Natural Studio Interfaces

This chapter covers the following topics:

Root Interface

Plug-ins access Natural Studio functionality through an Automation interface. All individual interfaces that form the Natural Studio Automation interface can be reached from the root interface, `INatAutoStudio`. A handle to this interface is passed to each plug-in during activation and deactivation.

Interface Structure

The following tree diagram shows the hierarchical structure of the Automation interface:

```
INatAutoStudio
  └── INatAutoObjects
          └── INatAutoDataAreas
                  └── INatAutoDataArea
          └── INatAutoDialogs
                  └── INatAutoDialog
          └── INatAutoPrograms
```

```
                └── INatAutoProgram
        └── INatAutoGenericDocuments
                └── INatAutoGenericDocument
        └── INatAutoGenericTexts
                └── INatAutoGenericText
        └── INatAutoObjectLists
                └── INatAutoObjectList
        └── INatAutoObjectTrees
                └── INatAutoObjectTree
                        └── INatAutoObjectTreeNode
        └── INatAutoSelectedObjects
                └── INatAutoSelectedObject
        └── INatAutoRefreshObject
└── INatAutoControlBars
        └── INatAutoImages
        └── INatAutoCommands
                └── INatAutoCommand
        └── INatAutoToolBars
                └── INatAutoToolBar
        └── INatAutoFrameMenus
                └── INatAutoFrameMenu
                        └── INatAutoPopupMenu
        └── INatAutoContextMenus
                └── INatAutoContextMenu
                        └── INatAutoPopupMenu
└── INatAutoTypes
        └── INatAutoNodeTypes
                └── INatAutoNodeType
        └── INatAutoNodeImages
└── INatAutoPlugIns
        └── INatAutoPlugIn
└── INatAutoResultViews
        └── INatAutoResultView
└── INatAutoSystem
        └── INatAutoEnvironments
                └── INatAutoEnvironment
                        └── INatAutoNatparm
                        └── INatAutoNatsvar
        └── INatAutoApplications
                └── INatAutoApplication
                        └── INatAutoLinkedApplications
                INatAutoEnvironment
        INatAutoSysmain
    INatAutoProgressIndicator
```

Working with Control Bars

Natural Studio users access plug-in functionality by using commands. A plug-in identifies each command by a number. The number can be freely chosen, but must of course be unique per plug-in. A command can have a caption and an image assigned. Caption and image represent the command in menus and toolbars.

To provide a command to the user, a plug-in first creates an `INatAutoCommand` interface in the `INatAutoCommands` collection:

```
send "Add" to #commands
with 4711 "MyCommand" #myImage
return #myCommand
```

While creating the command, in the above example the plug-in refers to an image `#myImage`. This image is used to represent the command visually in menus and toolbars. The plug-in may have loaded the image before, using the method `INatAutoImages::LoadImage`:

```
send "LoadImage" to #images
with "e:\images\myimage.bmp"
return #myImage
```

This results in an `IPictureDisp` interface that can be passed to the method `INatAutoCommands::Add`. The `IPictureDisp` interface is a predefined interface in Windows. An `IPictureDisp` interface can be created in Natural using the method `INatAutoImages::LoadImage`.

Alternatively, the plug-in can pass the image file name directly to the method `INatAutoCommands::Add`:

```
send "Add" to #commands
with 4711 "MyCommand" "e:\images\myimage.bmp"
return #myCommand
```

When the user later chooses the command in a menu or toolbar, the plug-in is notified by using the method `INaturalStudioPlugIn::OnCommand`.

But in order to make the command accessible to users in the first place, the plug-in must insert it into a menu or toolbar. We show this with a toolbar as example. Here the plug-in first locates the Tools toolbar. Then it inserts the previously created command into the toolbar.

```
send "Item" to #toolbars
with "Tools"
return #toolsToolbar
send "InsertCommand" to #toolsToolbar
with #myCommand
```

The plug-in needs to create the command only once and can then assign it to different toolbars or menus.

