relation );
```

Visual Basic

```
Relation = IDO.GetRelationClass ClassName
```

C++

```
Relation = IDO.GetRelationClass ( ClassName );
```

The **GetRelationClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which can uniquely identify a relation class.
<i>Relation</i>	A relation object returned by InterAct.

See Also

AddRelationClass, GetEntityClass

GetVersion

This returns the version number of InterAct.

Syntax

ANSI C

```
version = idoGetVersion ( IDO );
```

Visual Basic

```
version = IDO.GetVersion
```

C++

```
version = IDO.GetVersion ( );
```

The **GetVersion** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>version</i>	A long which contains the InterAct version number.

IterateEntityClassFirst

This prepares InterAct to allow a program to iterate a list of entity classes available in the diagram. This method returns 0 if no entity classes exist in the diagram. If a tools palette is displayed, each entity class will have a corresponding button displayed on the tools palette.

Syntax

ANSI C

```
idoIterateEntityClassFirst ( IDO );
```

Visual Basic

```
IDO.IterateEntityClassFirst
```

C++

```
IDO.IterateEntityClassFirst ( );
```

The **IterateEntityClassFirst** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

IterateEntityClassNext

This returns the next entity class available in the diagram. This method returns 0 when the end of the list is reached. This means an entity class has not been returned.

Syntax

ANSI C

```
idoIterateEntityClassNext ( IDO, szClass );
```

Visual Basic

```
IDO.IterateEntityClassNext szClass
```

C++

```
IDO.IterateEntityClassNext ( szClass );
```

The **IterateEntityClassNext** method syntax has these parts:

Part	Description
------	-------------

IDO

[An InterAct object.](#)

szClass

[A string](#) which will store the next available entity class.

IterateEntityFirst

This prepares InterAct to allow a program to iterate a list of entities available in the diagram. This method returns 0 if no entities exist in the diagram.

Syntax

ANSI C

```
idoIterateEntityFirst ( IDO );
```

Visual Basic

```
IDO.IterateEntityFirst
```

C++

```
IDO.IterateEntityFirst ( );
```

The **IterateEntityFirst** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

IterateEntityNext

This returns the next entity available in the diagram. This method returns 0 when the end of the list is reached. This means an entity has not been returned.

Syntax

ANSI C

```
idoIterateEntityNext ( IDO, Entity );
```

Visual Basic

```
IDO.IterateEntityNext Entity
```

C++

```
IDO.IterateEntityNext ( Entity );
```

The **IterateEntityNext** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Entity</i>	An entity object returned by InterAct.

IterateRelationClassFirst

This prepares InterAct to allow a program to iterate a list of relation classes available in the diagram. This method returns 0 if no relation classes exist in the diagram. If a tools palette is displayed, each relation class will have

a corresponding button displayed on the tools palette.

Syntax

ANSI C

```
idoIterateRelationClassFirst ( IDO );
```

Visual Basic

```
IDO.IterateRelationClassFirst
```

C++

```
IDO.IterateRelationClassFirst ( );
```

The **IterateRelationClassFirst** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

IterateRelationClassNext

This returns the next relation class available in the diagram. This method returns 0 when the end of the list is reached. This means a relation class has not been returned.

Syntax

ANSI C

```
idoIterateRelationClassNext ( IDO, szClass );
```

Visual Basic

```
IDO.IterateRelationClassNext szClass
```

C++

```
IDO.IterateRelationClass ( szClass );
```

The **IterateRelationClassNext** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>szClass</i>	A string which will store the next available relation class.

IterateRelationFirst

This prepares InterAct to allow a program to iterate a list of relations available in the diagram. This method returns 0 if no relations exist in the diagram.

Syntax

ANSI C

```
idoIterateRelationFirst ( IDO );
```

Visual Basic

```
IDO.IterateRelationFirst
```

C++

```
IDO.IterateRelationFirst ( );
```

The **IterateRelationFirst** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

IterateRelationNext

This returns the next relation available in the diagram. This method returns 0 when the end of the list is reached. This means an relation has not been returned.

Syntax

ANSI C

```
idoIterateRelationNext ( IDO, Relation );
```

Visual Basic

```
IDO.IterateRelationNext Relation
```

C++

```
IDO.IterateRelationNext ( Relation );
```

The **IterateRelationNext** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Relation</i>	A relation object returned by InterAct.

IterateRuleFirst

This prepares InterAct to allow a program to iterate a list of rules enforced in the diagram. This method returns 0 if no rules exist in the diagram.

Syntax

ANSI C

```
idoIterateRuleFirst ( IDO );
```

Visual Basic

```
IDO.IterateRuleFirst
```

C++

```
IDO.IterateRuleFirst ( );
```

The **IterateRuleFirst** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

IterateRuleNext

This returns the next rule enforced in the diagram. This method returns 0 when the end of the list is reached. This means a rule has not been returned.

Syntax

ANSI C

```
idoIterateRuleNext ( IDO, Rule );
```

Visual Basic

```
Rule = IDO.IterateRuleNext
```

C++

```
Rule = IDO.IterateRuleNext ( );
```

The **IterateRuleNext** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Rule</i>	A rule object containing the rule definition.

IterateSelectedEntityFirst

This prepares InterAct to allow a program to iterate a list of selected entities available in the diagram. This method returns 0 if no entities are selected in the diagram.

Syntax

ANSI C

```
idoIterateSelectedEntityFirst ( IDO );
```

Visual Basic

```
IDO.IterateSelectedEntityFirst
```

C++

```
IDO.IterateSelectedEntityFirst ( );
```

The **IterateSelectedEntityFirst** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

IterateSelectedEntityNext

This returns the next selected entity in the diagram. This method returns 0 when the end of the list of selected entities is reached. This means an entity has not been returned.

Syntax

ANSI C

```
idoIterateSelectedEntityNext ( IDO, Entity );
```

Visual Basic

```
IDO.IterateSelectedEntityNext Entity
```

C++

```
IDO.IterateSelectedEntityNext ( Entity );
```

The **IterateSelectedEntityNext** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Entity</i>	An entity object returned by InterAct.

MapDeviceToLogical

This method takes a logical point in InterAct and converts it to the corresponding screen coordinate. See the notes below on the InterAct coordinate system.

Syntax

ANSI C

```
idoMapDeviceToLogical ( IDO, Xvalue, Yvalue );
```

Visual Basic

```
IDO.MapDeviceToLogical Xvalue, Yvalue
```

C++

```
IDO.MapDeviceToLogical ( Xvalue, Yvalue );
```

The **MapDeviceToLogical** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Xvalue</i>	A short value passed by reference contains the x coordinate to convert. When this method returns the Xvalue will contain the converted value.
<i>Yvalue</i>	A short value passed by reference contains the y coordinate to convert. When this method returns the Yvalue will contain the converted value..

MapLogicalToDevice

This method takes a device point in InterAct and converts it to the corresponding logical coordinate. See the notes below on the InterAct coordinate system.

Syntax

ANSI C

```
idoMapLogicalToDevice ( IDO, Xvalue, Yvalue );
```

Visual Basic

```
IDO.MapLogicalToDevice Xvalue, Yvalue
```

C++

```
IDO.MapLogicalToDevice ( Xvalue, Yvalue );
```

The **MapLogicalToDevice** method syntax has these parts:

Part	Description
------	-------------

<i>IDO</i>	An InterAct object.
<i>Xvalue</i>	A short value passed by reference contains the x coordinate to convert. When this method returns the Xvalue will contain the converted value.
<i>Yvalue</i>	A short value passed by reference contains the y coordinate to convert. When this method returns the Yvalue will contain the converted value..

ManageClasses

This displays the InterAct's built-in dialog for managing diagram classes.

Syntax

ANSI C

```
idoManageClasses ( IDO );
```

Visual Basic

```
IDO.ManageClasses
```

C++

```
IDO.ManageClasses ();
```

The **ManageClasses** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

ManageRules

This displays the InterAct's built-in dialog for managing diagram rules.

Syntax

ANSI C

```
idoManageRules ( IDO );
```

Visual Basic

```
IDO.ManageRules
```

C++

```
IDO.ManageRules ();
```

The **ManageRules** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

Paste

This pastes entities and lines from the Windows clipboard to a diagram.

Syntax

ANSI C

idoPaste (*IDO*);

Visual Basic

IDO.Paste

C++

IDO.Paste ();

The **Paste** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

Cut, Copy

PrintDiagram

This displays the InterAct's built-in dialog for printing the diagram. This dialog allows a destination printer to be selected as well as pagination.

Syntax

ANSI C

idoPrintDiagram (*IDO*);

Visual Basic

IDO.PrintDiagram

C++

IDO.PrintDiagram ();

The **PrintDiagram** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

PropertyPage

This displays the InterAct's built-in property page dialog for manipulating InterAct itself.

Syntax

ANSI C

idoPropertyPage (*IDO*, *cPage*);

Visual Basic

IDO.PropertyPage *cPage*

C++

IDO.PropertyPage (*cPage*);

The **PropertyPage** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

<i>cPage</i>	A constant which determines which page of the property sheet to initially display.
--------------	--

Settings

The settings for *cPage* are:

Setting	Description
IDO_PPI_COLORS	Initially displays the color property page.
IDO_PPI_GRID	Initially displays the grid property page.

ReadDiagram

This reads a diagram from a file on disk and restores it in [an InterAct object](#). Any existing diagram will be removed before the diagram on disk is restored.

Syntax

ANSI C

```
idoReadDiagram ( IDO, szFileName );
```

Visual Basic

```
IDO.ReadDiagram szFileName
```

C++

```
IDO.ReadDiagram ( szFileName );
```

The **ReadDiagram** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>szFileName</i>	A string with the name of an IDO file. Optionally with a fully qualified path.

See Also

ResetDiagram, SaveDiagram

ReadPalette

This reads a set of entity and relation classes, as well as a set of rules associated with these classes, from a file on disk. If the tools palette is visible, icons for these new classes are added to the tools palette. Any existing rules or classes will still be available in the diagram. If a rule or class already exists in the diagram, any duplicate rules or classes in the palette file will be ignored, but the rest of the classes or rules will still be loaded.

Note: When you save a diagram, you are also by default saving the palette used to create that diagram. When you load a diagram, you will also load that palette. It is possible to use **ReadPalette** on a file created with **SaveDiagram** and extract only the palette information from the diagram file.

Syntax

ANSI C

```
idoReadPalette ( IDO, szFileName );
```

Visual Basic

```
IDO.ReadPalette szFileName
```

C++

```
IDO.ReadPalette ( szFileName );
```

The **ReadPalette** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>szFileName</i>	A string with the name of an IDO palette file. Optionally with a fully qualified path.

See Also

ResetPalette, SavePalette

RedefineFromEntityClass

Often when you change attributes of an entity class you will want all the entities of that class to inherit those changes as well. If you invoke this method, all entities of the class *ClassName* will immediately adopt the characteristics of *ClassName*.

Syntax

ANSI C

```
idoRedefineFromEntityClass ( IDO, ClassName );
```

Visual Basic

```
IDO.RedefineFromEntityClass ClassName
```

C++

```
IDO.RedefineFromEntityClass ( ClassName );
```

The **RedefineFromEntityClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>ClassName</i>	A string which uniquely identifies an entity class definition.

See Also

AddEntityClass, DeleteEntityClass, DoesEntityClassExist, GetEntityClass, IterateEntityClassFirst, IterateEntityClassNext

RedefineFromRelationClass

Often when you change attributes of a relation class you will want all the relations of that class to inherit those changes as well. If you invoke this method, all relations of the class *ClassName* will immediately adopt the characteristics of *ClassName*.

Syntax

ANSI C

