

Introduction to Event-Driven Programming

The following topics are covered below:

- What is an Event-Driven Application?
- GUI Development Environments
- GUI Design Tips
- Tasks Involved in Creating an Application
- Tutorial - Overview
- Creating a Dialog
- Assigning Attributes to the Dialog
- Creating Dialog Elements Inside the Dialog
- Assigning Attributes to the Dialog Elements
- Creating the Application's Local Data Area
- Attaching Event Handler Code to the Dialog Element
- Checking, Stowing and Running the Application
- Basic Terminology

For further information on Event-driven Programming see Event-Driven Programming Techniques.

What is an Event-Driven Application?

- Program-Driven Applications
- Event-Driven Applications
- What is Happening Here?
- Writing Event-Driven Code
- Components of an Event-Driven Application

Event-driven applications represent a new approach to development in addition to the program-driven approach. Natural offers you both. Event-driven programming allows the application to be driven by input received through the graphical user interface.

In program-driven applications, the application controls the portions of code that execute - not an event. Execution starts with the first line of executable code and follows a defined pathway through the application, calling additional programs as instructed in the predetermined sequence.

In event-driven programming, the user's action or a system event triggers the code attached to that event. Thus, the order in which your code executes depends on which events occur, which in turn depends on what the user does. This is the essence of graphical user interfaces and event-driven programming: The user is in charge, and the code responds. Even though event-driven programming is possible in character-oriented interfaces, it is more common in graphical user interfaces.

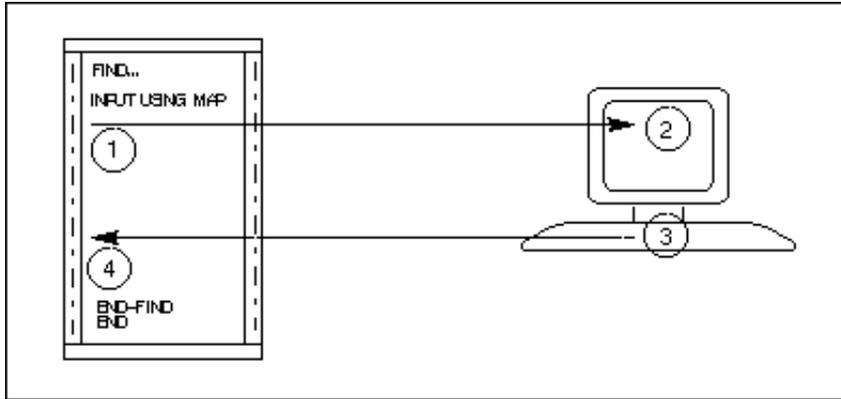
Because you cannot predict what the user will do, your code must make a few assumptions when it executes. For example, the application might assume that the user added text to an edit-area control before pressing the OK button.

When you must make assumptions, you should try to structure your application so these assumptions are always valid. For example, to ensure the user added text, you can disable the button and enable it only when the change event occurs for the edit-area control.

Your code can trigger additional events as it performs certain operations. For example, moving the slider in a scroll-bar control triggers the change event.

The following diagrams illustrate the difference between program-driven and event-driven applications.

Program-Driven Applications

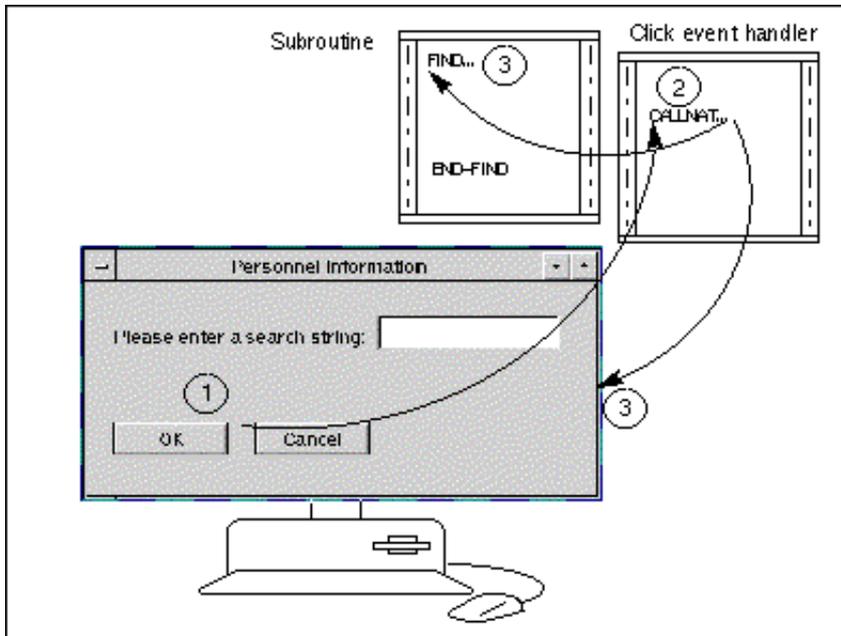


In typical program-driven applications, the following sequence of steps applies:

1. The program sends a screen to the terminal.
2. The user reacts by filling in the data fields.
3. The user then presses ENTER or a function key.
4. The program then decides whether or not the user's entries are valid.