The plug-in might as well create its own toolbar and add the command to this toolbar:

```
send "Add" to #toolbars
with "MyToolbar"
return #myToolbar
send "InsertCommand" to #myToolbar
with #myCommand
```

We saw the plug-in use the interfaces `INatAutoCommands`, `INatAutoImages` and other interfaces. But how does the plug-in get access to these interfaces in the first place? The plug-in accesses them by querying properties of the root interface, `INatAutoStudio`. A handle to this interface is passed to each plug-in during activation and deactivation. From this interface the plug-in can navigate to any other section of the Natural Studio Automation interface.

To work with control bars, a plug-in uses the interfaces described in the following sections:

```
INatAutoControlBars
INatAutoImages
INatAutoCommands
INatAutoCommand
INatAutoToolBars
INatAutoToolBar
INatAutoContextMenus
INatAutoContextMenu
INatAutoFrameMenus
INatAutoFrameMenu
INatAutoPopupMenu
```

Working with Node Types

Natural Studio frequently uses tree views and list views to display development objects and to navigate through them. Each node in a tree view or list view is characterized by a node type. The node type defines how nodes of a given type are represented in the user interface.

Each node type is identified by an integer number. A development object belonging to a given node type is identified by the number of that node type and by an alphanumeric key. The format of the key varies from node type to node type.

Plug-ins that wish to create their own tree views and list views in Natural Studio can refer to the predefined node types. In addition, plug-ins can define their own node types and can then refer to these user-defined node types.

The following topics are covered below:

- Predefined Node Types
- User-defined Node Types

Predefined Node Types

The built-in Natural Studio development objects such as program, dialog, class, library or application have a predefined node type and key format. Many interfaces and methods in the Natural Studio Automation interface refer to the predefined node types. The full list of available predefined node types and the format of their keys is defined in the following tables.

Predefined Node Types

Node Type Number	Node Type Name	Key Format
1001	Parameter data area	NATID
1002	Copycode	NATID
1003	DDM	NATID
1004	Global data area	NATID
1005	Helproutine	NATID
1006	Local data area	NATID
1007	Map	NATID
1008	Subprogram	NATID
1009	Program	NATID
1010	Subroutine	NATID
1011	Text	NATID
1012	View	NATID
1013	Dialog	NATID
1014	Class	NATID
1015	Command processor	NATID
1016	Adapt view	NATID
1017	Mainframe DDM	DDMID
1018	Function	NATID
1019	Shared resource	RESID
1020	Error message file	NATID
1021	Adapter	NATID
1051	Parameter data area (in application)	NATID
1052	Copycode (in application)	NATID
1053	DDM (in application)	NATID
1054	Global data area (in application)	NATID

Node Type Number	Node Type Name	Key Format
1055	Helproutine (in application)	NATID
1056	Local data area (in application)	NATID
1057	Map (in application)	NATID
1058	Subprogram (in application)	NATID
1059	Program (in application)	NATID
1060	Subroutine (in application)	NATID
1061	Text (in application)	NATID
1062	View (in application)	NATID
1063	Dialog (in application)	NATID
1064	Class (in application)	NATID
1065	Command processor (in application)	NATID
1066	Adapt view (in application)	NATID
1067	Mainframe DDM (in application)	DDMID
1068	Function (in application)	NATID
1069	Shared resource (in application)	RESID
1070	Error message file (in application)	NATID
1071	Adapter (in application)	NATID
1101	System file	FILEID
1102	Natural system file	FILEID
1103	User system file	FILEID
1104	DDM system file	FILEID
1111	Library	LIBID
1112	Library (in application)	LIBID
1121	Environment	BSTR
1131	Base application	BSTR
1132	Compound application	BSTR
1141	Application server	BSTR

Format NATID

Syntax	Description
name library fnr dbnr	Identifies the Natural development object with the given name in the given library in the given system file. The individual parts of the identifier are separated by spaces.
name library	Identifies the Natural object with the given name in the given library. The system file is then determined from the library according to the usual Natural logic, depending on the library name. The individual parts of the identifier are separated by spaces.
name	Identifies the Natural object with the given name in the current logon library.