```
idoRedefineFromRelationClass ( IDO, ClassName );
```

Visual Basic

IDO.RedefineFromRelationClass *ClassName*

C++

IDO.RedefineFromRelationClass (*ClassName*);

The **RedefineFromRelationClass** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>ClassName</i>	A string which uniquely identifies a relation class definition.

See Also

AddRelationClass, DeleteRelationClass, DoesRelationClassExist, GetRelationClass, IterateRelationClassFirst, IterateRelationClassNext

ResetDiagram

This resets InterAct, deleting all entities and relations.

Syntax

ANSI C

idoResetDiagram (*IDO*);

Visual Basic

IDO.ResetDiagram

C++

IDO.ResetDiagram ();

The **ResetDiagram** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

ReadDiagram, SaveDiagram

ResetPalette

This resets the InterAct's palette, deleting all entity and relation classes and rules associated with InterAct.

Syntax

ANSI C

idoResetPalette (*IDO*);

Visual Basic

IDO.ResetPalette

C++

IDO.ResetPalette ();

The **ResetPalette** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.

See Also

ReadPalette, SavePalette

SaveDiagram

This saves a diagram to permanent storage as a file on disk. This diagram can be restored at any time.

Syntax

ANSI C

```
idoSaveDiagram ( IDO, szFileName );
```

Visual Basic

```
IDO.SaveDiagram szFileName
```

C++

```
IDO.SaveDiagram ( szFileName );
```

The **SaveDiagram** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>szFileName</i>	A string with the name of an IDO file. Optionally with a fully qualified path.

See Also

ReadDiagram, ResetDiagram

SavePalette

This saves a diagram's palette (the collection of rules and classes used to create the diagram) to permanent storage as a file on disk. This palette can be restored at any time for any number of other diagrams, allowing easy reuse of a set of classes and rules.

Note: When you call **SaveDiagram**, you are also saving the diagram's palette at the same time in the same file. **SavePalette** is useful when you wish to save only a set of classes and rules, and not an entire diagram.

Syntax

ANSI C

```
idoSavePalette ( IDO, szFileName );
```

Visual Basic

```
IDO.SavePalette szFileName
```

C++

```
IDO.SavePalette ( szFileName );
```

The **SavePalette** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>szFileName</i>	A string with the name of an IDO palette file. Optionally with a fully qualified path.

See Also

ReadPalette, ResetPalette

SetToolsPaletteButtonText

This method sets the text on the Options button of the default Tools Palette. This text should not be more than ten letters or it may not fit on the button.

Syntax

ANSI C

```
idoSetToolsPaletteButtonText ( IDO, Text );
```

Visual Basic

```
IDO.SetToolsPaletteButtonText Text
```

C++

```
IDO.SetToolsPaletteButtonText ( Text );
```

The **SetToolsPaletteButtonText** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Text</i>	A string with the text to display.

Zoom

This changes the zoom mode of InterAct. Zooming can allow more of a diagram to be visible but with less detail, or more detail in a smaller area displayed.

Syntax

ANSI C

```
idoZoom (IDO, Mode, Percent );
```

Visual Basic

```
IDO.Zoom Mode, Percent
```

C++

```
IDO.Zoom ( Mode, Percent );
```

The **Zoom** method syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Mode</i>	A constant which specifies whether to zoom the diagram view in or out.
<i>Percent</i>	The amount to zoom in or out.

Settings

The settings for *Mode* are:

Setting	Description
IDO_ZOOM_IN	Zoom the view in (show greater detail) by the amount specified in <i>Percent</i> .
IDO_ZOOM_OUT	Zoom the view out (show less detail but more of the diagram) by the amount specified in <i>Percent</i> .
IDO_ZOOM_TO_FIT	Zoom the diagram so the entire contents are visible in the InterAct window.

See Also

ZoomValue

Introduction

Properties are simple attributes exposed by an object like an entity to allow a program to tailor the way the entity will look or behave. Each entity exposes properties that allow you to specify colors for the entity, whether a 3D border will be displayed, if a graphic will be displayed, and various other properties.

InterAct is a collection of entity objects. These entity objects can have properties and methods invoked just like InterAct. For the ActiveX, invoke the method **Entity**, passing the unique ID and/or Name of an entity. If the entity could be located, [an entity object](#) with an interface to that entity will be returned. Using that object you can invoke properties and methods for that particular entity.

```
Dim Ent as Entity  
  
Ent = IDO.Entity 1, "A"  
Ent.Text = "New Text"
```

Visual Basic Example of setting text using [an Entity Object](#).

```
IDO.Entity(1, "A").Text = "New Text"
```

Visual Basic Example of setting text without storing [an Entity Object](#).

```
LPENTITY lpEnt;  
  
lpEnt = IDO.Entity(1, "A");  
lpEnt.SetText("New Day");
```

Visual C++ Example of setting text using [an Entity Object](#).

```
IDO.Entity(1, "A").SetText("New Text");
```

Visual C++ Example of setting text without storing [an Entity Object](#).

For the DLL, a similar approach is used. The container application declares a variable of type ENTITY. It then calls **idoGetEntity**, passing the window handle of InterAct and an ID or UserName to select a particular entity. If the specified entity is located, the variable of type ENTITY will now refer to that entity. You can then tell the DLL to set a property or invoke a method using the variable reference to the entity without the need even for the window handle to InterAct.

```
ENTITY entity;  
  
if(idoGetEntity(hIDO, 1, "A", &entity))  
    idoEntitySetText(&entity, "New Text");
```

ANSI C DLL example of setting text on an entity.

In the above code, first the DLL is told which entity to select for further processing using **idoGetEntity**. The programmer passes a reference to an entity structure. That structure is filled with all information needed to refer to [an entity object](#). Next, the call **idoEntitySetText** is made. The entity reference stored in the structure is passed as a parameter to identify the particular entity to set the text for without having to be given the handle to InterAct or an entity ID or Name again.

AutoResize

This property determines if an entity will automatically resize to fit the text of the entity. If the entity contains no text this property has no effect.

Syntax

ANSI C

```
idoEntitySetAutoResize ( Entity, Value );  
Value = idoEntityGetAutoResize ( Entity );
```

Visual Basic

```
Entity.AutoResize [= Value ]  
Value = Entity.AutoResize
```

C++

```
Entity.SetAutoResize ( Value );  
Value = Entity.GetAutoResize ( );
```

The **AutoResize** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity bject.
<i>Value</i>	A Boolean which determines if the entity will auto resize to its text.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity will auto resize.
FALSE	The entity will not auto resize.

BackColor

This property determines the color used to paint the entity background.

Syntax

ANSI C

```
idoEntitySetBackColor ( Entity, value );  
value = idoEntityGetBackColor ( Entity );
```

Visual Basic

```
Entity.BackColor [= value]  
value = Entity.BackColor
```

C++

```
Entity.SetBackColor ( value );  
value = Entity.GetBackColor ( );
```

The **BackColor** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A long value specifying a color for the entity background.
Settings	
The settings for <i>value</i> are:	
Setting	Description
Color Value	A long value describing the color to paint the entity background. This is often created using the RGB command.
See Also	
TextColor	

Border

This property determines if an entity has a border drawn around it. This border uses the same color as the **TextColor** of the entity. An entity can also have a **Frame** drawn around it in addition to the border.

Syntax

ANSI C

```
idoEntitySetBorder ( Entity, value );
value = idoEntityGetBorder ( Entity );
```

Visual Basic

```
Entity.Border [= value ]
value = Entity.Border
```

C++

```
Entity.SetBorder ( value );
value = Entity.GetBorder ( );
```

The **Border** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A constant which determines if a border is drawn around an entity.
Settings	
The settings for <i>value</i> are:	
Setting	Description
TRUE	The entity has a border drawn around it.
FALSE	The entity has a border drawn around it.
See Also	

Frame

Bottom

This property determines the actual bottom coordinate of an entity.

Syntax

ANSI C

```
idoEntitySetBottom ( Entity, Value );  
Value = idoEntityGetBottom ( Entity );
```

Visual Basic

```
Entity.Bottom = Value  
value = Entity.Bottom
```

C++

```
Entity.Bottom ( Value );  
Value = Entity.Bottom ( );
```

The **Bottom** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	An integer which represents the bottom edge of an entity in a diagram.

See Also

Left, Right, Top, Rect

CanDelete

This property determines if an entity can be deleted by the user with the **Delete** key. This has not affect on whether an entity can be deleted programmatically.

Syntax

ANSI C

```
idoEntitySetCanDelete ( Entity, Value );  
Value = idoEntityGetCanDelete ( Entity );
```

Visual Basic

```
Entity.CanDelete [= Value ]  
Value = Entity.CanDelete
```

C++

```
Entity.SetCanDelete ( Value );  
Value = Entity.GetCanDelete ( );
```

The **CanDelete** property syntax has these parts:

Part	Description
------	-------------

<i>Entity</i>	An entity bject.
<i>Value</i>	A Boolean which determines if the entity can be deleted.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity can be deleted.
FALSE	The entity cannot be deleted.

CanMove

This property determines if an entity can be moved by the user with the mouse. This has not affect on whether an entity can be moved programmatically.

Syntax

ANSI C

```
idoEntitySetCanMove ( Entity, Value );
Value = idoEntityGetCanMove ( Entity );
```

Visual Basic

```
Entity.CanMove [= Value ]
Value = Entity.CanMove
```

C++

```
Entity.SetCanMove ( Value );
Value = Entity.GetCanMove ( );
```

The **CanMove** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity bject.
<i>Value</i>	A Boolean which determines if the entity can be moved.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity can be moved.
FALSE	The entity cannot be moved.

CanResize

This property determines if an entity can be resized by the user with the mouse. This has not affect on whether an entity can be resized programmatically.

Syntax

ANSI C

```
idoEntitySetCanResize ( Entity, Value );  
Value = idoEntityGetCanResize ( Entity );
```

Visual Basic

```
Entity.CanResize [= Value ]  
Value = Entity.CanResize
```

C++

```
Entity.SetCanResize ( Value );  
Value = Entity.GetCanResize ( );
```

The **CanResize** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A Boolean which determines if the entity can be resized.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity can be resized.
FALSE	The entity cannot be resized.

Container

This property determines if an entity behaves as a container. A container is drawn as an empty frame with a caption along the top. The caption will contain the entity's text, if any, and use the **BackColor** property of the entity. Any entities placed inside a container will move when the container is moved. Entities can still be moved inside or out of the container without affecting the container.

Syntax

ANSI C

```
idoEntitySetContainer ( Entity, value );  
value = idoEntityGetContainer ( Entity );
```

Visual Basic

```
Entity.Container [= value ]  
value = Entity.Container
```

C++

```
Entity.SetContainer ( value );  
value = Entity.GetContainer ( );
```

The **Container** property syntax has these parts:

Part	Description
------	-------------

<i>Entity</i>	An entity object.
<i>Value</i>	A constant which determines if the entity behaves as a container.

Settings

The settings for *value* are:

Setting	Description
TRUE	The entity is a container.
FALSE	The entity is not a container.

CoordX

This property determines the X position of an entity according to the diagram grid. This allows for easy positioning of entities within a diagram without having to set the specific X value of each entity. This property is allowed even if gridlines are not displayed.

Syntax

ANSI C