If the data are valid, it processes the results until it reaches an END statement.

Event-Driven Applications



In typical event-driven applications, the following sequence of steps applies:

1. The user requests an action on the screen.
2. The event handler code reacts in the background according to the context.
3. If certain conditions are fulfilled, the executed event handler code triggers other Natural code (here: a subroutine) or returns control to the screen.

In the program-driven approach, the user interacts with the code through the ENTER and function keys, the user of an event-driven application triggers specific pieces of code (event handlers). Typically, an event-driven application is not executing any code when waiting for user input; in the same situation, the program-driven application might be processing an INPUT statement.

What is Happening Here?

Graphical user interface programs require you to write programs that react to isolated events initiated by the user.

An event is an action recognized by a dialog or a dialog element. Event-driven applications execute code in response to an event. Each dialog or dialog element has a predefined set of events. If one of these events occurs, Natural invokes the code in the associated event handler.

You decide if and how the dialogs and dialog elements in your application respond to a particular event. When you want a program to respond to an event, you write event code for that event.

Writing Event-Driven Code

For each dialog or dialog element you create, Natural predefines a set of events to which your program (event handler) can respond. It is easy to respond to events: dialogs and dialog elements have the built-in ability to recognize user actions and execute the code associated with them.

You do not have to write code for all events. When you do want a dialog object to respond to an event, you write event code that Natural executes in response to that event.

In a typical event-driven application, the following series of actions takes place:

- A dialog or dialog element recognizes an action as an event. The action can be caused by the user (such as a click or keystroke).
- If there is event code corresponding to the event, it is executed.
- The application waits for the next event.

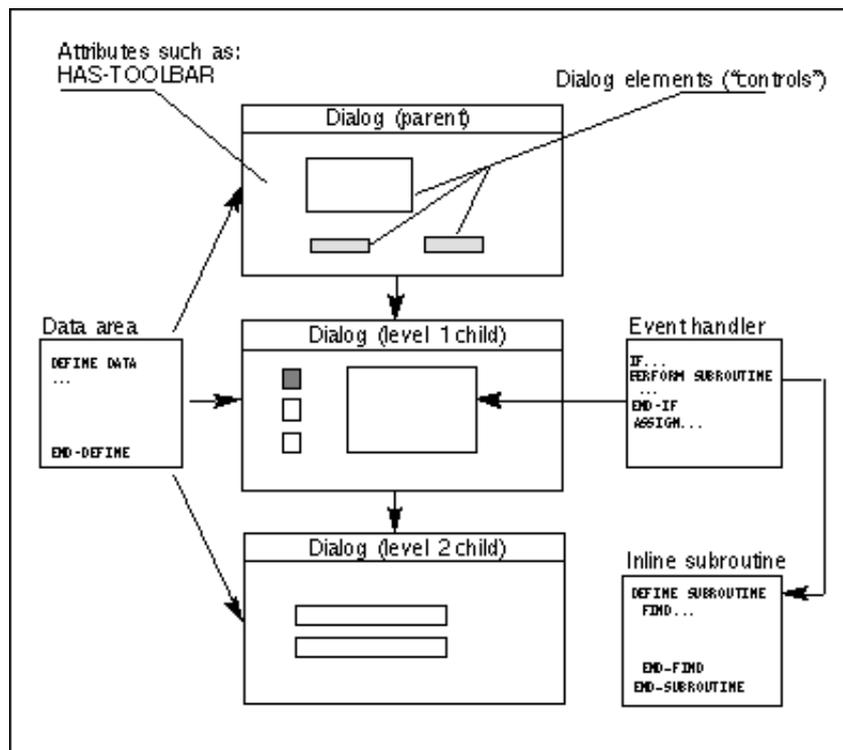
The event code you write to respond to events can perform calculations, get input, and manipulate parts of the interface. Using Natural, you manipulate dialogs or dialog elements by changing the values of their attribute settings.

Note:

Avoid creating cascading events in your code caused by events occurring repeatedly. For example, when the user drags the slider in the scroll-bar control, the current SLIDER attribute setting is automatically changed and the change event is triggered. If the code attached to the change event also changes the current SLIDER attribute setting, then the change event is triggered again, the current SLIDER attribute setting is again adjusted, the change event is once again triggered, and so on. At this rate, you quickly run out of memory.

Components of an Event-Driven Application

Overview



Dialogs

The dialog is the central Natural object in an event-driven application. An event-driven application is started by running or executing the base dialog. This may open other dependent dialogs when the OPEN DIALOG statement is specified. As opposed to program-driven applications, these dialogs are usually modeless, that is, all open dialogs can be processed concurrently by the end user. The application terminates when the base dialog is closed.