Format RESID

Syntax	Description
name/library/fnr/dbnr	Identifies the shared resource with the given name in the given library in the given system file. The individual parts of the identifier are separated by slashes.
name/library	Identifies the shared resource with the given name in the given library. The system file is then determined from the library according to the usual Natural logic, depending on the library name. The individual parts of the identifier are separated by slashes.
name	Identifies the shared resource with the given name in the current logon library.

Format DDMID

Syntax	Description
name fnr dbnr	Identifies the mainframe DDM with the given name in the given `FDIC` system file. The individual parts of the identifier are separated by spaces.
name	Identifies the mainframe DDM with the given name in the current `FDIC` system file.

Format LIBID

Syntax	Description
name fnr dbnr	Identifies the Natural library with the given name in the given system file. The individual parts of the identifier are separated by spaces.
name	Identifies the Natural library with the given name. The system file is then determined from the library according to the usual Natural logic, depending on the library name.

Format FILEID

Syntax	Description
fnr dbnr	Identifies the Natural system file with the given numbers. The individual parts of the identifier are separated by spaces.

Example

Assume that in a certain development environment, we have a system file with the database number "101" and the file number "99", containing a library `MYLIB` with a program `MYPROG`.

In the given environment

- the program is identified by the node type 1009 and the key "MYPROG MYLIB 99 101",
- the library is identified by the node type 1111 and the key "MYLIB 99 101",
- the system file is identified by the node type 1103 and the key "99 101".

User-defined Node Types

Plug-ins can define their own node types. This is useful if a plug-in wants to display tree views or list views of development objects not belonging to the predefined set of Natural Studio objects. An example is the Object Description plug-in. It is also useful for plug-ins that want to display certain aspects of Natural Studio objects not covered by built-in Natural Studio functionality. An example is the XRef Evaluation plug-in.

When defining its own node type, a plug-in is free to choose an arbitrary positive integer value starting with "20000". Values below "20000" are reserved for predefined node types. It does not matter if different plug-ins chose the same integer value for a node type. Internally, Natural Studio distinguishes the node types by their numbers and by the plug-in that defined the node type.

The plug-in is free to define the key format for each user-defined node type. Natural Studio does not interpret the keys of user-defined node types, but treats them as opaque strings.

To define new node types and their visual representations, a plug-in uses the interfaces described in the following sections:

```
INatAutoTypes
INatAutoNodeImages
INatAutoNodeTypes
INatAutoNodeType
```

Working with Selections

Through the interfaces described in this section, plug-ins can access the set of objects the user has currently selected in Natural Studio. A plug-in might need this information to decide if a specific menu or toolbar command is applicable to the current selection and must hence be enabled in the user interface. If the user then executes the command, the plug-in again needs to know the set of selected objects in order to apply the command to each of them. A plug-in has access to the current selection through the interfaces described in this section.

In order to work with the current selection, a plug-in starts with the root interface `INatAutoStudio`, retrieves the `INatAutoObjects` interface and then the `INatAutoSelectedObjects` interface. We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#objs := #studio.Objects
#selobjs := #objs.SelectedObjects
```

The returned `INatAutoSelectedObjects` interface gives access to the set of objects the user has currently selected. Using this interface, the plug-in can, for instance, iterate across the selected objects and inspect them. The method `Item` returns an `INatAutoSelectedObject` interface to a specific selected object.

```
#iCount := #selobjs.Count
for #i := 1 to #iCount
  send "Item" to #selobjs
  with #i return #selobj
  #iType := #selobj.Type
  #aKey := #selobj.Key
end-for
```

The property `FocusObject` returns the index of the specific selected object that currently has the focus. This index can be used to retrieve the `INatAutoSelectedObject` interface of the focus object.

```
#iFocus := #selobjs.FocusObject
send "Item" to #selobjs
with #iFocus return #focus
```

The method `ContainsObjectType` can be used for a quick check if the current selected set contains objects of a specific type. This might be sometimes sufficient to decide if a specific command shall be enabled or not.