```
idoEntitySetCoordX ( Entity, cX);
cX = idoEntityGetCoordX ( Entity );
```

Visual Basic

```
Entity.CoordX [= cX ]
cX = Entity.CoordX
```

C++

```
Entity.CoordX ( cX);
cX = Entity.CoordX ( );
```

The **CoordX** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>cX</i>	An integer value which specifies the x coordinate to position the entity at.

Settings

The settings for *cX* are:

Setting	Description
0-999	A value which represents the grid line you wish to position the entity on.

See Also

EntityCoordY, AddEntityFromClass

CoordY

This property determines the Y position of an entity according to the diagram grid. This allows for easy positioning of entities within a diagram without

having to set the specific Y value of each entity. This property is allowed even if gridlines are not displayed.

Syntax

ANSI C

```
idoEntitySetCoordY ( Entity, cY );  
cY = idoEntityGetCoordY ( Entity );
```

Visual Basic

```
Entity.CoordY [= cY ]  
cY = Entity.CoordY
```

C++

```
Entity.SetCoordY ( cY );  
cY = Entity.GetCoordY ( );
```

The **CoordY** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>cY</i>	An integer value which specifies the x coordinate to position the entity at.

Settings

The settings for *cY* are:

Setting	Description
0-999	A value which represents the grid line you wish to position the entity on.

See Also

Entity.CoordY, AddEntityFromClass

Font

This property determines the font used to draw the entity's text. For the DLL the font object returned by InterAct is a temporary pointer and must not be saved.

Syntax

ANSI C

```
idoEntitySetFont ( Entity, Font );  
Font = idoEntityGetFont ( Entity );
```

Visual Basic

```
Entity.Font [= Font ]  
Font = Entity.Font
```

C++

```
Entity.SetFont ( Font );  
Font = Entity.GetFont ( );
```


The **Font** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .
<i>Font</i>	A font object which describes the font used to paint the entity.

Frame

This property determines the type of 3D frame drawn around an entity. This frame will give the impression of depth to an entity. An entity can also have a simple border drawn around it in addition to a 3D frame using the **Border** style.

Syntax

ANSI C

```
idoEntitySetFrame ( Entity, value );  
value = idoEntityGetFrame ( Entity );
```

Visual Basic

```
Entity.Frame [= value ]  
value = Entity.Frame
```

C++

```
Entity.SetFrame ( value );  
value = Entity.GetFrame ( );
```

The **Frame** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .
<i>Value</i>	A constant which defines the type of 3D frame drawn around the entity.

Settings

The settings for *value* are:

Setting	Description
STYLE_3D_NONE	The entity has no frame.
STYLE_3D_HEAVYOUTDENT	The entity has a heavy outdented frame.
STYLE_3D_OUTDENT	The entity has a slight outdented frame.
STYLE_3D_HEAVYINDENT	The entity has a heavy indented frame.
STYLE_3D_INDENT	The entity has a slight indented frame.
STYLE_3D_SHADOW	The entity has a shadow displayed below it.

See Also

Border

Graphic

This property sets or returns the graphic displayed in an entity.

Syntax

ANSI C

```
idoEntitySetGraphic ( Entity, Graphic );  
Graphic = idoEntityGetGraphic ( Entity );
```

Visual Basic

```
Entity.Graphic [= Graphic]  
Graphic= Entity.Graphic
```

C++

```
Entity.SetGraphic ( Graphic );  
Graphic = Entity.GetGraphic ( );
```

The **Graphic** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Graphic</i>	An string which contains the name and optional full path to a BMP file.

InetHost

This property sets the host to execute a URL when an entity is double-clicked.

Syntax

ANSI C

```
idoSetInetHost ( IDO, Host );  
Host = idoGetInetHost ( IDO );
```

Visual Basic

```
IDO.InetHost [= Host]  
Host = IDO.InetHost
```

C++

```
IDO.SetInetHost ( Host );  
Host = IDO.GetInetHost ( );
```

The **InetHost** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object.
<i>Host</i>	A constant which represents an internet host.

Settings

The settings for *Host* are:

Setting	Description
MS_INET2	The Microsoft internet explorer, version 2.0 (IEXPLORE.EXE).
NETSCAPE	The Netscape Navigator (NETSCAPE.EXE).

See Also

InetUrl; InetPath in *Chapter 11: InterAct Properties*

InetUrl

This property sets a URL to execute when an entity is double-clicked.

Syntax

ANSI C

```
idoSetInetUrl ( IDO, Url );  
Url = idoGetInetUrl ( IDO );
```

Visual Basic

```
IDO.InetUrl [= Url]  
Url = IDO.InetUrl
```

C++

```
IDO.SetInetUrl ( Url );  
Url = IDO.GetInetUrl ( );
```

The **InetUrl** property syntax has these parts:

Part	Description
<i>IDO</i>	An InterAct object .
<i>Url</i>	A string which contains a URL command to execute to an internet host.

See Also

InetHost; InetPath in *Chapter 11: InterAct Properties*

Left

This property determines the actual left coordinate of an entity.

Syntax

ANSI C

```
idoEntitySetLeft ( Entity, Value );  
Value = idoEntityGetLeft ( Entity );
```

Visual Basic

```
Entity.Left = Value  
value = Entity.Left
```

C++

```
Entity.Left ( Value );  
Value = Entity.Left ( );
```

The **Left** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	An integer which represents the left edge of an entity in a diagram.

See Also

Right, Top, Bottom, Rect

ReadOnly

This property determines if the text of an entity is read-only. This means the user cannot double-click the entity to edit its text.

Syntax

ANSI C

```
idoEntitySetReadOnly ( Entity, Value );  
Value = idoEntityGetReadOnly ( Entity );
```

Visual Basic

```
Entity.ReadOnly [= Value ]  
Value = Entity.ReadOnly
```

C++

```
Entity.SetReadOnly ( Value );  
Value = Entity.GetReadOnly ( );
```

The **ReadOnly** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A Boolean which determines if the entity text is read-only.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity is read-only.
FALSE	The entity is not read-only.

Rect

This property determines the actual top, left, right and bottom coordinates of an entity. This property is only available for the DLL and takes a pointer to a RECT structure for both the Get and Set values. If you call the Get version of the property, the RECT structure will be filled with the values of

the particular entity. If you call the Set version of the property, the entity will adopt the coordinates specified in the RECT structure.

This property is available only for the DLL.

Syntax

ANSI C

```
idoEntitySetRect ( Entity, lpRect );
```

```
idoEntityGetRect ( Entity, lpRect );
```

The **Rect** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>lpRect</i>	A pointer to a RECT structure.

See Also

Left, Right, Top, Bottom

Right

This property determines the actual right coordinate of an entity.

Syntax

ANSI C

```
idoEntitySetRight ( Entity, Value );
```

```
Value = idoEntityGetRight ( Entity );
```

Visual Basic

```
Entity.Right = Value
```

```
value = Entity.Right
```

C++

```
Entity.Right ( Value );
```

```
Value = Entity.Right ( );
```

The **Right** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	An integer which represents the right edge of an entity in a diagram.

See Also

Left, Top, Bottom, Rect

Select

This property determines the selection state of an entity.

Syntax

ANSI C

```
idoEntitySetSelect ( Entity, Value );
```

Value = **idoEntityGetSelect** (*Entity*);

Visual Basic

Entity.**Select** = *Value*

value = *Entity*.**Select**

C++

Entity.**Select** (*Value*);

Value = *Entity*.**Select** ();

The **Select** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A Boolean which determines whether an entity is selected.

See Also

Current

Shape

This property determines the shape of an entity.

Syntax

ANSI C

idoEntitySetShape (*Entity*, *value*);

value = **idoEntityGetShape** (*Entity*);

Visual Basic

Entity.**Shape** [= *value*]

value = *Entity*.**Shape**

C++

Entity.**SetShape** (*value*);

value = *Entity*.**GetShape** ();

The **Shape** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A constant which defines the shape of the entity.

Settings

The settings for *value* are:

Setting	Description
ENTITYSHAPE_ CIRCLE	The entity is elliptic in shape.

[ENTITYSHAPE_RECTANGLE](#)

The entity is rectangular in shape.

[ENTITYSHAPE_ROUNDRECT](#)

The entity is rectangular in shape with rounded corners.

StretchBitmap

This property determines if an entity will stretch a bitmap to occupy its entire area. If the **Graphic** property is not set this property has no effect. (Note: If the **StretchBitmap** property is set, **Transparent** has no effect on the bitmap).

Syntax

ANSI C

```
idoEntitySet StretchBitmap ( Entity, Value );  
Value = idoEntityGet StretchBitmap ( Entity );
```

Visual Basic

```
Entity.StretchBitmap [= Value ]  
Value = Entity.StretchBitmap
```

C++

```
Entity.SetStretchBitmap ( Value );  
Value = Entity.GetStretchBitmap ( );
```

The **StretchBitmap** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A Boolean which determines if the graphic will be stretched.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The graphic is stretched.
FALSE	The graphic is not stretched.

Text

This property sets or returns the text displayed in an entity.

Syntax

ANSI C

```
idoEntitySetText ( Entity, Text );  
Text = idoEntityGetText ( Entity );
```

Visual Basic

```
Entity.Text [= Text ]
```

Text = *Entity*.**Text**

C++

Entity.**SetText** (*Text*);

Text = *Entity*.**GetText** ();

The **Text** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Text</i>	A string which contains the text for the entity.

See Also

TextColor

TextColor

This property determines the color used to paint the entity text.

Syntax

ANSI C

idoEntitySetTextColor (*Entity*, *value*);

value = **idoEntityGetTextColor** (*Entity*);

Visual Basic

Entity.**TextColor** [= *value*]

value = *Entity*.**TextColor**

C++

Entity.**SetTextColor** (*value*);

value = *Entity*.**GetTextColor** ();

The **TextColor** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A long value specifying a color for the entity text.

Settings

The settings for *value* are:

Setting	Description
RGB	A long value describing the color to paint the entity text. This is often created using the RGB command.

See Also

BackColor

TextOrientation

This property determines the orientation of the entity text if the entity is

displaying a graphic. The text can be displayed above, below, to the left or to the right of the entity.

Syntax

ANSI C

```
idoEntitySetTextOrientation ( Entity, value );  
value = idoEntityGetTextOrientation ( Entity );
```

Visual Basic

```
Entity.TextOrientation [= value]  
value = Entity.TextOrientation
```

C++

```
Entity.SetTextOrientation ( value );  
value = Entity.GetTextOrientation ( );
```

The **TextOrientation** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Value</i>	A constant which defines the manner in which text will be drawn on an entity with a graphic.

Settings

The settings for *value* are:

Setting	Description
TEXTPOS_BOTTOM	A constant which specifies the entity's text to be displayed below the entity.
TEXTPOS_TOP	A constant which specifies the entity's text to be displayed above the entity.
TEXTPOS_LEFT	A constant which specifies the entity's text to be displayed to the left of the entity.
TEXTPOS_RIGHT	A constant which specifies the entity's text to be displayed to the right of the entity.

See Also

Graphic

Top

This property determines the actual top coordinate of an entity.

Syntax

ANSI C

```
idoEntitySetTop ( Entity, Value );  
Value = idoEntityGetTop ( Entity );
```

Visual Basic