You create a dialog with the dialog editor. Just like the map editor, the dialog editor assembles a Natural object from the specification of the dialog window and its dialog elements, the global data area (GDA), the local data areas (LDAs), the parameter data areas (PDAs), the subroutines and the specified event handler sections.

At runtime of the dialog, there is a difference between the runtime instance identified by the system variable *DIALOG-ID and the GUI instance (handle) of the dialog window (the default handle name is #DLG\$WINDOW).

Whenever you want to work with more than one dialog in your application, you must decide how the base dialog window relates to the other dialogs. First you have to decide whether the application should be MDI (Multiple Document Interface) or not.

If you have opted for an MDI application, the base dialog must be of the type "MDI frame window" and the dependent dialogs must be of the type "MDI child window" and "Standard window".

If you have opted for non-MDI, the application may contain only dialogs of the type "Standard window".

Dialogs of type "Standard window" can have the styles Popup, Modal or Dialog Box.

Dialog Elements

Almost all dialog elements are graphical elements inside a dialog that allow the end user to interact with the event-driven application. After a dialog has been opened with the dialog editor and its attributes have been set (see below), the programmer will go on to "draw" the dialog elements inside the window; usually, this comprises a menu control, possibly a toolbar, and other elements, such as push-button controls, input-field controls.

"Drawing" a dialog element means that you select the type of dialog element from the dialog editor's menu or toolbar, and use the mouse to place it at the desired location. It is also possible to define a grid where the dialog elements can be placed more conveniently by aligning them to the grid.

Attributes

Attributes are the properties of dialogs and dialog elements. After creating a dialog or dialog element, you double-click with the mouse on it and the window with the corresponding attributes appears. You can then set the attributes to a value; if not, they remain at the system default value. The attributes window also contains a push-button control that opens up the event handler window.

Event Handlers

The event handlers represent the Natural code that is triggered when an event occurs. A click event occurs, for example, when the end user clicks on a push-button control. Inside the event handler window, you must first select the type of event from the list of events available for the dialog or dialog element (the one whose attributes have just been set). Then, the code window is enabled and Natural code can be entered.

Data Areas - Global, Local, Parameter

- A global data area (GDA) is used to share data fields between Natural objects within the application. One GDA per application may be specified.
- A local data area (LDA) contains the data fields private to the dialog.
- A parameter data area (PDA) is always present in dialogs. It is used to pass parameters to a dialog in the OPEN DIALOG or SEND EVENT statements. In these statements, parameters are passed either by specifying their name (WITH clause), or by listing parameters one after the other. You can use the dialog editor PDA window to type in your PDA in free-form style or to include PDAs defined externally.

Inline Subroutines

An inline subroutine defines standard code to be used for a frequently needed task called by a number of event handlers. You access an inline subroutine window via the "Inline Subroutines" push-button control in the dialog window.

GUI Development Environments

To understand the functions of Natural, you must first understand the environment in which it runs.

A graphical user interface (GUI) environment differs from a traditional mainframe environment in at least two important ways:

- Applications share screen space. A Natural application runs in a group of one or more windows and rarely occupies the full screen.
- Applications share computing time. An application cannot run continually, or if it does, it must run in the background.

Using Natural, your applications share computing time and other resources (such as the clipboard). An event-driven application consists of dialogs and dialog elements that wait for a particular event to happen.

While your application is waiting to execute an event, it remains on the desktop (unless the user closes the application). In the meantime, the user can run other applications, resize windows, or customize system settings (such as color). However, your code is always present, ready to be activated when the user returns to your application.

GUI Design Tips

- Do Your Research
- Screen Design
- Menu Design
- Color Usage
- Consistency Check

Designing the screens for a GUI application requires different knowledge than designing the 3270 screens for a mainframe. Why is it different?

It is different, because GUI applications put the users in control; these applications are non-modal and unstructured. The users choose the order in which they access windows, and fields within the windows. Traditional database applications often require the users to perform operations in a specific order; these applications are form-oriented and structured.

Designing a GUI screen is also different, because the GUI interface has different capabilities than a traditional mainframe interface. You can design windows that incorporate dialog elements, such as push-button controls and list-box controls. As you design your GUI windows, which are called dialogs in event-driven Natural, you define the font type and size of the text, the background and foreground colors, and the size of each window.

The following sections provide some tips for effective GUI design.

Do Your Research

- Spend a few hours with your users before prototyping.

A couple of sessions with your users to iron out their needs, likes, and dislikes is enough to give you to a good basis for beginning your design.