```
send "ContainsObjectType" to #selobjs
with 1009 return #bContainsPrograms
```

For specific checks the plug-in can also retrieve and process the current selection as an XML document.

```
#aSelectedObjectsXML := #selobjs.SelectedObjects
```

To work with selections, a plug-in uses the interfaces described in the following sections:

`INatAutoObjects`
`INatAutoSelectedObjects`
`INatAutoSelectedObject`

Working with Natural Development Objects

Through the interfaces described in this section, plug-ins can create and edit Natural development objects. Being able to create new development objects, load existing objects into an editor, manipulate their contents and to save and stow them, enables plug-ins to provide generation functions and thus to help automating the development process. An example is the Program Generation plug-in.

To open a program in the program editor, for instance, a plug-in starts with the root interface `INatAutoStudio`, retrieves the `INatAutoObjects` interface and then the `INatAutoPrograms` interface. Now it uses the method `INatAutoPrograms::Open` to load the program into the editor.

We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#objects := #studio.Objects
#programs := #objects.Programs
send "Open" to #programs
with 1009 "MYPGM" "MYLIB" return #program
```

The resulting `INatAutoProgram` interface can now be used to operate on the program, for instance, to stow it and then to close the editor.

```
send "Stow" to #program
send "Close" to #program
```

A new program source is created by using the method `Add`.

```
send "Add" to #programs
with 1009 return #program
```

Source code is added to the program either as a whole by using the property `Source`:

```
#program.Source := "WRITE ""HELLO, WORLD!"" END
```

Or incrementally by using the method `InsertLines`.

```
send "InsertLines" to #program
with "WRITE ""HELLO, WORLD!" #return #next
send "InsertLines" to #program
with "END" #next return #next
```

The interface `INatAutoProgram` provides also search and replace methods and other methods to modify the source code.

```
send "Search" to #program
with "HELLO" return #found
send "ReplaceLines" to #program
with "WRITE ""Good morning" #found
```

Dialogs and data areas are accessed in a similar way by using the interfaces `INatAutoDialog` and `INatAutoDataArea`. But there is one particularity with these objects: Even though there is a graphical or structured editor in the user interface for these objects, they are edited textually through the Automation interface. Applied to data areas this means: If a plug-in wants to generate a data area, it actually has to generate a `DEFINE DATA` statement.

```
#objects := #studio.Objects
#dataareas := #objects.DataAreas
send "Add" to #dataareas
with 1006 return #lda
*
send "StartEdit" to #lda
send "InsertLines" to #lda
with "DEFINE DATA LOCAL" return #next
send "InsertLines" to #lda
with "1 MYSTRING(A10)" #next return #next
send "InsertLines" to #lda
with "1 MYNUMBER(I4)" #next return #next
send "InsertLines" to #lda
with "END-DEFINE" #next return #next
send "EndEdit" to #lda
*
send "Stow" to #lda
send "Close" to #lda
```

The calls to the methods `INatAutoDataArea::StartEdit` and `INatAutoDataArea::EndEdit` are used to mark the beginning and end of a series of editing operations.

To work with Natural development objects, a plug-in uses the interfaces described in the following sections:

`INatAutoObjects`
`INatAutoPrograms`
`INatAutoProgram`
`INatAutoDialogs`
`INatAutoDialog`
`INatAutoDataAreas`
`INatAutoDataArea`

Working with Generic Text Documents

Through the interfaces described in this section, plug-ins can use the Natural Studio program editor as editor for arbitrary text objects. A plug-in can open a program editor session, pass a buffer with text data to it, let the user edit the data and then retrieve the modified data back. The plug-in itself is responsible for providing and storing the data to be edited. The program editor provides the usual editing functions, as far as they are appropriate for generic text objects.

To let the user edit a given text in the program editor, a plug-in starts with the root interface `INatAutoStudio`, retrieves the `INatAutoObjects` interface and then the `INatAutoGenericTexts` interface. Now it uses the method `INatAutoGenericTexts::Open` to load the text buffer into the editor.