```
Entity. Top = Value
```

value = *Entity*. **Top**

C++

Entity.**Top** (*Value*);

Value = *Entity*.**Top** ();

The **Top** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .
<i>Value</i>	An integer which represents the top edge of an entity in a diagram.

See Also

Left, Right, Bottom, Rect

Transparent

This property determines if an entity is transparent. This means the entity will not paint its background. Text will be painted as the **TextColor** property specifies. If the **Graphic** property is used, the graphic will also be painted transparent. (**Note**: If the **StretchBitmap** property is also set, **Transparent** has no effect on the bitmap).

Syntax

ANSI C

idoEntitySetTransparent (*Entity*, *Value*);

Value = **idoEntityGetTransparent** (*Entity*);

Visual Basic

Entity.**Transparent** [= *Value*]

Value = *Entity*.**Transparent**

C++

Entity.**SetTransparent** (*Value*);

Value = *Entity*.**GetTransparent** ();

The **Transparent** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity bject.
<i>Value</i>	A Boolean which determines if the entity will be painted transparent.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity is transparent.
FALSE	The entity is not transparent.

UserData

This property allows a container application to store any data with an entity. The UserData can be any 32-bit value ([a long](#)). This means any numerical data can be stored in the UserData, or even a pointer to a object or class.

Syntax

ANSI C

```
idoEntitySetUserData ( Entity, Value );  
Value = idoEntityGetUserData ( Entity );
```

Visual Basic

```
Entity.UserData = Value  
value = Entity.UserData
```

C++

```
Entity.UserData ( Value );  
Value = Entity.UserData ( );
```

The **UserData** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .
<i>Value</i>	A long which contains any value the container application wishes to associate with the entity.

ValidRelationDragSource

This property determines if an entity can used as a source when dragging a new relation. This has not affect on whether an entity can be used as a relation source programmatically.

Syntax

ANSI C

```
idoEntitySetValidRelationDragSource ( Entity, Value );  
Value = idoEntityGetValidRelationDragSource ( Entity );
```

Visual Basic

```
Entity.ValidRelationDragSource [= Value ]  
Value = Entity.ValidRelationDragSource
```

C++

```
Entity.SetValidRelationDragSource ( Value );  
Value = Entity.GetValidRelationDragSource ( );
```

The **ValidRelationDragSource** property syntax has these parts:

Part	Description
<i>Entity</i>	An entity bject.
<i>Value</i>	A Boolean which determines if the entity is a

valid relation source.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The entity can be used as a relation source.
FALSE	The entity cannot be used as a relation source.

Introduction

Methods are routines you can access to make an entity perform special actions. You can iterate a list of relations pointing to or from an entity, repaint it, or display a property page for it.

AddEntityToContainer

This method adds an entity to another entity acting as a container. If the entity is not a container, this method fails. This entity being added to the container does not have to be within the boundaries of the container, and this method will not move the entity into the boundaries of the container. The InterAct container application must perform that action.

Syntax

ANSI C

```
idoEntityAddEntityToContainer ( Container, Entity );
```

Visual Basic

```
Container.AddEntityToContainer Entity
```

C++

```
Container.AddEntityToContainer ( Entity );
```

The **AddEntityToContainer** method syntax has these parts:

Part	Description
<i>Container</i>	An entity object acting as a container.
<i>Entity</i>	An entity object added to the container.

See Also

IterateContainedEntityFirst, IterateContainedEntityNext,
GetContainerEntityCount, RemoveEntityFromContainer

BringIntoView

This method scrolls the InterAct control to bring an entity into view. This function takes into consideration the current zoom mode of the diagram. This function has no effect if the entity is already visible. This function repaints the InterAct control if it is scrolled. This function returns TRUE if the InterAct control was scrolled and FALSE if the control was not scrolled.

Syntax

ANSI C

```
idoEntityBringIntoView ( Entity );
```

```
Entity.classname;
```

Visual Basic

```
Entity.BringIntoView
```

C++

```
Entity.BringIntoView ( );
```

The **BringIntoView** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .

ClassName

This returns [a string](#) which contains the ClassName of the entity.

Syntax

ANSI C

```
idoEntityClassName ( Entity );  
Entity.classname;
```

Visual Basic

```
Entity.ClassName
```

C++

```
Entity.ClassName ( );
```

The **ClassName** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.

Delete

This method deletes the entity object from the diagram. Note the entity interface itself is no longer valid after this operation.

Syntax

ANSI C

```
idoEntityDelete ( Entity );
```

Visual Basic

```
Entity.Delete
```

C++

```
Entity.Delete ( );
```

The **Delete** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.

GetContainedEntityCount

This method returns the number of entities contained within an entity acting as a container. If no entities are contained or the entity is not a container, this method fails and returns zero.

Syntax

ANSI C

```
Value = idoEntityGetContainedEntityCount ( Entity );
```

Visual Basic

```
Value = Entity.idoGetContainedEntityCount
```

C++

Value = *Entity*.**GetContainedEntityCount** ();

The **GetContainedEntityCount** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object acting as a container.
<i>Value</i>	A long which contains the number of entities contained within this entity.

See Also

IterateContainedEntityFirst, IterateContainedEntityNext,
AddEntityToContainer, RemoveEntityFromContainer

GetRelationInCount

This method returns the number of relations pointing to this particular entity. The determination of a relation “pointing to” an entity has nothing to do with the physical arrow displayed in the diagram. Rather, an internal distinction is made when the relation is created of which entity is the source entity and which is the destination entity. This method returns the count of the number of relations which consider this entity their destination.

Syntax

ANSI C

Value = **idoEntityGetRelationInCount** (*Entity*);

Visual Basic

Value = *Entity*.**GetRelationInCount**

C++

Value = *Entity*.**GetRelationInCount** ();

The **GetRelationInCount** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .
<i>Value</i>	A long containing the number of entities pointing to this entity.

See Also

IterateRelationInFirst, IterateRelationInNext, GetRelationOutCount,
IterateRelationOutFirst, IterateRelationOutNext

GetRelationOutCount

This method returns the number of relations pointing away from this particular entity. The determination of a relation “pointing away from” an entity has nothing to do with the physical arrow displayed in the diagram. Rather, an internal distinction is made when the relation is created of which entity is the source entity and which is the destination entity. This method returns the count of the number of relations which consider this entity their source.

Syntax

ANSI C

Value = **idoEntityGetRelationOutCount** (*Entity*);

Visual Basic

Value = *Entity*.**GetRelationOutCount**

C++

Value = *Entity*.**GetRelationOutCount** ();

The **GetRelationOutCount** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .
<i>Value</i>	A long containing the number of entities pointing away from this entity.

See Also

GetRelationInCount, IterateRelationInFirst, IterateRelationInNext, IterateRelationOutFirst, IterateRelationOutNext

IterateContainedEntityFirst

This prepares an entity acting as a container to allow a program to iterate a list of entities it contains. This method returns 0 if no entities are contained or the entity is not marked as a container.

Syntax

ANSI C

idoEntityIterateContainedEntityFirst (*Entity*);

Visual Basic

Entity.**IterateContainedEntityFirst**

C++

Entity.**IterateContainedEntityFirst** ();

The **IterateContainedEntityFirst** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object acting as a container.

See Also

IterateContainedEntityNext, GetContainedEntityCount, AddEntityToContainer, RemoveEntityFromContainer

IterateContainedEntityNext

This returns [an entity object](#) contained in the container entity. If there are no more entities contained this method fails.

Syntax

ANSI C

idoEntityIterateContainedEntityNext (*Entity*, *ContainedEntity*);

Visual Basic

ContainedEntity = *Entity*.**IterateContainedEntityNext**

C++

ContainedEntity = *Entity*.**IterateContainedEntityNext** ();

The **IterateContainedEntityNext** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object acting as a container.
<i>ContainedEntity</i>	An entity object returned from the container entity.

See Also

IterateContainedEntityFirst, **GetContainedEntityCount**,
AddEntityToContainer, **RemoveEntityFromContainer**

IterateRelationInFirst

This prepares an entity to allow a program to iterate a list of relations pointing to the entity. This method returns 0 if no relations point to this entity.

Syntax

ANSI C

idoEntityIterateRelationInFirst (*Entity*);

Visual Basic

Entity.**IterateRelationInFirst**

C++

Entity.**IterateRelationInFirst** ();

The **IterateRelationInFirst** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .

See Also

GetRelationInCount, **IterateRelationInNext**, **GetRelationOutCount**,
IterateRelationOutFirst, **IterateRelationOutNext**

IterateRelationInNext

This returns [a relation object](#) pointing to an entity. If there are no more relations pointing to the entity this method fails.

Syntax

ANSI C

idoEntityIterateRelationInNext (*Entity*, *Relation*);

Visual Basic

Relation = *Entity*.**IterateRelationInNext**

C++

Relation = *Entity*.**IterateRelationInNext** ();

The **IterateRelationInNext** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>Relation</i>	A relation object returned by InterAct.

See Also

GetRelationInCount, IterateRelationInFirst, GetRelationOutCount, IterateRelationOutFirst, IterateRelationOutNext

IterateRelationOutFirst

This prepares an entity to allow a program to iterate a list of relations pointing away from the entity. This method returns 0 if no relations point away from this entity.

Syntax

ANSI C

```
idoEntityIterateRelationOutFirst ( Entity );
```

Visual Basic

```
Entity.IterateRelationOutFirst
```

C++

```
Entity.IterateRelationOutFirst ( );
```

The **IterateRelationOutFirst** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.

See Also

GetRelationInCount, IterateRelationInFirst, IterateRelationInNext, GetRelationOutCount, IterateRelationOutNext

IterateRelationOutNext

This returns [a relation object](#) pointing away from an entity. If there are no more relations pointing away from the entity this method fails.

Syntax

ANSI C

```
idoEntityIterateRelationOutNext ( Entity, Relation );
```

Visual Basic

```
Relation = Entity.IterateRelationOutNext
```

C++

```
Relation = Entity.IterateRelationOutNext ( );
```

The **IterateRelationOutNext** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.

Relation [A relation object](#) returned by InterAct.

See Also

GetRelationInCount, IterateRelationInFirst, IterateRelationInNext,
GetRelationOutCount, IterateRelationOutFirst

PropertyPage

This displays the InterAct's built-in property page dialog for manipulating an entity in the diagram. The entity to edit can be identified by either its unique name or ID. If no ID or name is provided, the currently selected entity will be edited. If an entity cannot be identified by the name or ID provided, or no entity is currently selected, this method is ignored.

Syntax

ANSI C

```
idoEntityPropertyPage ( Entity, cPage );
```

Visual Basic

```
Entity.PropertyPage cPage
```

C++

```
Entity.PropertyPage ( cPage );
```

The **PropertyPage** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.
<i>cPage</i>	A constant which determines which page of the property sheet to initially display.

Settings

The settings for *cPage* are:

Setting	Description
IDO_PPE_TEXT	Initially displays the text property page.
IDO_PPE_COLORS	Initially displays the colors property page.
IDO_PPE_STYLES	Initially displays the styles property page.
IDO_PPE_GRAPHICS	Initially displays the graphics property page.
IDO_PPE_INTERNET	Initially displays the internet property page.

RemoveEntityFromContainer

This method removes an entity from another entity acting as a container. If the entity is not a container, this method fails. This method does not delete the entity, just mark it as no longer residing in a container. This method will not remove the entity from the boundaries of the container. The InterAct

container application must perform that action.

Syntax

ANSI C

