19

Workspace Manager

Library: None, this API is defined by the Workspace Manager application

Header File Directory: /NextDeveloper/Headers/apps

Import: apps/Workspace.h

Introduction

The Workspace Manager lets the user navigate the file system and manipulate the files and directories therein. Workspace Manager's Inspector panel, with its four displays, gives additional information about a selected item, for example, file ownership and size (Attributes display), applications capable of opening the file (Tools display), and file permissions (Access display). Finally, the Inspector panel's Contents display can give the user information about the contents of certain types of files.

Since there is no limited to the number of file formats, it's impossible for the contents inspector to display the contents of all file types. NeXT provides inspectors for many of the most common formatsÐRTF, TIFF, EPS, to name a fewÐand provides a way for you to install a contents inspector for any other file format. It's even possible to replace a standard inspector with one of your own.

Contents inspectors are stored in bundle files that contain the code and interface objects that are loaded into the Workspace Manager. (For more information on bundles, see the class specification for NXBundle, a common class.)

When the Workspace Manager begins running, it locates contents inspector bundles by scanning the application search path (in this order):

~/Apps
/LocalApps
/NextApps

Finally, it searches in its own application file package, where it finds the standard modules.

For each directory that it searches, the Workspace Manager looks for bundles both within the directory and within application file packages (ª.appº files) in the directory. When it finds a bundle, it checks the executable within it for registry information, information that the Workspace Manager uses to associate an inspector module with a specific file extension. (If more than one module registers for the same file extension, the module later in the search path is ignored.) When the user attempts to inspect the contents of a file, the Workspace Manager consults its registry of file extensions and inspectors and loads the appropriate inspector, if it hasn't been loaded already.

The Workspace Manager application and the contents inspector communicate through the API found in **/NextDeveloper/Headers/apps/Workspace.h**. This API consists of the declaration of the

WMInspector class, an abstract superclass that defines the owner of the inspector module's interface. In creating your own contents inspector, you'll create a subclass of WMInspector and an interface that will appear in the Inspector panel.

The principal messages that the Workspace Manager sends your inspector object are **new** and **revert:**. It sends a **new** message whenever it needs to access the inspector object or, through the object, the interface to the inspector. It sends a **revert:** message whenever the inspector might need to be updated, such as when the selection in the File Viewer has changed. Thus, all inspector objects must implement these methods.

The inspector object, in turn, can query the Workspace Manager for information about the selection. The WMInspector class declares a method (**selectionCount**) for determining whether the selection contains a single item or multiple items, and another (**selectionPathsInto:separator:**) that returns the full path to each item in the selection. Your inspector object, therefore, can access this information by sending itself **selectionCount** and **selectionPathsInto:separator:** messages.

The Components of an Inspector Project

Contents inspectors are created as bundle projects; the process is outlined below. Before looking at the process, let's examine the components that are common to all inspector projects. As an illustration, we'll take the example of an inspector that shows the contents of files containing 3D graphics data in RIB format: ª.ribº files. Even the simplest RIB inspector would have these components:

File	Description
*.lproj/RIBInspector.nib	A nib file containing the user interface to the contents inspector. (The nib file is stored in a language-specific subdirectory, such as **English.lproj**.)
Makefile	The standard makefile for a bundle project
Makefile.preamble	Additional instructions to the **make** utility to load the information in **bundle.registry** into the executable file.
PB.project	The standard project file for a bundle project.
RIBInspector.h	The class interface file for the subclass of WMInspector.
RIBInspector.m	The class implementation file for the subclass of WMInspector.
bundle.registry	A specification file describing which file extension is associated with this inspector, among other things.

Two files of special interest are **bundle.registry** and **Makefile.preamble**.

The bundle.registry File

This file contains the instructions that the Workspace Manager uses to associate a contents inspector with a specific type of file. Using the example of a RIB file inspector, the **bundle.registry** file would look like this:

```
{type=InspectorCommand; mode=contents; extension=rib;
    selp=selectionOneOnly; class=RIBInspector}
```

The registry information consists of a list of key words and their assigned values. Here are the keys and their possible values. (The quotation marks below are for clarity; don't include them in your registry file):

Key	Description
type	The type of registration. For inspector commands, the value must be ªInspectorCommandº.
mode	The mode of the Inspector panel. For Release 3.0, this must be ªcontentsº.
extension	The file extension to be associated with this inspector. (Don't include the ª.º in the extension.) You can only list one extension for each inspector module; wildcard characters aren't permitted.
class	The name of the subclass of WMInspector. In general, an instance of this class owns the nib file that contains the inspector's user interface. Workspace Manager instantiates an object of this class when the inspector is loaded.
selp	The *selection predicate*; that is, the requirements concerning the selection. The value can be either ªselectionOneOnlyº or ªselectionOneOrMoreº.

The **selp** key controls whether your inspector is confined to operating on one file of the given extension at a time (**selectionOneOnly**), or whether it can be displayed if the selection consists of more than one file of the give extension (**selectionOneOrMore**). If you specify **selectionOneOnly** (the usual case), the message ªNo Contents Inspector for Multiple Selectionº appears in the panel when the selection contains multiple files of the given extension.

All six keys must be present in the registry file for your inspector to work properly. The order of these key/value pairs in the registry file isn't important, although the case of the key and value words is.