- Take some ideas from existing GUI designs.
Save time by not re-inventing the GUI. Try out other GUIs with an eye for what works and what does not. Consistency within GUIs helps users learn to use new applications, improves efficiency, and reduces training costs. Get user feedback on existing GUI applications - listen to their likes and dislikes rather than develop a prototype that replicates the weaknesses of poor GUI design.
- Develop your ideas on paper before spending time developing the application online.
It is faster for you to run through a number of screen design options for your main windows on paper before spending time to create multiple prototypes online. It is quicker than coding and you do not become attached to poor designs.

If you include your users in the development process, they can quickly comment about their needs and likes before the application is installed in the system. Try to use a paper prototype before reaching for the online development tool.

Screen Design

- Design multiple windows for related subject matter.
Unlike designing for 3270 monitors, where you try to maximize the number of fields per screen, GUI screens are better designed using subwindows. You can, for example, have the essential fields in the main window, and all optional or supplemental information stored in one or more subwindows. Subwindows can include choices, such as drop-down lists, for the user to browse through if they do not know the information to input into the main window. Messages and field-dependent information are more effectively presented in supplemental windows than in the main window.
- Design clear, uncluttered windows.
Avoid cluttering your windows with more than three colors, multiple graphics, and a variety of shapes. Balance your objects on the screen with lots of white space so users are not overwhelmed by variety and distracted by the presentation. Try to keep shapes and objects to a minimum and the number of colors low.
- Design accessible, not overwhelming, windows.
Multiple fonts, font sizes, font types or families, and color schemes can overwhelm your users, making your application seem inaccessible to them. Use a maximum of three fonts, font sizes, and font types per window. Avoid using italics and serif fonts because they often break up on the screen. Use color sparingly. Neutral colors are kindest to your users eyes. Though vibrant reds and greens are very eye-catching, remember that your users spend a lot of their day working in the windows you design.
- Design for both keyboard and mouse use.
Some users prefer using the keyboard and memorize the short cut commands, while other users are more comfortable using the mouse. Each action should be accessible by both the mouse and the keyboard.
- Design the windows according to your users' needs.
Though it is tempting to create fabulous-looking screens with lots of functionality, if your users do not use it, it is of no value. Remember that you are designing the application for your users to get a job done, not for you to experiment with all the functionality you have available. First find out what your users need, then tailor your design to meet their needs. You design screens with different purposes in different ways. If you want to prompt the user, you use a conversational style; if you want the user to enter values from a form, you use a data-entry style.

Conversational Screens

- Design conversational screens with field prompts.
In a conversational-based style, users enter data from a conversation (travel reservations, for example). Conversational-based styles, in which the user relies on the screen for prompting, can be rich with labels, hints, instructions, and even questions for the users to ask their clients.

Data-Entry Screens

- Design data-entry screens with terse labels.
In a form-based style, users enter data from a form. Each line on the input screen must match a line on the form - and the lines must be in the same order. To maintain a line-for-line correspondence, you can abbreviate labels. Headings and instructions are kept to a minimum. The only purpose of labels is to help users find their places again after interruptions.

Menu Design

The following three criteria are recommended for designing menus.

- Organize menus using the conventions defined by the operating system on which your users run the application. Microsoft Windows, for example, recommends certain menus ("File", "Edit", and "View", for example), options on menus ("Cut", "Copy", and "Paste" on the "Edit" menu, for example), and a particular order of the menus on the menu bar (Help always appears at the right margin, for example).
- Arrange menus by frequency of use and decide this information through observation or usability testing. Anticipate whether usage changes as users become more expert. Watch that this does not violate conventions established for the operating system.
- List menu items alphabetically.
Remember to follow the operating-system conventions and user recommendations for frequency of using menu items.

Color Usage

- Be as conservative as possible with color.
Humans can remember the meaning of no more than five colors at a time, plus or minus two.
- Use color as an additional signal, not as the primary signal.
Using bright red text to warn a user is not enough; add a warning tone. Eight percent of all males are red-green color-blind and may not notice the red text.
- On charts, do not use colors without adding a secondary key (for example, a broken or solid underline).
Users with black and white monitors must be able to understand the key without the benefit of color. Also, most users do not have color printers.

Consistency Check

- Be consistent throughout the application.
Do not change fonts, colors, or shapes for related subjects. For example, design all the OK buttons in an application with the same shape, size, color, and font. If related objects are presented in different ways, users cannot use the visual clues, taking them longer to become comfortable with the application. Present similar actions in a similar way, using the same font, color, and size for related buttons.
- Adopt a naming convention (and stick with it throughout the application).
While traditional programs tend to have one large program you modify for a name change, object-oriented programs have numerous pieces of event code that you must edit individually every time you make a name change. When you design GUI applications, you must be much more rigorous about sticking to naming conventions. This avoids a lot of cleaning up time later.

Tasks Involved in Creating an Application

There are a number of main tasks you perform to create an application in event-driven Natural. The order in which they are explained in this section is the typical order in which you perform them. However, this sequence is not inflexible. For example, you may very well test a dialog several times in the process of designing it, and you will no doubt save your work more often during the development process.