```
idoEntityRemoveEntityFromContainer ( Container, Entity );
```

Visual Basic

```
Container.RemoveEntityFromContainer Entity
```

C++

```
Container.RemoveEntityFromContainer ( Entity );
```

The **RemoveEntityFromContainer** method syntax has these parts:

Part	Description
<i>Container</i>	An entity object acting as a container.
<i>Entity</i>	An entity object removed from the container.

See Also

IterateContainedEntityFirst, IterateContainedEntityNext,
GetContainerEntityCount, AddEntityToContainer

Repaint

This redraws an entity in a diagram. An entity is never repainted when a property is set. This allows you to set several properties without unnecessary screen flicker.

Syntax

ANSI C

```
idoEntityRepaint ( Entity );
```

Visual Basic

```
Entity.Repaint
```

C++

```
Entity.Repaint ( );
```

The **Repaint** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object .

UserID

This returns [a long](#) which contains the User ID associated with an entity when it was created. If the entity was created with the InterAct method **AddEntityFromClass**, the User ID was supplied with the method. If an entity was created in another means, then InterAct will supply a User ID.

Syntax

ANSI C

```
idoEntityUserID ( Entity );
```

```
Entity.id;
```

Visual Basic

Entity.UserID

C++

Entity.UserID ();

The **UserID** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.

UserName

This returns [a string](#) which contains the User Name associated with an entity when it was created. If the entity was created with the InterAct method **AddEntityFromClass**, the User Name was supplied with the method. If an entity was created in another means, then InterAct will supply a User Name.

Syntax

ANSI C

idoEntityUserName (*Entity*);

Entity.name;

Visual Basic

Entity.UserName

C++

Entity.UserName ();

The **UserName** method syntax has these parts:

Part	Description
<i>Entity</i>	An entity object.

Introduction

Properties are simple attributes exposed by an object like an entity to allow a program to tailor the way the entity will look or behave. Each relation exposes properties that allow you to specify colors for the relation, the type of line drawn, and symbols at the ends of the line, and various other properties.

InterAct is a collection of relation objects. These relation objects can have properties and methods invoked just like InterAct. For the ActiveX, invoke the method **Relation**, passing the unique ID and/or Name of a relation. If the relation could be located, [a relation object](#) with an interface to that relation will be returned. Using that object you can invoke properties and methods for that particular relation .

```
Dim Rel as Relation  
  
Rel = IDO.Relation 10, "Z"  
Rel.Text = "New Text"
```

Visual Basic Example of setting text using [a Relation Object](#).

```
IDO.Relation (1, "A").Text = "New Text"
```

Visual Basic Example of setting text without storing an Relation Object.

```
LPRELATION lpRel;  
  
lpRel = IDO.Relation(10, "Z");  
lpRel.SetText("New Day");
```

Visual C++ Example of setting text using an Relation Object.

```
IDO.Entity(10, "Z").SetText("New Text");
```

Visual C++ Example of setting text without storing an Relation Object.

For the DLL, a similar approach is used. The container application declares a variable of type **RELATION**. It then calls **idoGetRelation**, passing the window handle of InterAct and an ID or UserName to select a particular relation. If the specified relation is located, the variable of type **RELATION** will now refer to that relation. You can then tell the DLL to set a property or invoke a method using the variable reference to the relation without the need even for the window handle to InterAct.

```
RELATION relation;  
  
if(idoGetRelation(hIDO, 1, "A", &relation))  
    idoRelationSetText(&relation, "New Text");
```

ANSI C DLL example of setting text on a relation.

In the above code, first the DLL is told which relation to select for further processing using **idoGetRelation**. The programmer passes a reference to an relation structure. That structure is filled with all information needed to refer to an relation object. Next, the call **idoRelationSetText** is made. The relation reference stored in the structure is passed as a parameter to identify the particular relation to set the text for without having to be given the handle to InterAct or an relation ID or Name again.

BackColor

This property determines the color used to paint the relation.

Syntax

ANSI C

```
idoRelationSetBackColor ( Relation, value );
```

```
idoRelationGetBackColor ( Relation );
```

Visual Basic

```
Relation.BackColor [= value]
```

```
value = Relation.BackColor
```

C++

```
Relation.SetBackColor ( value );
```

```
value = Relation.GetBackColor ( );
```

The **BackColor** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.
<i>value</i>	A long value specifying a color for the relation.

Settings

The settings for *value* are:

Setting	Description
Color Value	A long value describing the color to paint the relation. This is often created using the RGB command.

See Also

TextColor

CanDelete

This property determines if a relation can be deleted by the end-user with the **Delete** key. This does not affect whether a relation can be deleted programmatically

Syntax

ANSI C

```
idoRelationSetCanDelete ( Relation, Value );
```

```
Value = idoRelationGetCanDelete ( Relation );
```

Visual Basic

```
Relation.CanDelete [= Value ]
```

```
Value = Relation.CanDelete
```

C++

```
Relation.SetCanDelete ( Value );
```


Value = *Relation*.**GetCanDelete**();

The **CanDelete** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>Value</i>	A Boolean which determines if the relation can be deleted.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The relation can be deleted.
FALSE	The relation cannot be deleted.

CanMoveEndPoints

This property determines if a relation's end-points can be moved by the user with the mouse.

Syntax

ANSI C

```
idoRelationSetCanMoveEndPoints ( Relation, Value );  
Value = idoRelationGetCanMoveEndPoints( Relation );
```

Visual Basic

```
Relation.CanMoveEndPoints [= Value ]  
Value = Relation.CanMoveEndPoints
```

C++

```
Relation.SetCanMoveEndPoints ( Value );  
Value = Relation.GetCanMoveEndPoints( );
```

The **CanMoveEndPoints** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>Value</i>	A Boolean which determines if the relation's end-points can be moved.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The relation's end-points can be moved.
FALSE	The relation's end-points cannot be moved.

CanMoveMidPoints

This property determines if a relation's mid-points can be moved by the user with the mouse. This property has no effect if the relation has no

midpoints.

Syntax

ANSI C

```
idoRelationSetCanMoveMidPoints ( Relation, Value );  
Value = idoRelationGetCanMoveMidPoints( Relation );
```

Visual Basic

```
Relation.CanMoveMidPoints [= Value ]  
Value = Relation.CanMoveMidPoints
```

C++

```
Relation.SetCanMoveMidPoints ( Value );  
Value = Relation.GetCanMoveMidPoints( );
```

The **CanMoveMidPoints** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.
<i>Value</i>	A Boolean which determines if the relation's mid-points can be moved.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The relation's mid-points can be moved.
FALSE	The relation's mid-points cannot be moved.

DestinationArrow

This property determines the type of arrow, if any, to draw at the destination point of a relation.

Syntax

ANSI C

```
idoRelationSetDestinationArrow ( Relation, value );  
value = idoRelationGetDestinationArrow( Relation );
```

Visual Basic

```
Relation.DestinationArrow [= value ]  
value = Relation.DestinationArrow
```

C++

```
Relation.SetDestinationArrow ( value );  
value = Relation.GetDestinationArrow ( );
```

The **DestinationArrow** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.

value A constant which defines the arrow to paint.

Settings

The settings for *value* are:

Setting

ARROWSTYLE_ NONE

ARROWSTYLE_ STANDARD

ARROWSTYLE_ NARROW

ARROWSTYLE_ WIDE

ARROWSTYLE_ SWEPT

ARROWSTYLE_ WHITECIRCLE

ARROWSTYLE_ BLACKCIRCLE

Description

No arrow is drawn at the destination point.



Font

This property determines the font used to draw the relation's text.

Syntax

ANSI C

```
idoRelationSetFont ( Relation, Font );  
Font = idoRelationGetFont ( Relation );
```

Visual Basic

```
Relation.Font [= Font ]
```

Font = *Relation*.**Font**

C++

Relation.**SetFont** (*Font*);

Font = *Relation*.**GetFont** ();

The **Font** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>Font</i>	A font object which describes the font used to paint the relation.

ReadOnly

This property determines if the text of a relation is read-only. This means the user cannot double-click the line to edit its text.

Syntax

ANSI C

idoRelationSetReadOnly (*Relation*, *Value*);

Value = **idoRelationGetReadOnly** (*Relation*);

Visual Basic

Relation.**ReadOnly** [= *Value*]

Value = *Relation*.**ReadOnly**

C++

Relation.**SetReadOnly** (*Value*);

Value = *Relation*.**GetReadOnly** ();

The **ReadOnly** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>Value</i>	A Boolean which determines if the relation text is read-only.

Settings

The settings for *Value* are:

Setting	Description
TRUE	The relation is read-only.
FALSE	The relation is not read-only.

SourceArrow

This property determines the type of arrow, if any, to draw at the source point of a relation.

Syntax

ANSI C

```
idoRelationSetSourceArrow ( Relation, value );
value = idoRelationGetSourceArrow ( Relation );
```

Visual Basic

```
Relation.SourceArrow [= value ]
value = Relation.SourceArrow
```

C++

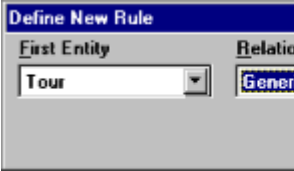

```
Relation.SetSourceArrow ( value );
value = Relation.GetSourceArrow ( );
```

The **SourceArrow** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.
<i>value</i>	A constant which defines the arrow to paint.

Settings

The settings for *value* are:

Setting	Description
ARROWSTYLE_NONE	No arrow is drawn at the source point.
ARROWSTYLE_STANDARD	
ARROWSTYLE_NARROW	
ARROWSTYLE_WIDE	
ARROWSTYLE_SWEPT	
ARROWSTYLE_WHITECIRCLE	

ARROWSTYLE_BLACKCIRCLE



Text

This property determines the text displayed in a relation.

Syntax

ANSI C

```
idoRelationSetText ( Relation, Text );  
Text = idoRelationGetText ( Relation );
```

Visual Basic

```
Relation.Text [= Text ]  
Text = Relation.Text
```

C++

```
Relation.SetText ( Text );  
Text = Relation.GetText ( );
```

The **Text** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.
<i>Text</i>	A string which contains the text for the relation.

TextColor

This property determines the color used to paint the relation text.

Syntax

ANSI C

```
idoRelationSetTextColor ( Relation, value );  
idoRelationGetTextColor ( Relation );
```

Visual Basic

```
Relation.TextColor [= value]  
value = Relation.TextColor
```

C++

```
Relation.SetTextColor ( value );  
value = Relation.GetTextColor ( );
```

The **TextColor** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.

<i>value</i>	A long value specifying a color for the relation text.
--------------	--

Settings

The settings for *value* are:

Setting	Description
Color Value	A long value describing the color to paint the relation relation. This is often created using the RGB macro.

See Also

BackColor

Thickness

This property determines the thickness of a line drawn representing a relation.

Syntax

ANSI C

```
idoRelationSetThickness ( Relation, value );
value = idoRelationGetThickness ( Relaton );
```

Visual Basic

```
Relation.Thickness [= value ]
value = Relation.Thickness
```

C++

```
Relation.SetThickness ( value );
value = Relation.GetThickness ( );
```

The **Thickness** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>value</i>	An integer value which specifies the thickness of the line drawn to represent the relation.

Settings

The settings for *value* are:

Setting	Description
1-5	A value which specifies the thickness of the line drawn to represent the relation.

Type

This property determines the type of line drawn for the relation.

Syntax

ANSI C

```

idoSetRelationType ( Relation, value );
value = idoGetRelationType ( Relation );

```

Visual Basic