We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`. Also we assume that the text to be edited is contained in the alphanumeric variable `#buffer`.

```
#objects := #studio.Objects
#texts := #objects.GenericTexts
send "Open" to #texts
with "Curriculum Vitae" "Dana Scully" #buffer
return #text
```

The editor is then opened and the user can edit the text interactively.

The plug-in can use the resulting `INatAutoGenericText` interface to operate on the text, for instance, to insert lines:

```
send "InsertLines" to #text
with "Taught for two years at Quantico Medical School"
```

If the user chooses the **Save** button in the editor, the plug-in receives the notification `PLUGIN-NOTIFY-SAVE`. In response to this notification, it will usually retrieve the edited text from the editor and save it.

```
#buffer := #text.Source
* Now save the text in a plug-in specific way.
```

To work with generic text documents, a plug-in uses the interfaces described in the following sections:

`INatAutoGenericTexts`
`INatAutoGenericText`
`INatAutoObjects`

Working with Generic Documents

A plug-in that maintains own development objects might want to provide its own editors for each of its development object types. Editors in Natural Studio typically maintain development objects in MDI (Multiple Document Interface) windows. In the following, we call them document windows. Natural Studio has a number of built-in editors, for instance, the program editor and the dialog editor. A plug-in can implement its own editor with a so-called generic document window.

Implementing such an editor as a generic document window makes the editor behave like the built-in editors in Natural Studio. Essentially this means: Several editor windows on different objects can be opened in parallel and the user can switch between them.

In order to implement a generic document window, you first create a Natural dialog of type "Plug-in MDI window". In your plug-in code, you can then open this dialog with the `OPEN DIALOG` statement and let Natural Studio display the dialog as a document window. Normally you will do this in the command handler of your plug-in, that is in the method `INaturalStudioPlugIn::OnCommand`:

```
open dialog "mydlg" null-handle giving #dialogid
#objects := #studio.Objects
#genericdocs := #objects.GenericDocuments
send "Add" to #genericdocs with #dialogid return #doc
```

The resulting `INatAutoGenericDocument` interface can now be used to operate on the document window.

The plug-in has several other means to communicate with the Natural dialog contained in the generic document window:

- The plug-in can send events to the dialog with the `SEND EVENT` statement and using the dialog ID.
- The dialog can send method calls to the plug-in. To achieve this, the plug-in should pass its own `*THIS-OBJECT` handle to the dialog in the `OPEN DIALOG` statement.
- The dialog can call the Natural Studio Automation interface. To achieve this, the plug-in should pass the `INatAutoStudio` interface pointer to the dialog in the `OPEN DIALOG` statement.

Whenever the user activates a document window, Natural Studio automatically switches the frame menu to a menu that contains the commands applicable to the active document. In the case of built-in document windows, these frame menus are predefined. In the case of a generic document window, the plug-in itself can provide an appropriate frame menu.

The plug-in can create a frame menu of its own by cloning an existing frame menu using the method `INatAutoFrameMenus::Clone` and adding new commands to the clone as necessary using the method `INatAutoFrameMenu::InsertCommand`.

Afterwards it passes the resulting `INatAutoFrameMenu` interface to Natural Studio when calling the method `INatAutoGenericDocuments::Add`.

To work with generic documents, a plug-in uses the interfaces described in the following sections:

`INatAutoGenericDocuments`
`INatAutoGenericDocument`
`INatAutoObjects`

Working with Tree Views and List Views

Through the interfaces described in this section, a plug-in can display its own tree views and list views in Natural Studio. Tree views and list views are frequently used in Natural Studio to display development objects and to navigate through them.

In order to display objects in tree views and list views, the plug-in must first register the types of the tree or list view nodes that it is going to display. This procedure is described in *Working with Node Types*.

A plug-in that displays objects in tree views and list views must also implement the methods of the interface `INaturalStudioPlugInTree` appropriately. Natural Studio calls the methods of this interface when expanding or refreshing the tree.