- Decide whether your application is Multiple Document Interface or Single Document Interface.
- Create one or more dialogs.
- Set the attributes of the dialog(s).
- Create and place dialog elements in the dialog(s).
- Set the attributes of the dialog elements.
- Define the tab order in each of the dialogs (from the menu, choose "Dialog > Control Sequence").
- Save the dialog(s) to a name.
- Define the global data area.
- Define the local data area(s).
- Write event handler code for the dialog(s).
- Write inline subroutines for the dialog(s).
- Write event handler code for the dialog elements.
- Stow the dialog(s).
- Test (check and run) the dialog(s).
- Execute the application.

The following short tutorial introduces you to the most frequently performed tasks.

Tutorial - Overview

This section is a simple tutorial that demonstrates how to add the components of an event-driven application one after the other. The tutorial describes how to develop a small sample application consisting of one dialog. The application you will create is a degressive depreciation calculator.

You can use this calculator, for example, to find out the value of your car by entering how much the car was worth when you bought it, how many years you have owned it, and the percentage by which the value of the car decreases each year.

You can save your application at any stage, allowing you to interrupt the tutorial and continue at a later time where you left.

To develop the sample application

1. Create a new dialog (represented by a window).
2. Assign the attributes to your dialog (decide the window's settings).
3. Create the dialog elements in the dialog (decide how the user can interact).
4. Assign the attributes to your dialog elements (decide attribute settings).
5. Create the application's local data area (define the variables that allow the event handler to use the end user's numeric input).
6. Attach event handler code to the dialog element (decide what happens at runtime when the user interacts).
7. Check, stow and run the application.

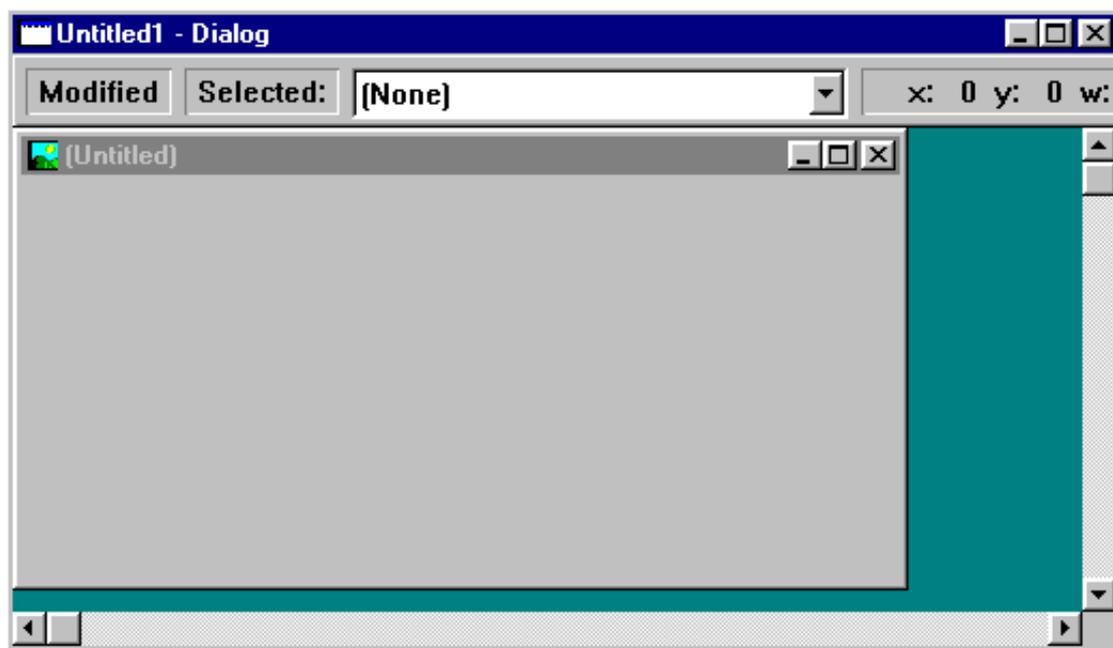
Apart from creating the local data area, this is the minimal number of steps required to create any event-driven application.