```

Relation.Type [= value ]
value = Relation.Type

```

C++

```

Relation.SetType ( value );
value = Relation.GetType ( );

```

The **Type** property syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.
<i>value</i>	A constant which defines the shape of the relation.

Settings

The settings for *value* are:

Setting	Description
LINESTYLE _STRAIGHT	The relation is a straight line.
LINESTYLE _3PT	The relation has a midpoint. Straight lines are drawn between the endpoints and the midpoint.
LINESTYLE _4PT	The relation has two midpoints. Straight lines are drawn between the endpoints and the midpoints and between the midpoints.
LINESTYLE _90DEGREE	The relation has two midpoints. Lines are drawn from the endpoints to the midpoints at 90 degree angles to each other.

UserData

This property allows a container application to store any data with a relation. The UserData can be any 32-bit value ([a long](#)). This means any numerical data can be stored in the UserData, or even a pointer to an object or class.

Syntax

ANSI C

```

idoRelationSetUserData ( Relation, Value );
Value = idoEntityGetUserData ( Relation );

```

Visual Basic

```

Relation.UserData= Value
value = Relation.UserData

```


C++

```
Relation.UserData ( Value );
```

```
Value = Relation.UserData ( );
```

The **UserData** property syntax has these parts:

Part	Description
<i>Relation</i>	<u>A relation object.</u>
<i>Value</i>	<u>A long</u> which contains any value the container application wishes to associate with the relation.

Introduction

Methods are routines you can access to make a relation perform special actions. You can retrieve the source and destination entity, repaint it, or display a property page for it.

BringIntoView

This method scrolls the InterAct control to bring a relation into view. This function takes into consideration the current zoom mode of the diagram. This function has no effect if the relation is already visible. This function repaints the InterAct control if it is scrolled. This function returns TRUE if the InterAct control was scrolled and FALSE if the control was not scrolled.

Syntax

ANSI C

```
idoRelationBringIntoView ( Relation );  
Relation .classname;
```

Visual Basic

```
Relation .BringIntoView
```

C++

```
Relation .BringIntoView ( );
```

The **BringIntoView** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.

ClassName

This returns [a string](#) which contains the ClassName of the relation.

Syntax

ANSI C

```
idoRelationClassName ( Relation );  
Relation .classname;
```

Visual Basic

```
Relation .ClassName
```

C++

```
Relation .ClassName ( );
```

The **ClassName** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.

DestinationEntity

This returns the entity object which the relation is pointing to. The determination of a relation “pointing to” an entity has nothing to do with the physical arrow displayed in the diagram. Rather, an internal distinction is made when the relation is created of which entity is the source entity and which is the destination entity.

Syntax

ANSI C

```
idoRelationDestinationEntity ( Relation, Entity );
```

Visual Basic

```
Entity = Relation.DestinationEntity
```

C++

```
Entity = Relation.DestinationEntity ( );
```

The **DestinationEntity** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.
<i>Entity</i>	An entity object returned by the relation.

See Also

SourceEntity

Delete

This method deletes the relation object from the diagram. Note the relation interface itself is no longer valid after this operation.

Syntax

ANSI C

```
idoRelationDelete ( Relation );
```

Visual Basic

```
Relation.Delete
```

C++

```
Relation.Delete ( );
```

The **Delete** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object.

PropertyPage

This displays the InterAct's built-in property page dialog for manipulating a relation in the diagram. The relation to edit can be identified by either its unique name or ID. If no ID or name is provided, the currently selected relation will be edited. If a relation cannot be identified by the name or ID provided, or no relation is currently selected, this method is ignored.

Syntax

ANSI C

```
idoRelationPropertyPage ( Relation, cPage );
```

Visual Basic

```
Relation.PropertyPage cPage
```

C++

```
Relation.PropertyPage ( cPage );
```

The **PropertyPage** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>cPage</i>	A constant which determines which page of the property sheet to initially display.

Settings

The settings for *cPage* are:

Setting	Description
IDO_PPR_TEXT	Initially displays the text property page.
IDO_PPR_COLORS	Initially displays the colors property page.
IDO_PPR_STYLES	Initially displays the styles property page.

Repaint

This redraws a relation in a diagram. A relation is never repainted when a property is set. This allows you to set several properties without unnecessary screen flicker.

Syntax

ANSI C

```
idoRelationRepaint ( Relation );
```

Visual Basic

```
Relation.Repaint
```

C++

```
Relation.Repaint ( );
```

The **Repaint** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .

SourceEntity

This returns the entity object which the relation is pointing away from. The determination of a relation “pointing away from” an entity has nothing to do with the physical arrow displayed in the diagram. Rather, an internal distinction is made when the relation is created of which entity is the source entity and which is the destination entity.

Syntax

ANSI C

```
idoRelationSourceEntity ( Relation, Entity );
```

Visual Basic

```
Entity = Relation.SourceEntity
```

C++

Entity = *Relation*.**SourceEntity** ();

The **SourceEntity** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .
<i>Entity</i>	An entity object returned by the relation.

See Also

DestinationEntity

UserID

This returns [a long](#) which contains the User ID associated with a relation when it was created. If the relation was created with the InterAct method **AddRelationFromClass**, the User ID was supplied with the method. If a relation was created in another means, then InterAct will supply a User ID.

Syntax

ANSI C

```
idoRelationUserID ( Relation );  
Relation.id;
```

Visual Basic

```
Relation.UserID
```

C++

```
Relation.UserID ( );
```

The **UserID** method syntax has these parts:

Part	Description
<i>Relation</i>	A relation object .

UserName

This returns [a string](#) which contains the User Name associated with a relation when it was created. If the relation was created with the InterAct method **AddRelationFromClass**, the User Name was supplied with the method. If a relation was created in another means, then InterAct will supply a User Name.

Syntax

ANSI C

```
idoRelationUserName ( Relation );  
Relation.name;
```

Visual Basic

```
Relation.UserName
```

C++

```
Relation.UserName ( );
```

The **UserName** method syntax has these parts:

Part*Relation***Description**A relation object.

Introduction

Rules are objects which define behavior within a diagram. They limit instances where certain relations can be used to combine two entities. If rules are being enforced, InterAct will not allow a relation to combine two entities unless a rule exists to allow that combination. You can use methods on InterAct to add rules, delete rules, or iterate rules. A rule has three parts - A source entity class, a destination entity class, and a relation class which can be used to combine the two entity classes. Once you have [a rule object](#), you can use methods to determine the three parts of the rule.

DestinationEntityClassName

This returns [a string](#) which contains the ClassName of the destination entity.

Syntax

ANSI C

```
idoRuleDestinationEntityClassName ( Rule );  
Rule.E2;
```

Visual Basic

```
Rule.DestinationEntityClassName
```

C++

```
Rule.DestinationEntityClassName ( );
```

The **DestinationEntityClassName** method syntax has these parts:

Part	Description
<i>Rule</i>	A rule object.

RelationClassName

This returns [a string](#) which contains the ClassName of the relation which can be used to link the source entity and destination entity.

Syntax

ANSI C

```
idoRuleRelationClassName ( Rule );  
Rule.R1;
```

Visual Basic

```
Rule.RelationClassName
```

C++

```
Rule.RelationClassName ( );
```

The **RelationClassName** method syntax has these parts:

Part	Description
<i>Rule</i>	A rule object.

SourceEntityClassName

This returns [a string](#) which contains the ClassName of the source entity.

Syntax

ANSI C

```
idoRuleSourceEntityClassName ( Rule );  
Rule.E1;
```

Visual Basic

```
Rule.SourceEntityClassName
```

C++

Rule.SourceEntityClassName ();

The **SourceEntityClassName** method syntax has these parts:

Part	Description
<i>Rule</i>	<u>A rule object.</u>

Appendix A Code Samples

This appendix provides a single repository of code samples for InterAct. Whereever possible, samples are provided for ANSI C users of the DLL, and ActiveX examples for Visual C++ and Visual Basic users. Programmers using InterAct in other programming environments should be able to use one of the three samples provided, with minor modifications, in their programming environment.

Setting Fonts on an Entity

This sample shows how to set a font on an entity. First it retrieves an interface to an entity and gets the font properties for that entity. It then changes attributes of that font and sets it on the entity. This same example can be used with relation objects.

ANSI C DLL Example

```
LPIDO_FONT lpFont;
ENTITY entity;
HWND hIDO;

// get the window handle to InterAct
hIDO = GetDlgItem(hWnd, IDC_IDO1);

// get an entity interface
idoGetEntity(hIDO, 1, NULL, &entity);

// get a pointer to an InterAct allocated font object
lpFont = idoEntityGetFont(&entity);

// is this font bold?
if(!lpFont->bBold)
{
    // set this font to be bold
    lpFont->bBold = TRUE;

    // reset the entity font with the new information
    idoEntitySetFont(&entity, lpFont);

    // repaint the entity with the new font
    idoEntityRepaint(&entity);
}
```

Visual Basic ActiveX Example

```

Dim idoFont As Font
Dim e1 as Entity

Rem get an entity interface
Set e1 = ido1.GetEntity (1, "")

Rem get a interface to the entity's font object
Set idoFont = e1.Font

Rem is this font bold?
If ( idoFont.Bold = False) Then

    Rem set this font to be bold
    Font.Bold = True

    Rem reset the entity font with the new information
    e1.Font = idoFont

    Rem repaint the entity with the new font
    e1.Repaint

End If

```

C++ ActiveX Example

```

CFont idoFont;
CEntity entity;

// get an entity interface
Entity = m_ido.GetEntity(1, NULL);

// get a pointer to an InterAct allocated font object
idoFont = entity.GetFont();

// is this font bold?
if( !idoFont.GetBold() )
{
    // set this font to be bold
    idoFont.SetBold(TRUE);

    // reset the entity font with the new information
    entity.SetFont(idoFont);

    // repaint the entity with the new font
    entity.Repaint();
}

```

Iterating a List of Entities

This sample shows how to iterate a list of entities. First, it iterates all the entities and sets their background color to grey. It then sets the background color of selected entities to red.

ANSI C DLL Example

```

HWND hIDO;
ENTITY entity;

// get the window handle of InterAct
hIDO = GetDlgItem(hWnd, IDC_IDO1);

// initialize list of entities to traverse
if(idolterateEntityFirst(hIDO))
    // retrieve an interface to each entity in the diagram
    while(idolterateEntityNext(hIDO, &entity))
    {
        // set the BackColor of the entity to grey
        idoEntitySetBackColor(&entity, RGB(192,192,192));
        // repaint the entity
        idoEntityRepaint(&entity);
    }

// initialize list of selected entities
if(idolterateSelectedEntityFirst(hIDO))
    // retrieve an interface to each selected entity in the diagram
    while(idolterateSelectedEntityNext(hIDO, &entity))
    {
        // set the BackColor of the selected entity to red
        idoEntitySetBackColor(&entity, RGB(255, 0, 0));
        // repaint the entity
        idoEntityRepaint(&entity);
    }

```

Visual Basic ActiveX Example

```

Dim e1 As Entity

Rem initialize list of entities to traverse
If (ido1.IterateEntityFirst) Then
    Rem retrieve an interface to each entity in the diagram
    Set e1 = ido1.IterateEntityNext
    While (e1 Is Nothing = False)
        Rem set the BackColor of the entity to grey
        e1.BackColor = RGB(192,192,192)
        Rem repaint the entity
        e1.Repaint
        Set e1 = ido1.IterateEntityNext
    Wend
End If