In order to open a tree view, the plug-in starts with the root interface `INatAutoStudio`, retrieves the `INatAutoObjects` interface and then the `INatAutoObjectTrees` interface. We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#objs := #studio.Objects
#trees := #objs.ObjectTrees
```

The resulting `INatAutoObjectTrees` interface gives access to the currently open tree view document windows. Through this interface the plug-in can open a new tree view with a given root object.

```
send "Open" to #trees
with #type #key #caption
return #tree
```

The node type specified in the method `Open` must have been registered before, as described in *Working with Node Types*. The `INatAutoObjectTree` interface returned from the method `Open` gives access to the tree view document window just opened. Through this interface the plug-in can, for instance, later close the document window.

```
send "Close" to #tree
```

When opening a tree view, the plug-in specifies at least the type and key of the root object and a caption to be displayed on the tree view document window. Natural Studio will retrieve additional information needed to expand the tree view by using the interface `INaturalStudioPlugInTree` that must be implemented by the plug-in.

The nodes of a tree view can be accessed through the interface `INatAutoObjectTreeNode`. The root node of a tree view is retrieved with the method `INatAutoObjectTree::GetRootNode`, which returns an interface `INatAutoObjectTreeNode`. This interface can then be used, for instance, to expand the node and to access the child nodes. In the same way, the currently selected node of a tree view can be retrieved.

```
send "GetRootNode" to #tree
return #rootnode
send "Expand" to #rootnode
send "GetChild" to #rootnode
return #firstchildnode
send "GetNext" to #firstchildnode
return #nextchildnode
send "Expand" to #nextchildnode
send "GetSelectedNode" to #tree
return #selectednode
send "Expand" to #selectednode
```

The interface `INatAutoObjectTreeNode` controls only the visual appearance of an individual tree view, not the underlying object structure, which is possibly represented differently in several views at a time. The object structure itself is under the control of the plug-in that defines and provides it through its `INaturalStudioPlugInTree` interface.

List view document windows are created in a similar way as tree view document windows, except that the interface `INatAutoObjectLists` is used instead of `INatAutoObjectTrees`.

To work with tree views and list views, a plug-in uses the interfaces described in the following sections:

`INatAutoObjects`
`INatAutoObjectTrees`
`INatAutoObjectTree`
`INatAutoObjectTreeNode`
`INatAutoObjectLists`
`INatAutoObjectList`
`INatAutoRefreshObject`

Working with Result Views

Through the interfaces described in this section, a plug-in can display the results of its work in the Natural Studio result view. Objects displayed in a result view can be target of commands and can be used as starting point for navigation. Examples of built-in functions that use result views are the **Cat All** command and the **Find** command.

In order to display objects in result views, the plug-in must first register the types of nodes that it is going to display. This procedure is described in the section *Working with Node Types*.

In order to work with result views, the plug-in starts with the root interface `INatAutoStudio` and retrieves the `INatAutoResultViews` interface. We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#resultviews := #studio.ResultViews
```

The resulting `INatAutoResultViews` interface gives access to the result view control bar and the currently open result views. The plug-in can use this interface, for instance, to show or hide the result view control bar.

```
send "Show" to #resultviews
```

Through the interface `INatAutoResultViews` the plug-in can open a new result view.

```
send "Open" to #resultviews
with #caption #image #headers
return #resultview
```

When opening a result view, the plug-in specifies a caption and an image to be displayed on the result view tab and (if needed) column headers for the result view.

The `INatAutoResultView` interface returned from the method `Open` gives access to the result view just opened. Through this interface the plug-in can activate the result view, insert rows into it and update the display. The method `SetVisible` scrolls a specific row into view.

```
#resultview.Active := true
send "InsertRows" to #resultview
with #rows return #last
send "Update" to #resultview
send "SetVisible" to #resultview
with #last
```

Finally the plug-in can close its result view.

```
send "Close" to #resultview
```

To work with result views, a plug-in uses the interfaces described in the following sections:

`INatAutoResultViews`
`INatAutoResultView`

Working with Environments

Through the interfaces described in this section, plug-ins can inspect the available local and remote development environments, map environments, connect to and disconnect from a remote environment and activate an environment.