Creating a Dialog

▶ To create a new Dialog

1. Invoke Natural.
2. From the Natural menu, select "Object > New > Dialog".

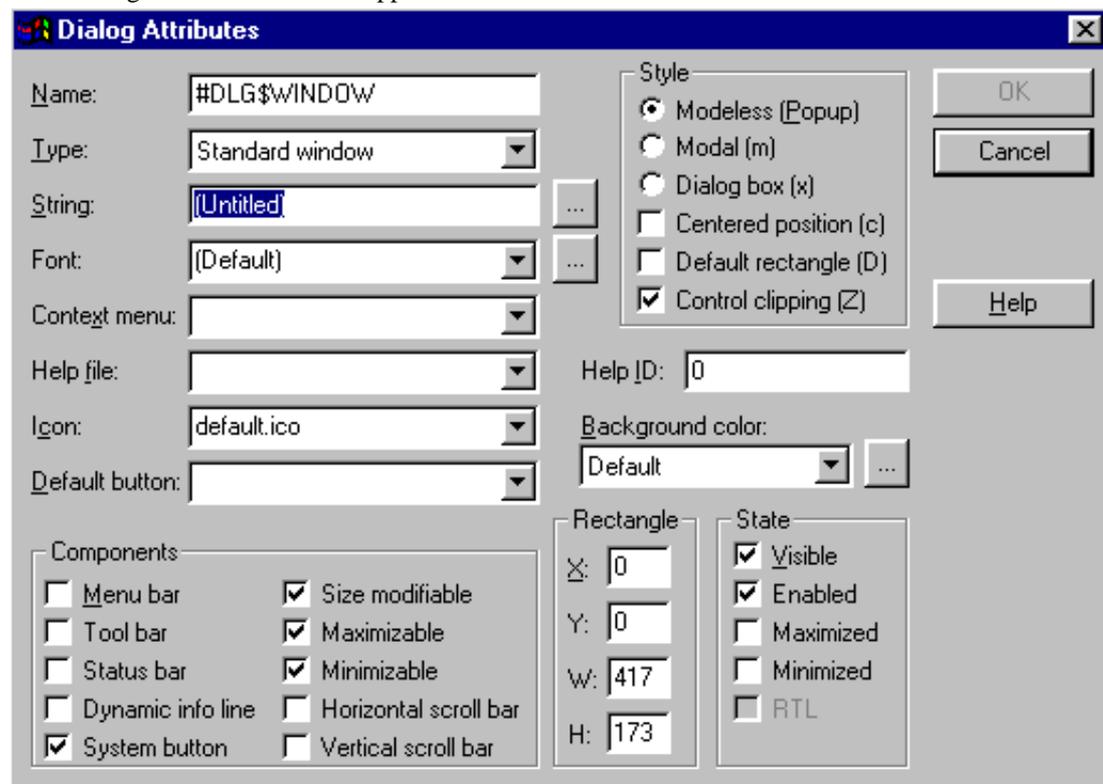
The Natural window displays the dialog editor's menu bar and toolbar. It displays an editing window called "Untitled1-Dialog". You can resize this editing window. The editing window contains the new dialog window, titled "(Untitled)". You can also resize this new dialog window, or use the editing window's scroll bars.



Assigning Attributes to the Dialog

▶ To assign attributes to the dialog

1. Double-click inside the dialog window.
The "Dialog Attributes" window appears.



2. With the cursor in the "String" field, type in the new dialog window's title: "Degressive Depreciation".
3. Open the "Background" selection box by clicking on the down arrow.
A list with predefined colors drops down.
4. Mark the desired color, for example "Gray".
5. Choose OK.

The attributes window closes.

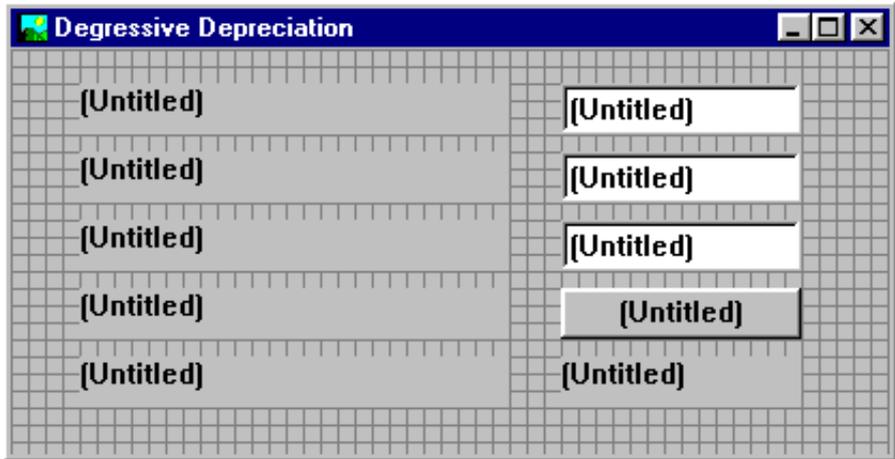
You have set the attribute STRING to the value "Degressive Depreciation" and the attribute BACKGROUND-COLOUR-NAME to the value of your desired color, for example GRAY.