The Makefile.preamble File

The registry information from **bundle.registry** must be copied into the _ _ICON segment of the module's Mach object file: This is where the Workspace Manager searches for the information. To accomplish this, you have to create a **Makefile.preamble** file with the proper instructions. These instructions are:

```
BUNDLELDFLAGS = -sectcreate __ICON __header bundle.registry
  OTHER_PRODUCT_DEPENDS = bundle.registry
```

The first line instructs the linker to create an **__header** section in the **__ICON** segment of the executable file and to copy the registry information into it. (Note that the prefixes are two underbar characters). The second ensures that **make** will rebuild the project whenever **bundle.registry** is altered.

Building an Inspector Module

The following steps show you the process for assembling an inspector, using the RIB inspector as an example. (These steps won't show you how to do the actual imaging of RIB files. For that, you'll have to look into the 3D Graphics Kit.)

To build the inspector, follow these steps:

1. Start Project Builder and create a new bundle project. In the New Project panel, make sure that the Project Type pop-up list reads ªBundleº. Save the project as **~/RIBInspector**.

2. Start Interface Builder and create a new empty module. (Choose New Empty from the New Module menu). Save the module in **~/RIBInspector/English.lproj/RIBInspector**. When the attention panel appears, confirm that you want to add the nib file to the RIBInspector project.

3. Now, you have to inform Interface Builder of the WMInspector class, as declared in

/NextDeveloper/Headers/apps/Workspace.h. The easiest way to do this is to drag the file icon for **Workspace.h** from the File Viewer into Interface Builder's File window. Interface Builder will parse the header file and insert the WMInspector class in the Classes browser.

4. Declare a subclass of WMInspector by selecting WMInspector in the Classes browser and dragging to Subclass in the Operations browser. Rename this subclass ªRIBInspectorº.

5. Using the Class inspector, add any outlets and actions that you want to the RIBInspector class. For example, in most cases you would add outlets for the text fields that the inspector uses to display information about the selected file. For this illustration, skip this step.

6. Select the File's Owner object in the Objects display of the Files window. Using the Inspector panel, specify that the File's Owner is of the RIBInspector class.

7. Drag a Panel object from the Palettes window into the workspace. This panel will contain the interface to your contents inspector. When the Workspace Manager displays your contents inspector, it will take the Panel's content view and installs it in the view hierarchy of the Inspector panel. For the purposes of this example, drag a Button or two into the panel. Finally, using Interface Builder's Panel inspector, make sure that the panel is not deferred.

8. Connect the **window** outlet of the File's Owner to the Panel. This outlet must be set so that the Workspace Manager can locate the content view to be displayed in the contents inspector. If your File's Owner had other outlets or actions, you would connect them at this point.

9. Switch to the Classes browser in the Files window and select the RIBInspector class. Using the pull-down list, drag to Unparse and confirm that you want to add the class files to the project.

10. Open the class files and implement the **new** method (see the class specification for WMInspector for an example). You must also implement the **revert:** method for your inspector to take any action based on the selection in the File Viewer. For this example, you can omit the **revert:** method. (Note: You will also have to change the #import line at the top of **RIBInspector.h** from #import ªWMInspector.hº to #import <apps/Workspace.h>.)

11. Create **bundle.registry** and **Makefile.preamble** files (as described above) and add them to the Supporting Files suitcase in Project Builder's Files display.

12. Save the project and build it. When done, copy the **RIBInspector.bundle** file into your **Apps** directory.

Registering an Inspector

Workspace Manager must be made aware of this new inspector. If you use Workspace Manager to copy the bundle into **~/Apps** (or anywhere in the application search path, for that matter), it will read the registry information the bundle contains. If you move the file by other means, use Workspace Manager's Update Viewers command to make it recheck for applications and inspectors in the application search path. After Workspace Manager has registered the new inspector, whenever a file of the proper extension (ª.ribº in the example) is selected and the Contents inspector panel is visible, the custom contents inspector will be displayed.

Once an inspector has been loaded, it can't be unloaded without restarting the Workspace Manager (that is, logging out and back in again). For this and other reasons, it's often better to create a test application to debug a new inspector, as discussed in the next section.

Debugging an Inspector

Your inspector operates within the main thread of execution of the Workspace Manager application,

so errors occurring with the inspector can crash the Workspace Manager, bringing down with it all applications launched from the Workspace. Given the severity of the consequences, it's imperative that you ensure the reliability of your inspector's code. Unfortunately, at this time there's no standard way to debug inspectors; you'll have to devise your own test mechanisms.

The best strategy is to create a stand-alone debugging application, one that loads your inspector module into its own window just as the Workspace Manager does. You'll have to create a substitute WMInspector class since the main class in your module must inherit from WMInspector. You could perhaps use an OpenPanel as a means of selecting specific files for your module to inspect.

A debugging application makes it easier, safer, and faster to debug your inspector; however, at times you may find it necessary to debug the inspector after it's been loaded into the Workspace Manager. To do this, you'll need to prevent the inspector's symbol table from being stripped.

By default, when the Workspace Manager loads an inspector bundle, it strips the executable code of its symbols. To prevent this, in a shell window enter:

```
dwrite Workspace StripAfterLoading NO
```

Now, when an inspector is loaded, its symbol data will be preserved. You'll be able to attach to the Workspace Manager process using GDB and trace execution through your inspector's code.

Working Within the Workspace Manager

Some contents inspectors display the actual contents of a file while others show only a synopsis. For example, the contents inspector for TIFF and RTF files shows the complete contents, but the Sound inspector shows only summary information. The Sound inspector, however, does offer the user a button that, when clicked, plays the sound.

Since contents inspectors operate in the Workspace Manager's main thread, it's best to let the user decide whether the panel should embark on time- or resource-intensive operations, as illustrated by the Sound inspector panel. (The RIB inspector example outlined above would no doubt include a Render button.)