In order to work with environments, a plug-in starts with the root interface `INatAutoStudio`, retrieves the `INatAutoSystem` interface and then the `INatAutoEnvironments` interface. We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#system := #studio.System
#envs := #system.Environments
```

The returned `INatAutoEnvironments` interface gives access to the local environment and all remote environments that have once been connected during the current Natural Studio session.

Through this interface the plug-in can, for instance, map a new remote environment, specifying host name, port number, user ID, password and other arguments.

```
send "Add" to #envs
with "IBM2" "4712" "SCULLY" "secret" "STACK=(LOGON XFILES)"
return #env
```

The returned interface `INatAutoEnvironment` gives access to attributes of the environment.

```
#bIsActive := #env.Active
#bIsConnected := #env.Connected
```

The property `Parameters` gives access to the interface `INatAutoNatparm`. This interface contains properties that represent the Natural parameters under which the environment is running. Only a subset of the Natural parameters is available through this interface.

```
#natparm := #env.Parameters
#fuserDBnr := #natparm.FuserDBnr
#fuserFnr := #natparm.FuserFnr
```

The property `SystemVariables` gives access to the interface `INatAutoNatsvar`. This interface contains properties that represent the system variables currently set in the environment. Only a subset of the system variables is available through this interface.

```
#natsvar := #env.SystemVariables
#language := #natsvar.Language
```

The plug-in uses the method `Disconnect` to disconnect from the remote environment.

```
send "Disconnect" to #env
```

To work with environments, a plug-in uses the interfaces described in the following sections:

`INatAutoEnvironments`
`INatAutoEnvironment`
`INatAutoSystem`
`INatAutoNatparm`

INatAutoNatsvar

Working with Applications

Through the interfaces described in this section, plug-ins can inspect the applications available on the application server, map applications into the Natural Studio session, connect to and disconnect from an application, activate an application and create and modify applications. An example of a plug-in that uses this section of the interface is the Application Wizard.

In order to work with applications, the plug-in starts with the root interface `INatAutoStudio`, retrieves the `INatAutoSystem` interface and then the `INatAutoApplications` interface. We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#system := #studio.System
#apps := #system.Applications
```

The resulting `INatAutoApplications` interface gives access to the currently active application server and the applications it contains. Through this interface the plug-in can, for instance, ask for the currently active application.

```
#app := #apps.ActiveApplication
```

The plug-in can also create a new application and map it into the Natural Studio session.

```
send "Add" to #apps
with "MYAPPLICATION" return #app
```

The resulting `INatAutoApplication` interface gives access to attributes of the application.

```
#bIsActive := #app.Active
#bIsConnected := #app.Connected
```

For a compound application, the property `LinkedApplications` returns the interface `INatAutoLinkedApplications`. This interface allows accessing the base applications that are linked to the compound application.

```
#linkedapps:= #app.LinkedApplications
#iCount := #linkedapps.Count
```

For a base application, the property `LinkedObjects` returns an XML document containing the list of objects linked to the application.

```
#aObjects:= #app.LinkedObjects
```

The plug-in can also link and unlink objects to and from the application.

```
send "UnlinkObject" to #app
with 1009 "OLDPROG" "MYLIB"
send "LinkObject" to #app
with 1009 "NEWPROG" "MYLIB"
```

Finally the plug-in can disconnect and unmap the application.

```
send "Disconnect" to #app
send "Unmap" to #app
```

To work with applications, a plug-in uses the interfaces described in the following sections:

`INatAutoApplications`
`INatAutoApplication`
`INatAutoLinkedApplications`

Working with Plug-ins

Through the interfaces described in this section, a plug-in can inspect the currently installed plug-ins, read their properties and activate or deactivate a plug-in. This includes the possibility that a plug-in deactivates itself. An example for a plug-in that uses this section of the interface is the Plug-in Manager.