Rem initialize list of selected entities
If (ido1.IterateSelectedEntityFirst) Then
    Rem retrieve an interface to each selected entity in the diagram
    Set e1 = ido1.IterateSelectedEntityNext
    While (e1 Is Nothing = False)
        Rem set the BackColor of the selected entity to red
        e1.BackColor = RGB(255, 0, 0)
        Rem repaint the entity
        e1.Repaint
        Set e1 = ido1.IterateSelectedEntityNext
    Wend
End If

```

C++ ActiveX Example

```

CEntity entity;

// initialize list of entities to traverse
if(m_ido.IterateEntityFirst())
    // retrieve an interface to each entity in the diagram
    while(entity = m_ido.IterateEntityNext())
    {
        // set the BackColor of the entity to grey
        entity.SetBackColor(RGB(192,192,192));
        // repaint the entity
        entity.Repaint();
    }

// initialize list of selected entities
if(m_ido.IterateSelectedEntityFirst())
    // retrieve an interface to each selected entity in the diagram
    while(entity = m_ido.IterateSelectedEntityNext())
    {
        // set the BackColor of the selected entity to red
        entity.SetBackColor(RGB(255, 0, 0));
        // repaint the entity
        entity.Repaint();
    }

```

Setting Properties on an Entity

This sample show how to set and retrieve various properties on [an entity object](#).

ANSI C DLL Example

```

HWND hIDO
ENTITY entity
char szText[256+1];
int cRight, cBottom;

// get the handle to InterAct
hIDO = GetDlgItem(hWnd, IDC_IDO1);

// get an interface to an entity "Node A"
idoGetEntity(hIDO, 0, "Node A", &entity);

// set the TextColor of the entity to red
idoEntitySetTextColor(&entity, RGB(255,0,0));

// set the 3D style to heavy outdent
idoEntitySetFrame(&entity, STYLE_3D_HEAVYOUTDENT);

// retrieve the right and bottom coordinates of the entity
cRight = idoEntityGetRight(&entity);
cBottom = idoEntityGetBottom(&entity);

// make the entity wider and taller
cRight += 10;
cBottom += 10;

// set the right and bottom coordinates of the entity
idoEntitySetRight(&entity, cRight);
idoEntitySetBottom(&entity, cBottom);

// get the text of the entity, copying to a local buffer
lstrcpy(szText, idoEntityGetText(&entity));

// concat text
lstrcat(szText, " more text");

// set the entities text with the longer text
idoEntitySetText(&entity, (LPSTR)szText);

// repaint the entity with these new values
idoEntityRepaint(&entity);

```

Visual Basic ActiveX Example

```

Dim entity as Entity
Dim Text as String
Dim cRight as Integer
Dim cBottom as Integer

Rem get an interface to an entity "Node A"
Set entity = ido1.GetEntity(0, "Node A")

Rem set the TextColor of the entity to red
entity.TextColor = RGB(255,0,0)

Rem set the 3D style to heavy outdent
entity.Frame = STYLE_3D_HEAVYOUTDENT

Rem retrieve the right and bottom coordinates of the entity
cRight = entity.Right
cBottom = entity.Bottom

Rem make the entity wider and taller
cRight = cRight + 10
cBottom = cBottom + 10

Rem set the right and bottom coordinates of the entity
entity.Right = cRight
entity.Bottom = cBottom

Rem get the text of the entity, copying to a local buffer
Text = entity.Text

Rem concat text
Text = Text + " more text"

Rem set the entities text with the longer text
entity.Text = Text

Rem repaint the entity with these new values
entity.Repaint

```

C++ ActiveX Example


```

CEntity entity
CString Text;
int cRight, cBottom;

// get an interface to an entity "Node A"
entity = m_ido.GetEntity(hIDO, 0, "Node A");

// set the TextColor of the entity to red
entity.SetTextColor(RED);

// set the 3D style to heavy outdent
entity.SetFrame(STYLE_3D_HEAVYOUTDENT);

// retrieve the right and bottom coordinates of the entity
cRight = entity.GetRight();
cBottom = entity.GetBottom();

// make the entity wider and taller
cRight += 10;
cBottom += 10;

// set the right and bottom coordinates of the entity
entity.SetRight(cRight);
entity.SetBottom(cBottom);

// get the text of the entity, copying to a local buffer
Text = entity.GetText();

// concat text
Text += "more text";

// set the entities text with the longer text
entity.SetText(LPCTSTR)Text.GetBuffer(0);

// repaint the entity with these new values
entity.Repaint();

```

Setting Properties on InterAct

This sample shows how to set and retrieve various properties on the InterAct object itself.

ANSI C DLL Example

```

HWND hIDO;
int cWidth, cHeight;
COLORREF color;

// get the window handle to InterAct
hIDO = GetDlgItem(hWnd, IDC_IDO1);

// set the property EditMode to make the diagram editable
idoSetEditMode(hIDO, TRUE);

// set the property ToolsPalette to make the tools palette visible
idoSetToolsPalette(hIDO, TRUE);

// set the property GridLines to make grid lines visible
idoSetGridLines(hIDO, TRUE);

// retrieve the GridWidth and GridHeight values
cWidth = idoGetGridWidth(hIDO);
cHeight = idoGetGridHeight(hIDO);

// alter the values for the GridHeight and GridWidth properties
cWidth += 10;
cHeight -= 5;

// set the new values for GridHeight and GridWidth
// InterAct will automatically be repainted with these grid dimensions
idoSetGridWidth(hIDO, cWidth);
idoSetGridHeight(hIDO, cHeight);

// store the color value for red in a variable
color = RGB(255, 0, 0);

// set the InterAct background to red
idoSetBackColor(hIDO, color);

```

Visual Basic ActiveX Example

```

Dim cWidth as Integer
Dim cHeight as Integer

Rem set the property EditMode to make the diagram editable
ido1.EditMode = TRUE

Rem set the property ToolsPalette to make the tools palette visible
ido1.ToolsPalette = TRUE

Rem set the property GridLines to make grid lines visible
ido1.GridLines = TRUE

Rem retrieve the GridWidth and GridHeight values
cWidth = ido1.GridWidth
cHeight = ido1.GridHeight

Rem alter the values for the GridHeight and GridWidth properties
cWidth = cWidth + 10
cHeight = cHeight + 5

Rem set the new values for GridHeight and GridWidth
Rem InterAct will automatically be repainted with these grid dimensions
ido1.GridWidth = cWidth
ido1.GridHeight = cHeight

Rem set the InterAct background to red
ido1.BackColor = RGB(255, 0, 0)

```

C++ ActiveX Example

```
int cWidth, cHeight;
COLORREF color;

// set the property EditMode to make the diagram editable
m_ido.SetEditMode(TRUE);

// set the property ToolsPalette to make the tools palette visible
m_ido.SetToolsPalette(TRUE);

// set the property GridLines to make grid lines visible
m_ido.SetGridLines(TRUE);

// retrieve the GridWidth and GridHeight values
cWidth = m_ido.GetGridWidth();
cHeight = m_ido.GetGridHeight();

// alter the values for the GridHeight and GridWidth properties
cWidth += 10;
cHeight -= 5;

// set the new values for GridHeight and GridWidth
// InterAct will automatically be repainted with these grid dimensions
m_ido.SetGridWidth(cWidth);
m_ido.SetGridHeight(cHeight);

// store the color value for red in a variable
color = RGB(255, 0, 0);

// set the InterAct background to red
m_ido.SetBackColor(color);
```

MFC ActiveX Example.

Adding and Defining a Class

This code sample shows how you can add classes to InterAct. Any classes added will also be reflected on the tools palette, if visible. It then shows how you can get an interface to the class to define attributes on the class. Finally, it shows how you can get these changes reflected in entities of that class which already exist in your diagram.

ANSI C DLL Example

```

ENTITY entityclass;
ENTITY entity;

// add the class type "toolbox"
idoAddEntityClass(hIDO, "Toolbox", "/bitmaps/toolbox.bmp", NULL);

// add an entity of the class "toolbox"
idoAddEntityFromClass(hIDO, 1, NULL, "Toolbox", "Text", 10, 10, 50, 50);

// get an interface to the entity "1"
idoGetEntity(hIDO, 1, NULL, &entity);

// set the back color to blue in the entity "1"
idoEntitySetBackColor(&entity, RGB(0, 0, 255));

// retrieve an interface to the "toolbox" class
idoGetEntityClass(hIDO, "Toolbox", &entityclass);

// set the back color to red in the class definition for "toolbox"
// note the back color of entity "1" is still blue
idoEntitySetBackColor(&entityclass, RGB(255, 0, 0));

// this causes the entity "1" to inherit changes to the class "toolbox"
// it will now have a red background
idoRedefineFromEntityClass(hIDO, "Toolbox");

```

Visual Basic ActiveX Example

```

Dim entityclass as Entity
Dim entity as Entity

Rem add the class type "toolbox"
ido1.AddEntityClass("Toolbox", "/bitmaps/toolbox.bmp", 0)

Rem add an entity of the class "toolbox"
ido1.AddEntityFromClass(1, NULL, "Toolbox", "Text", 10, 10, 50, 50)

Rem retrieve an interface to the "toolbox" class
Set entity = ido1.GetEntity(1, NULL)

Rem set the back color to blue in the entity "1"
entity.BackColor = RGB(0, 0, 255)

Rem retrieve an interface to the "toolbox" class
entityclass = ido1.GetEntityClass("Toolbox")

Rem set the back color to red in the class definition for "toolbox"
Rem note the back color of entity "1" is still blue
entityclass.BackColor = RGB(255, 0, 0)

Rem this causes the entity "1" to inherit changes to the class "toolbox"
Rem it will now have a red background
ido1.RedefineFromEntityClass("Toolbox");

```

C++ ActiveX Example

```
CEntity entityclass;
CEntity entity;

// add the class type "toolbox"
m_ido.AddEntityClass("Toolbox", "/bitmaps/toolbox.bmp", NULL);

// add an entity of the class "toolbox"
m_ido.AddEntityFromClass(1, NULL, "Toolbox", "Text", 10, 10, 50, 50);

// retrieve an interface to the "toolbox" class
entity = m_ido.GetEntity(1, NULL);

// set the back color to blue in the entity "1"
entity.SetBackColor(RGB(0, 0, 255));

// retrieve an interface to the "toolbox" class
entityclass = m_ido.GetEntityClass("Toolbox");

// set the back color to red in the class definition for "toolbox"
// note the back color of entity "1" is still blue
entityclass.SetBackColor(RGB(255, 0, 0));