Creating Dialog Elements Inside the Dialog

▶ To create the dialog elements inside the dialog

1. From the menu bar, select "Tools > Options...".
The "Options" dialog box appears. Choose the "Dialog Editor" tab.
2. Choose "Lines".
This decides the way your grid will be displayed.
3. Choose OK to confirm the change.
The grid now helps you position and align the dialog elements.
4. To display the grid from the "Options" dialog, check the 'display grid' option on the "Dialog Editor > tab and choose OK to confirm the change.
5. From the menu bar, select "Insert > Text constant", or click on the toolbar button representing a text-constant control.
6. Move the cursor to the upper left corner of the dialog window.
Ensure that the Editor window's status bar displays an x and a y value of less than 50. Note that at this time, the text-constant control's width and height has an undefined value.
7. Click to fix the text-constant control's position.
A grey rectangle representing the dialog element appears, surrounded by small black squares. At the same time, the status bar indicates that "#TC-1" is selected.
8. Point to one of the small black squares.
The cursor shape now indicates the direction in which you can resize the text-constant control.
9. Resize "#TC-1" to a width of about 200.
10. From the menu bar, select "Edit > Copy" followed by "Edit > Paste" and a new text-constant control "#TC-2" is created on top of "#TC-1". Move the new text-constant control to a position below the first one by clicking and dragging via the mouse or via the keyboard arrow keys with the <SHIFT> held down.
11. Create another three text-constant controls below (in the same way).
12. Create three input-field controls in the upper right corner of the dialog window (by creating the first and duplicating it, as above). These input-field controls should have a height of 36. Align them horizontally with respect to each other and vertically with respect to the upper three text-constant controls.
13. Create a push-button control below the three input-field controls and align it.
14. Create a text-constant control below the push-button control and align it.
15. Align the whole layout until you get a harmonious, well-balanced picture.

Your dialog could now look like this:

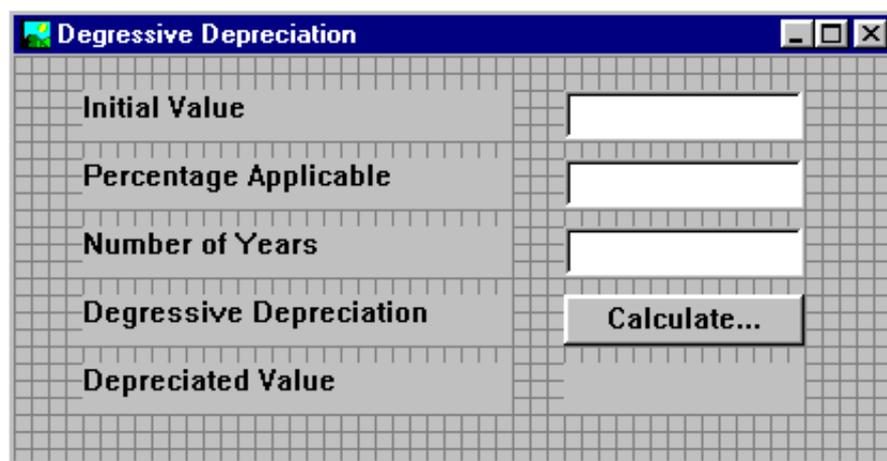


Assigning Attributes to the Dialog Elements

▶ To assign attributes to the dialog elements

1. Double-click on the text-constant control "#TC-1".
The corresponding attributes window appears.
2. In the "String" entry, type in the text string to be displayed: Initial value.
3. Choose OK or press ENTER.
The attributes window closes.
4. Set the following text strings for the four text-constant controls below: Percentage Applicable, Number of Years, Degressive Depreciation, Depreciated Value.
5. From the three input-field controls, remove any text string
6. Set the following text string for the push-button control: "Calculate..."
7. From the last text-constant control, remove any text string.
It will be used for output purposes.

Your dialog should now look like this:



Creating the Application's Local Data Area

The local data area in this application defines the application's linked variables. These linked variables receive the numeric values that the end user has entered in the input-field controls. The variables and their values are used in the calculation of the push-button control's click event handler code.

▶ To prepare the creation of your local data area, your input-field controls must use linked variables

1. Double-click on the first input-field control "#IF-1".
The corresponding attributes window appears.
2. Click on the "..." push-button to the right of the "String" entry.
The "Source for #IF-1.STRING" dialog box appears.
3. In the "Attribute Source" group frame, click (and enable) the "Linked variable" radio button
4. In the "Variable Name" entry, enter: #INITIAL-VALUE.
5. Choose OK twice to leave both the "Source for #IF-1.STRING" dialog box and the attributes window.
6. Set the following linked variable names for the remaining two input-field controls: #PERC-APPLIC, #YEAR-NUM.

▶ To create the application's local data area

1. From the menu bar, select "Dialog > Local data area...".
The "Dialog Local Data Area" definition section appears.
2. Define your local data as follows:
 - 1 #INITIAL-VALUE (N6.2)
 - 1 #PERC-APPLIC (N2.1)
 - 1 #YEAR-NUM (N2)
3. Choose OK.

Natural will now be able to process the input data.