In order to work with plug-ins, the plug-in starts with the root interface `INatAutoStudio` and retrieves the interface `INatAutoPlugIns`. We assume here that the plug-in has kept a handle to the interface `INatAutoStudio` in a variable named `#studio`. This handle was passed to the plug-in during activation, in the method `INaturalStudioPlugIn::OnActivate`.

```
#plugins := #studio.PlugIns
```

The resulting `INatAutoPlugIns` interface gives access to the currently installed plug-ins. Using this interface, the plug-in can, for instance, iterate across the installed plug-ins and inspect their attributes. The method `Item` returns an `INatAutoPlugIn` interface to a specific plug-in.

```
#iCount := #plugins.Count
for #i := 1 to #iCount
  send "Item" to #plugins
  with #i return #plugin
  #aName := #plugin.Name
  #aProgID := #plugin.ProgID
  #bIsActive := #plugin.Active
end-for
```

Through the interface `INatAutoPlugIn` the plug-in can also activate or deactivate a specific plug-in by modifying the property `Active`. The following sample toggles the activation state of a plug-in.

```
#bIsActive := #plugin.Active
if #bIsActive
  #plugin.Active := false
else
  #plugin.Active := true
end-if
```

The interface `INatAutoPlugIn` can be used to send a command to the plug-in. The following sample checks whether the plug-in command with the ID "200" is currently enabled and if so, lets the plug-in execute the command. Of course this requires that we know that the plug-in implements a command with the ID "200" and what this command does.

```
#bEnabled = false
#bChecked := false
send "OnCommandStatus" to #plugin
with 200 #bEnabled #bChecked
if #bEnabled
  send "OnCommand" to #plugin with 200
end-if
```

Through the interface `INatAutoPlugIn`, the plug-in can get access to arbitrary services that another plug-in provides with a so-called custom interface. The following sample retrieves the custom interface of a plug-in and calls one of its services. Of course this requires that the plug-in has documented the services it provides with its custom interface.

```
#icustom := null-handle
send "GetCustomInterface"
to #plugin return #icustom
if #icustom ne null-handle
  #result := 0
  send "GetMaritalStatus" to #icustom
  with "Anderson, Gillian" return #result
end-if
```

In order to provide a custom interface, a plug-in must implement an additional interface beside the two predefined interfaces `INaturalStudioPlugIn` and `INaturalStudioPlugInTree` and make this interface the default dispatch interface. For a plug-in implemented in Natural this means placing this interface at the first position in the `DEFINE CLASS` statement.

```
define class ...
  object using ...
  id "..."
  interface icustom
    id "..."
    method GetMaritalStatus id 1 is gstat-n
      parameter using gstat-a
    end-method
  end-interface
  interface using nstplg-i
  interface using nstplt-i
end-class
end
```

To work with plug-ins, a plug-in uses the interfaces described in the following sections:

`INatAutoPlugIns`
`INatAutoPlugIn`

Working with Dialogs

If a plug-in wants to open dialog boxes in Natural Studio, some special considerations have to be taken. The support of dialog boxes in plug-ins depends mainly on the condition if the plug-in is running in as an in-process ActiveX component or if it is running in a separate process.

The following topics are covered below:

- Plug-ins Running in a Separate Process
- Plug-ins Running In-process

Plug-ins Running in a Separate Process

In general, an ActiveX component running in a separate process cannot open a dialog box in the client process. This restriction of the Windows system itself is overcome in the special case that a plug-in is written in Natural.

If a plug-in is written in Natural, it can open modal dialog boxes in Natural Studio. Precisely this means: Natural dialogs that are defined in the dialog editor with the **Type** attribute set to "Standard window" and the **Style** attribute set to "Dialog box". Other styles of the dialog type "Standard window" cannot be used in plug-ins.

To open a dialog box, a plug-in uses the usual `OPEN DIALOG` statement.

Plug-ins Running In-process

If a plug-in is implemented as an in-process ActiveX component (this means: as a DLL), it can open modal and non-modal dialogs in Natural Studio. To open a dialog, the plug-in uses the statements usual in the programming language it is written in. Plug-ins written in Natural always run in a separate process, so this applies only to plug-ins written in programming languages that support implementing in-process ActiveX components.

For details on how to implement plug-ins in programming languages other than Natural, see *Developing Plug-ins in Other Programming Languages*.