// this causes the entity "1" to inherit changes to the class "toolbox"
// it will now have a red background
m_ido.RedefineFromEntityClass("Toolbox");
```

An InterAct object

This is any programmatic way of referring to InterAct, also referred to as an Interactive Diagramming Object (IDO). In C, this will be a window handle. In C++, it may be a C++ class or an ActiveX object. For Visual Basic, it will always be an ActiveX object.

An entity object

This is any programmatic way of referring to an entity contained in InterAct. In C, this will be a structure (ENTITY). In C++, it may be a C++ class or an ActiveX object. For Visual Basic, it will always be an ActiveX object.

A relation object

This is any programmatic way of referring to a relation contained in InterAct. In C, this will be a structure (RELATION). In C++, it may be a C++ class or an ActiveX object. For Visual Basic, it will always be an ActiveX object.

A rule object

This is any programmatic way of referring to an InterAct rule. In C, this will be a structure (IDO_RULE). In C++, it may be a C++ class or an ActiveX object. For Visual Basic, it will always be an ActiveX object.

A font object

This is any programmatic way of referring to an InterAct font. In C, this will be a structure ([IDO_FONT](#)). In C++, it may be a C++ class or an ActiveX object. For Visual Basic, it will always be an ActiveX object.

A string

A data type to represent strings. For the DLL, it is an array of **chars**. For the ActiveX in Visual Basic, it is the data type **String**. For Visual C++ it is a **CString** object.

A short

A data type to represent short integers. For the DLL, it is the data type **short**. For the ActiveX in Visual Basic, it is the data type **Short**. For Visual C++ it is the data type **short**.

A handle

A data type to represent local handles. For the DLL, it is the data type **hWnd**. For the ActiveX in Visual Basic, it is the data type **OLE_HANDLE**. For Visual C++ it is the data type **OLE_HANDLE**.

A long

A data type to represent long integers. For the DLL, it is the data type **long**. For the ActiveX in Visual Basic, it is the data type **Long**. For Visual C++ it is the data type **long**.

A constant

A numeric identifier which has a preset value. Constants are used to convey a symbolic name to a predefined number to allow for more readable code and portability.

An integer

A data type to represent short integers. For the DLL, it is the data type **int**. For the ActiveX in Visual Basic, it is the data type **Integer**. For Visual C++ it is the data type **int**.

A Boolean

A data type to represent logical expressions. For the DLL, it is the data type **BOOL**. For the ActiveX in Visual Basic, it is the data type **Boolean**. For Visual C++ it is the data type **BOOL**.

IDO RULE

An IDO_RULE is defined as follows:

```
char E1[99+1];
```

```
char R1[99+1];
```

```
char E2[99+1];
```

```
int MaxCount;
```

IDO_FONT

An IDO_FONT is defined as follows:

char szFontName;

int cSize;

BOOL bBold;

BOOL bItalic;

BOOL bUnderline;

BOOL bStrikeout;

ENTITY

An ENTITY is defined as follows:

char name;

long id;

char classname;

RECT rect;

char text;

HWND hIDO;

RELATION

A RELATION is defined as follows:

char name;

long id;

char classname;

char text;

long Src_id;

char Src_name[99+1];

long Dst_id;

char Dst_name[99+1];

HWND hIDO;

THREE D STYLE

STYLE_3D_NONE

STYLE_3D_OUTDENT

STYLE_3D_HEAVY_OUTDENT

STYLE_3D_INDENT

STYLE_3D_HEAVY_INDENT

STYLE_3D_SHADOW

ENTITYSHAPE

ENTITYSHAPE_CIRCLE

ENTITYSHAPE_RECTANGLE

ENTITYSHAPE_ROUNDRECT

ARROWSTYLE

ARROWSTYLE_NONE

ARROWSTYLE_STANDARD

ARROWSTYLE_NARROW

ARROWSTYLE_WIDE

ARROWSTYLE_SWEEPED

ARROWSTYLE_WHITECIRCLE

ARROWSTYLE_BLACKCIRCLE

LINESTYLE

LINESTYLE_STRAIGHT

LINESTYLE_90DEGREE

LINESTYLE_3PT

LINESTYLE_4PT

TEXTPOSSTYLE

TEXTPOS_BOTTOM

TEXTPOS_TOP

TEXTPOS_LEFT

TEXTPOS_RIGHT

RELATIVEPOSITION

RELATIVEPOS_NULL

RELATIVEPOS_LEFT

RELATIVEPOS_RIGHT

RELATIVEPOS_UP

RELATIVEPOS_DOWN

RELATIVEPOS_UPLEFT

RELATIVEPOS_UPRIGHT

RELATIVEPOS_DOWNLEFT

RELATIVEPOS_DOWNRIGHT



InterAct Help Reference

[License Agreement](#)

[What is InterAct?](#)

[Installing InterAct](#)

[Product Support](#)

[InterAct File Listing](#)



[Getting Started with InterAct](#)

Introduction to the InterAct control. The InterAct environment and how to modify it are explained.



[InterAct Tutorials](#)

Step-by-step tutorials for Visual Basic, Visual C++ and ProtoGen+.



[Programming InterAct](#)

Complete reference for programming InterAct. Events and interfaces for InterAct, entities and relations are categorized.

[Appendix A - Coding Samples](#)



Getting Started with

InterAct

[A Guided Tour of InterAct](#)

[The InterAct Environment](#)

[Property Pages](#)

[The Tools Palette](#)

[Classes and Rules](#)

-



A Guided Tour of

InterAct

[Introduction](#)

[Creating Diagrams](#)

[Modifying Objects in a Diagram](#)

[Working with the Tools Palette](#)

[Exploring Classes and Rules](#)

[Summary](#)

-



The InterAct

Environment

[Overview](#)

[How to Modify the InterAct Environment](#)

[Summary](#)

-



Property Pages

[Introduction](#)

[InterAct Property Pages](#)

[Entity Property Pages](#)

[Relation Property Pages](#)

[Summary](#)

-



The Tools Palette

[Overview](#)

[Customizing the Tools Palette](#)

[Summary](#)



Classes and Rules

[Introduction](#)
[Entity Classes](#)
[Relation Classes](#)
[Rules](#)
[Summary](#)



InterAct Tutorials

[A Guided Tour of InterAct](#)

[Visual C++ Tutorial \(ActiveX Version\)](#)

[Visual Basic Tutorial \(ActiveX Version\)](#)

[ProtoGen+ Tutorial \(DLL Version\)](#)



Visual C++ Tutorial

(ActiveX Version)

[Introduction](#)

[Adding InterAct to an Application](#)

[Installing InterAct in Microsofts Developer Studio](#)

[Adding InterAct to a Dialog](#)

[Programming InterAct](#)

[Responding to Events](#)

[Conclusion](#)



Visual Basic Tutorial

(ActiveX Version)

[Introduction](#)

[Installing InterAct in Visual Basic](#)

[Adding InterAct to a Form](#)

[Programming InterAct](#)

[Responding to Events](#)

[Conclusion](#)



ProtoGen+ Tutorial

(DLL Version)

[Introduction](#)

[Adding InterAct to an Application](#)

[Installing InterAct in ProtoViews ViewPaint](#)

[Adding InterAct to a Dialog](#)

[Programming InterAct](#)

[Responding to Events](#)

[Conclusion](#)



Programming

InterAct

[Programming InterAct](#)

[InterAct Events](#)

[InterAct Properties](#)

[InterAct Methods](#)

[Entity Properties](#)

[Entity Methods](#)

[Relation Properties](#)

[Relation Methods](#)

[Rule Methods](#)



Programming the InterAct Control

[Introduction](#)

[Identifying Objects](#)

[InterAct Architecture](#)

[Using Fonts](#)

[Using Pictures](#)

[Programming with the DLL](#)

[Programming with the OCX](#)



InterAct Events

[Introduction](#)

[Responding to Events from the DLL](#)

[Standard Events](#)

[Diagram Events](#)

[Palette Events](#)

[Tools Palette Events](#)

[Dialog Events](#)

[Mouse Events](#)

[Entity Events](#)

[Relation Events](#)



InterAct Properties

Introduction

BackColor

Border

CurrentEntity

CurrentRelation

EditMode

FileName

GridHeight

GridLineColor

GridLines

GridWidth

HorizontalScrollOffset

hWnd

InetPath

Modified

PopupMenu

PrinterLandscape

PrinterLines

Redraw

RulesEnforced

ScrollBars

SnapToGrid

ToolsPalette

VerticalScrollOffset

WindowStyle

ZoomValue



InterAct Methods

Introduction

[AddEntityClass](#)
[AddEntityFromClass](#)
[AddRelationClass](#)
[AddRelationFromClass](#)
[AddRule](#)
[AllowAction](#)
[CancelAction](#)
[Copy](#)
[Cut](#)
[DeleteEntity](#)
[DeleteEntityClass](#)
[DeleteRelation](#)
[DeleteRelationClass](#)
[DeleteRule](#)
[DoesEntityExist](#)
[DoesEntityClassExist](#)
[DoesRelationExist](#)
[DoesRelationClassExist](#)
[DoesRuleExist](#)
[DragAddEntity](#)
[DragAddRelation](#)
[GetEntity](#)
[GetEntityClass](#)
[GetNotifyEntity](#)
[GetNotifyRelation](#)
[GetNumberOfSelectedEntities](#)
[GetNumberOfEntities](#)
[GetNumberOfRelations](#)
[GetRelation](#)
[GetRelationClass](#)
[GetVersion](#)
[IterateEntityClassFirst](#)
[IterateEntityClassNext](#)
[IterateEntityFirst](#)
[IterateEntityNext](#)
[IterateRelationClassFirst](#)
[IterateRelationClassNext](#)
[IterateRelationFirst](#)

IterateRelationNext
IterateRuleFirst
IterateRuleNext
IterateSelectedEntityFirst
IterateSelectedEntityNext
ManageClasses
ManageRules
MapDeviceToLogical
MapLogicalToDevice
Paste
PrintDiagram
PropertyPage
ReadDiagram
ReadPalette
RedefineFromEntityClass
RedefineFromRelationClass
ResetDiagram
ResetPalette
SaveDiagram
SavePalette
SetToolsPaletteButtonText
Zoom



Entity Properties

Introduction

AutoResize

BackColor

Border

Bottom

CanDelete

CanMove

CanResize

Container

CoordX

CoordY

Font

Frame

Graphic

InetHost

InetUrl

Left

ReadOnly

Rect

Right

Select

Shape

StretchBitmap

Text

TextColor

TextOrientation

Top

Transparent

UserData

ValidRelationDragSource



Entity Methods

Introduction

AddEntityToContainer

BringIntoView

ClassName

Delete

GetContainedEntityCount

GetRelationInCount

GetRelationOutCount

IterateContainedEntityFirst

IterateContainedEntityNext

IterateRelationInFirst

IterateRelationInNext

IterateRelationOutFirst

IterateRelationOutNext

PropertyPage

RemoveEntityFromContainer

Repaint

UserID

UserName



Relation Properties

Introduction

BackColor

CanDelete

CanMoveMoveEndPoints

CanMoveMoveMidPoints

DestinationArrow

Font

ReadOnly

SourceArrow

Text

TextColor

Thickness

Type

UserData



A screenshot of a 'Define New Rule' dialog box. It has three dropdown menus: 'First Entity' with 'Tour' selected, 'Relationship' with 'Generic Relation' selected, and 'Second Entity' with 'Tour' selected. At the bottom are 'Apply' and 'Cancel' buttons.

Relation Methods

Introduction

[BringIntoView](#)

[ClassName](#)

[DestinationEntity](#)

[Delete](#)

[PropertyPage](#)

[Repaint](#)

[SourceEntity](#)

[UserID](#)

[UserName](#)



A screenshot of a 'Define New Rule' dialog box. It has three dropdown menus: 'First Entity' with 'Tour' selected, 'Relationship' with 'Generic Relation' selected, and 'Second Entity' with 'Tour' selected. At the bottom are 'Apply' and 'Cancel' buttons.

Rule Methods

Introduction

DestinationEntityClassName

RelationClassName

SourceEntityClassName

Warning

This property, method or event does not exist in the help file. You should consult the latest README information that came with your InterAct product for a description of this interface.