Attaching Event Handler Code to the Dialog Element

▶ To attach event handler code

1. Select the push-button control labelled "Calculate...".
2. From the menu bar, select "Control > Event Handlers...". The corresponding Event handler definition section appears.
The "Click" Event is preselected: when the end user clicks on this push-button control, the specified Natural code will be triggered.
3. In the event handler editing area, enter the following Natural code in free form:


```
#RESULT:= #INITIAL-VALUE * ( ( ( 100 - #PERC-APPLIC )
/ 100 ) ** #YEAR-NUM )
MOVE EDITED #RESULT (EM=Z(5)9.99) TO #TC-6.STRING
```
4. Choose OK to close the editing area, and choose OK again to close the attributes window.

Checking, Stowing and Running the Application

▶ To check the application for syntax errors

1. From the menu bar, select "Object > Check".
A dialog box comes up with a Natural error: a variable needs to be declared.
2. In the dialog box, select the "Edit" push button.
The dialog's code is displayed, the cursor pointing to the error.
3. Select "Cancel".
4. Select "Dialog > Local data area"
5. Add the definition "1 #RESULT (N6.2)".
6. Select OK.
7. Check your application again.

The Information message box should now confirm that the check was successful.

▶ To stow your application

1. From the menu bar, select "Object > Stow".
The "Stow Dialog As" dialog box appears.
2. Enter the name "Degrdep".
3. From the "Libraries" list box, select the library where you want the dialog to be stowed.
4. Choose OK.

The Information message box now confirms that the dialog was stowed successfully.

▶ To test your application

From the menu bar, select "Object > Run".

Basic Terminology

Event-driven Natural uses the following basic terminology:

Attribute

A property of a dialog or a dialog element which can assume specific values. Example: If the HAS-STATUS-BAR attribute is set to TRUE for a dialog, then the dialog contains a status bar. The following operations may be made on attributes:

Operation	Result
Query	In event handler code, you can query an attribute's value at runtime. Example: #L:= #DLG\$WINDOW.HAS-STATUS-BAR
Set	In event handler code, you can set an attribute to a value in the global attribute list before you create a dialog element dynamically. Example: #PUSH.STYLE:= 'O' PROCESS GUI ACTION ADD WITH #W PUSHBUTTON #PUSH
Modify	In event-handler code, you can modify an attribute value of an existing dialog element at runtime. Example: #PUSH.STYLE:= 'C'

Base Dialog

This is the main dialog of an application. It is started from the command line or via the object list. When this dialog is closed, all other dialogs of the application are closed as well.

Control

A type of dialog element. Examples: edit-area control, push-button control, list-box control.

Dialog

A Natural object similar to a map or a program that represents a window in an event-driven application, plus all event handlers and attributes directly attached to the window. It can be a window, a modal window, a dialog box, an MDI child window, and an MDI frame window. The window as such is identified by its handle, the whole dialog is represented by the value of the system variable *DIALOG-ID.

Dialog Box

A special kind of dialog that is exclusively processed in an application. While this dialog is active, all other dialogs of the application are disabled and do not accept any user input. If a dialog invokes a dialog box with an OPEN DIALOG statement, the dialog returns from the OPEN DIALOG statement only after the dialog box is closed. This allows the application to return parameters from the dialog box to the dialog.

Dialog Editor

The Natural editor with which you create and maintain dialogs.

Dialog Element

Dialog elements are (in most cases) graphical elements inside a window that enable the end user to interact with the event-driven application. After a dialog has been created, and its attributes have been set, the programmer places the dialog elements inside the window; usually, this comprises a menu control, possibly a toolbar, and other elements, such as push-button controls and input-field controls. There are two types of elements: controls and items.

Event

Occurs when a user interacts with a dialog element. An event may also be sent from within a piece of code (user-defined event). Example: a click event occurs when the user mouse-clicks on a push-button control for which a piece of click event handler code has been specified. The system variable *EVENT contains the event name.

Event Handler

Programming code that is connected with a dialog element, and is triggered when a particular type of event occurs.

Handle

Identifies a dialog element in code and is stored in handle variables. Example: #PB-1.

Item

A type of dialog element that is part of a control. Example: selection-box item, which is part of a selection-box control.

MDI - Multiple Document Interface

Allows an application to manage several different documents or several views of the same document within the main application window (MDI frame window). These views or documents are displayed in separate MDI child windows.

MDI Child Window

Displays a view of a document within the MDI frame window of an MDI application.

MDI Frame Window

The parent window to all other child (document) windows in an MDI application.

Modal Window

Similar to a dialog box, except that if a dialog invokes a modal window with an OPEN DIALOG statement, the dialog returns from the OPEN DIALOG statement immediately after the modal window has completed opening.

SDI - Single Document Interface

As opposed to MDI applications, SDI applications do not have an MDI frame window that contains the document windows. Only a single view of a single document is displayed.

Popup

A dialog with style "Popup" is modeless and can be moved anywhere on the desktop.

Window

The basic type of window.